Multiple aspects of DEX file for android malware family classification using FCM, FPT, and Decision Trees

A DISSERTATION

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE AWARD OF DEGREE
OF
MASTER OF TECHNOLOGY
IN
COMPUTER SCIENCE & ENGINEERING

Submitted by:

RAJU KUMAR RANJAN
2K18/CSE/501

Guided by:

MR. MANOJ SETHI

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
DELHI TECHNOLOGICAL UNIVERSITY
(Formerly Delhi College of Engineering) Bawana Road,
Delhi-110042
JUNE, 2021

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Bawana

Road, Delhi - 110042

CANDIDATE'S DECLARATION

I, **Raju Kumar Ranjan** , Roll No. 2K18/CSE/501 student of M.Tech (Computer Science Engineering), hereby declare that the project dissertation entitled **"Multiple aspects of DEX file for android malware family classification using FCM, FPT, and Decision Trees"** which is submitted by me to the Department of Computer Science and Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is original and not copied from any source without proper citation. This work has not previously formed the basis for the award of and Degree, Diploma Associateship, Fellowship or other similar title or recognition.

[illegible]

Place: Delhi

Raju Kumar Ranjan

Date: 30/07/2021

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Bawana

Road, Delhi - 110042

CERTIFICATE

I hereby certify that the Project Dissertation titled **"Multiple aspects of DEX file for android malware family classification using FCM, FPT, and Decision Trees"** which is submitted by Raju Kumar Ranjan, 2K18/CSE/501 Department of Computer Science Engineering, Delhi Technological University, Delhi in partial fulfilment of the requirement for the award of the degree of Master of Technology, is a record of the project work carried out by the students under my supervision. To the best of my knowledge this work has not been submitted in part or full for any Degree or Diploma to this University or elsewhere.

[illegible]

Raju Kumar Ranjan
(STUDENT)

[illegible]

Mr. Manoj Sethi
(SUPERVISOR)

DELHI TECHNOLOGICAL UNIVERSITY

(Formerly Delhi College of Engineering) Bawana

Road, Delhi - 110042

ACKNOWLEDGEMENT

The success of a dissertation requires help and contribution from numerous individuals and the organization. Writing the report of this project work gives me an opportunity to express my gratitude to everyone who has helped in shaping up the outcome of the project.

I express my heartfelt gratitude to my project guide Mr. Manoj Sethi for giving me an opportunity to do my dissertation work under his guidance. His constant support and encouragement has made me realize that it is the process of learning which weighs more than the end result. I am highly indebted to the panel faculties during all the progress evaluations for their guidance, constant supervision and for motivating me to complete my work. They helped me throughout by giving new ideas, providing necessary information and pushing me forward to complete the work. I also reveal my thanks to all my classmates and my family for constant support.

[illegible]

Raju Kumar Ranjan

Abstract

Android malware classification and assigning the appropriate android malware family is challenging. Traditional static analysis methods can easily be misguided by malware, and dynamic analysis consumes more space and time. This research proposed a fuzzy-based android malware family classification using multiple aspects of the DEX file. The considered aspects are Permissions of Android application, Image obtained from DEX file sectional features, Dalvik Opcode, and Bytecode of corresponding DEX file. The feature vectors acquired from these multiple aspects are fuzzified using a triangular fuzzifier. The obtained fuzzy sets are classified using an FPT classifier and clustered using Fuzzy C-means. FPT and FCM are combined according to the views, and a Decision Tree model is obtained for classifying the Android malware family. The final model produces an accuracy up to 95.75%.

Contents

List of Figures

List of Algorithms

List of Tables

1 INTRODUCTION

The android operating system evolved based on the Linux open-source kernel. Most of the smartphone and IoT devices are based on the android operating system. The extensive usage of the android operating system prepares a market for android application development and distribution. These android applications are distributed with the help of android application stores such as Google Play, Amazon App store, Xiaomi marketplace, etc. These are the official marketplace, but there are unofficial marketplaces for accessing and installing the android application. Some applications are also distributed directly. Official marketplace provides security tools for scanning and analyzing the maliciousness of published applications on their platforms like Google play store offers "Google Play Protect." Applications installed directly from a third-party store, or direct source may act as malicious applications and harm the user. If the authenticity of the application is not confirmed, then the publisher of the fake application may also steal revenue that is entitled to the original developer. For the quick check of maliciousness of application, the user can apply to the platform like [30], and [15].

According to AV-TEST android malware statistics, as of March 2020, 10.5 million android malware get detected in 2019, and new Android malware samples are growing at the rate of 482,579 per month. This data shows that the android operating system is one of the most attractive targets for android malware developers. This malware consists of various variants of the same android family. Android malware is produced via the piggybacking of the legitimate application with the repackaging of the malicious code. Hence, classifying the android malware based on their family is essential.

In the present scenario, the classification of android malware is based on static, dynamic, or hybrid methods. Android application package (APK) static features like permissions, registered receivers, execution code [24], etc., are used to classify the android malware in traditional static methods. "DroidMoss" proposed in [34] calculates the fuzzy hash of bytecode after decomposing the DEX file of APK. Based on the calculated fuzzy hash, it is determined that the application is repackaged or not, android application maliciousness gets detected. In this API, calls are mainly considered for improving the efficacy of the system. [13] taken the API as a feature set for classifying the samples using machine learning. [17] enhanced the android malware detection by adding frequency and characteristics based sequence of API calls. These methods have great accuracy for determining the maliciousness of android applications but are highly affected by confusion and reinforcement. Techniques in [33, 29, 18] consider android permission as a feature for detecting maliciousness. These are not affected by confusion and reinforcement, but accuracy is not achieved due to minor variations in permissions for the android application. [19] used multiple static features to improve accuracy. [23] combined control flow diagram and [36] added sensitive API with permission feature. In general, static features may classify android malware but have limitations in order to achieve accuracy.

For overcoming the limitation of static analysis, dynamic analysis methods are proposed. [6] uses the system runtime API calls as a feature vector for classifying android malware. [27] proposed android malware classification framework "Andromaly." This framework monitor runtime features and events and uses machine learning algorithms for classification. For effectively detecting android malware, this framework needs sufficient time to collect events and runtime features. In [28, 9, 8] API call graph and frequent subgraph are extracted and used as features for classification of android malware. [4] proposed a hybrid method by integrating the static and dynamic characteristics of android malware—the proposed method, "SAMADroid", consists of a three-layer detection model. The hybrid method effectively introduces the shortcoming of static and dynamic methods like a waste of space and time.

In recent years, image processing methodologies are being widely applied to detect and classify Android malware. Android application is converted in the form of images, and images are used to classify the android malware samples. File visualization technique for visualization of features is used because Android application is a packaged file, and all the logical data is stored in DEX file (classes.dex). This technique does not need to reverse engineer the DEX file for code analysis concerning other visualization techniques. The code analysis includes the knowledge about classes, variables, functions, API calls, etc. This method also can efficiently handle the large volume of Android malware samples.

The majority of the ML-based solutions are biased towards specific features (static, dynamic, and hybrid). The chances of failure increase when android malware mutates itself according to components involved in the target defense system. Therefore AI system based on multiple aspects is an optimal alternative for the android malware classification. The proposed method benefits the features generated on numerous aspects like permission, Dalvik opcode, Bytecode, and transformed images. These views are then used to create corresponding fuzzy (loosely defined) views. These fuzzy views are ensemble using the supervised Fuzzy Pattern tree (FPT) classifier and unsupervised Fuzzy C-means Clustering with Decision Tree Classifier. The proposed work achieves the accuracy of 95.7% for the classification of the Android malware family. The significant contribution in this research are:

- Multiple Views generations that are Permission View, Image View, Dalvik Opcode Frequency View, Dalvik Opcode TF-IDF View, Bytecode View, Bytecode TF-IDF View, from the Android APK file.
- Android DEX executable transformation into images based on its sectional structure.
- Transformation of crisp views to fuzzy views and train the FPT and FCM based models.
- Ensemble the FPT and FCM output dataset and training a Decision Tree for Android Malware Classification

The dissertation is organized as follows. First, provides the background android malware classification problem in Section 2 and reviews the related work in Section 3. Then, in Section 4, the proposed methodology is discussed with all of its internals, including View Generation, Fuzzification of views, FPT classifier, FCM clustering of views, Ensemble both FPT and FCM, and Decision Tree Classifier. Next, in section 5, obtained results are analyzed to present a systematic comparison between single-view and multi-view-based android malware classifiers with a discussion on the proposed approach. Finally, In Section 6, this paper concludes and illustrates the path for future research in android malware classification.

‘

2 BACKGROUND

2.1 Overview of Android OS

Android is the open source software based on Linux. Figure 1 depicts the android software stack and each component assumes that the underlying components are properly secured. Android Platform is having following major components:

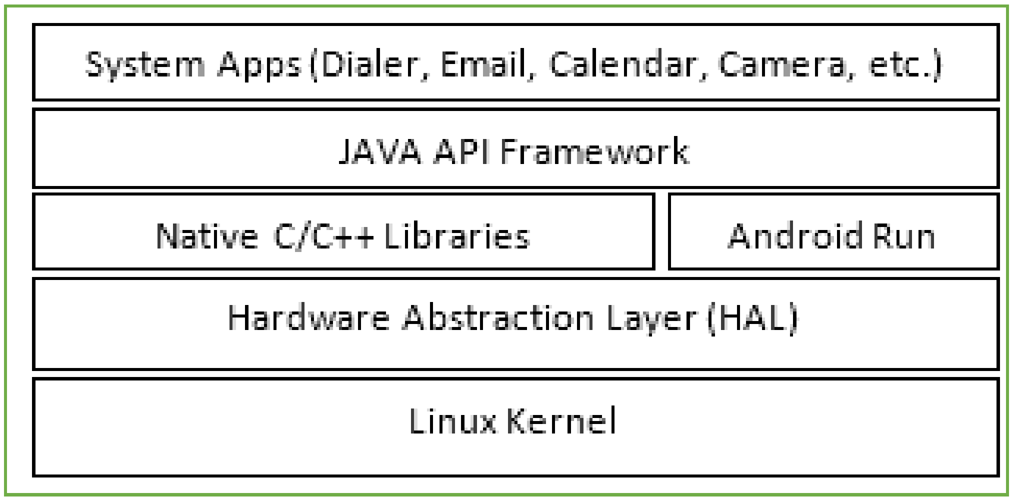

System Apps (Dialer, Email, Calendar, Camera, etc.)
JAVA API Framework
Native C/C++ Libraries
Android Run
Hardware Abstraction Layer (HAL)
Linux Kernel

Figure 1: Overview of Android OS

(A) **System Applications**

The system applications are sets of android APK that comes inbuilt into Android devices. These apps are made for some basic operations such as calendars, email, contacts, internet browsing, SMS messaging, and several added functions. These Apps have special permissions that a user can not change, similar to the user application have access to add or remove permissions. These system applications provide the functionality for a typical user and developers to incorporate or access their features.

(B) **Java API Framework**

As the name it is, the java application programming interfaces are the building blocks that we necessitate to build Android applications by utilizing the reuse of core, modular system components and services that are:

- View System, which is rich and extensible, helps make app's UI that uses lists, grids, text boxes, and an embeddable web browser.
- Resource Manager To access resources such as a string with the localized feature, layout files, and graphics
- Notification Manager To provide users a custom message provided by the Android applications.
- Activity Manager manages the app's lifecycle and produces a common navigation back stack.
- Content Providers provide apps access data in another app like Contacts or share its own data. The Developer has complete access to the corresponding framework APIs that are being used by Android system applications.

(C) **Native C/C++ Libraries**

ART and HAL component of the Android system are developed using native code C and C++. Even Java framework APIs are using some of the functionality in apps. The native android platform libraries are accessed through Android NDK directly from your native code. To make an application using Native Libraries, we need to use Android NDK.

(D) **Android Runtime**

Android has developed Android Runtime, commonly named ART, written to manage various virtual machines on low memory devices by executing Dalvik Executable format (DEX) file. The primary features of ART are following:

- Just-in-time (JIT) compilation
- Ahead-of-time (AOT)
- Optimized garbage collection (GC)

- From Android 9 (API level 28) Android provides transformation of DEX files to more compact machine code commonly known as dex loader.
- Debugging support which including a dedicated sampling profiler, detailed exceptions, and the ability to set Toggle points.

(E) **Hardware Abstraction layer (HAL)**

Android device hardware capabilities are exposed to the JAVA API framework via interfaces that are implemented in Hardware Abstraction Layer (HAL). The various library modules of HAL implements the interface for particular hardware like Bluetooth and Camera. To access hardware components, Java API calls are performed for loading the library module of a hardware component.

(F) **The Linux Kernel**

The heart of the Android operating system is the Linux kernel. Android-based intelligent devices would not be capable of working without Linux Kernel. It provides an interface between user-space applications and physical hardware components. It emphasizes the isolation between processes and governs to manage privileges of a particular process. The kernel is monolithic. Android developers made numerous modifications and implemented new Linux kernel modules: Binder,wake-locks, ashmem, RAM_CONSOLE, pmem, yaffs2, logger, timed output/gpio, oom modifications, Alarm Timers, and Paranoid Networking.

2.2 Android APK File Structure

Android applications are compressed into APK files which are used for distribution. APK files are primarily ZIP files such as JAR files which use Java libraries. Android APK file comprises executable data in the form of DEX executables, resources, native libraries, assets, etc. A digitally signed APK is necessary with a certificate of publisher for publishing it on any android marketplace. An APK comprises with following directories and files:

- Assets: It contains the assets of the Android APK.
- res: This folder contains all resource files that are not compiled into the resource.arsc. All XML files contain res/values from all the APK components.

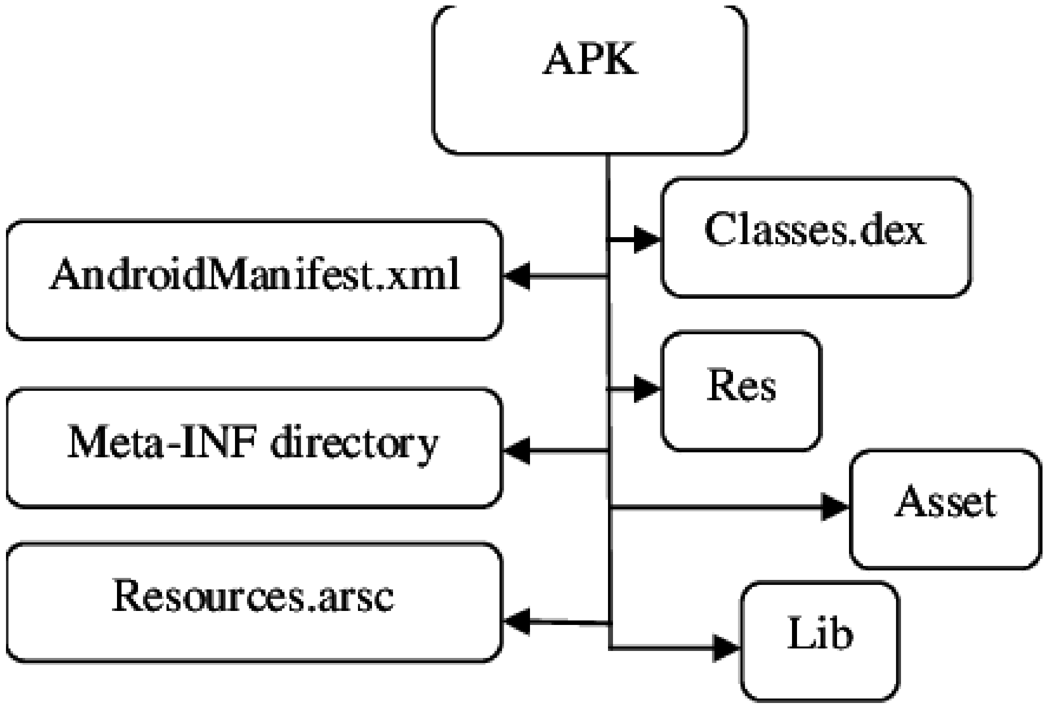


Figure 2: Android APK Structure

- lib: This folder has multiple directories according to CPU architecture. Each separate directory contains the natively compiled libraries accordingly.
- META-INF: This folder contains the signature and metadata of the Android Application.
- AndroidManifest.xml: Manifest file is the main index of Android APK file and contains the application metadata, for example, permissions, name, version, etc. The manifest is present in binary XML format.
- classes.dex: It is the main executable in the Android APK bundle. If APK uses multiple DEX files, then it contains classes2.dex, Classes3.dex,..,ClassesN.dex.
- resources.arsc: These are precompiled resources that contain colors, strings, and styles.

2.3 Compilation of Android Application

Android applications are majorly developed in Java and Kotlin programming languages. Java and Kotlin compiled class files are get executed in JVM (Java Virtual Machine Environment) but android doesn't provide a JVM environment. Android uses DVM (Dalvik Virtual Machine) and ART (Android Runtime). These virtual environments use DEX byte code for the execution and hence a DEX compiler is introduced for compiling the class file to DEX file.

Steps involved in for compiling Java code to Class file is as:
Step 1: javac compiler compiles the Sample.java and generates Sample.class file (java byte-code).
Step 2: Java byte-code is executable in a JVM environment.
Step 3: JVM uses JIT (Just-In-time) compiler to convert the byte code into machine code.
Step 4: Resulted machine code is fed into memory and gets executed on the target machine.
Similarly, Kotlin code also generates a byte code compatible with to JVM environment. By default, it generates byte code compatible with Java 6 and configurable for the higher versions.

The complete mechanism of android application compilation and packaging in the form of the ".apk" file is depicted in figure 3. After the compilation of Java/Kotlin source code into the java byte code is fed into the DEX compiler. Dex compilers generate the "classes.dex" file which gets executed in a DVM and ART environment. The major components of the android application packaged file are:

- DEX executable (Classes.dex): This is the main executable that gets executed in the target environment.
- Encoded XML files: There are the layouts of the android application which get encoded with the help of the "aapt" tool. "aapt" is the tool in Android SDK used for this purpose.
- Encoded Manifest: Manifest file is an XML file that is the main configuration file used for executing the application.
- Signature Files: The package contains the signature of the Developer. Applications are signed with the "JarSigner" tool.

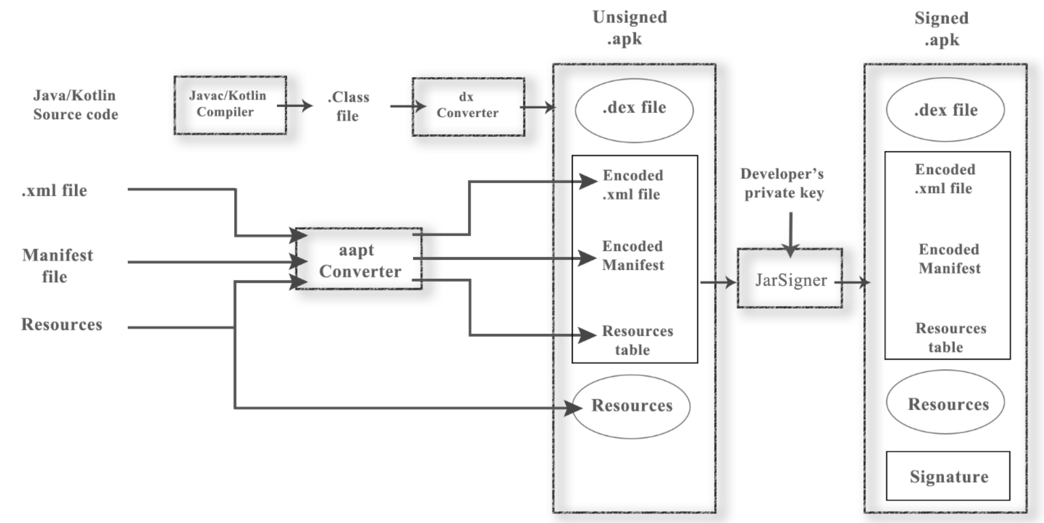


Figure 3: APK Compilation and Packaging

2.4 Android APK attack Vectors

The seriousness of the user towards security and privacy is very less. The users are more concerned about the ease of usage and getting the service free of cost. Most of the android applications generate their revenue based on the advertisement or the services provided by the original developer. Due to the enormous growth of technology a gap generates for understanding the risk related to that technology. Currently, the smartphone is the major source that contains the highly sensitive data of the user and has become the reason to bait cybercriminals. Cybercriminals always try to find an optimal way to target smartphones. According to Zhou and Jiang [35], the most common attack vectors for delivering malware are drive-by download, update, and repackaging.

- A drive-by download is the most common attack vector in a desktop environment. Internet users become a victim while surfing the internet. Users unknowingly download and execute malicious programs. A similar phenomenon is achieved on the Android platform by cybercriminals. In [21] a compromised website is embedded with a hidden iframe tag. When the compromised website is opened in an Android device which is identified based on the user-agent string, then the hidden iframe serves the malicious Android payload. This malicious payload was triggered due to some vulnerabilities or social engineering tricks.

- Update attack in android is served via the official app stores. The concerned benign application is published on Google play store or any other official market place. Application is installed by the user and at the first time of installation, no attack is launched. The developer of the android application pushes an update on the store with the malicious payload. Baskaran and Ralescu [5] states that this malicious application is get detected by analyzing the diff of both the published version.

- Repackaging is the most common attack vector used by cybercriminals in the Android ecosystem. This attack vector abuses the legitimate android application and the selection of applications is based on their popularity. For achieving the repackaged application with a malicious payload, legitimate applications are disassembled and malicious code is placed within a legitimate application. Now, this get compiled and signed for redistribution.

2.5 Android Malware

Any software intentionally designed to jeopardize the IT infrastructure is termed Malware. As in the rise of Android devices, malwares are designed to prey on these mobile devices. The success of android malware relies on the exploitable vulnerabilities present in the Android ecosystem. Android malware implantation in the victim environment is achieved based on three primary methodologies of social engineering: 1) Drive-by download, 2) update-attack, and 3) repackaging. The traditional attack methodology drive-by download is the technique that gets transferred from a general attack in the cyber domain to the mobile space domain. Although attackers do not exploit any vulnerabilities of mobile browsers directly, they typically entice users to download attractive or enrich featured android applications. In an update-attack scenario, rather than embedding the malicious portion of the application completely, mainly the updated and malicious component involved retrieving or downloading the malicious application as a patch of the already present application dynamically. The updated application contains the malicious code, not the initially installed original application. Hence it is more clandestine than the malware implantation methods that directly insert the complete malicious code initially. The third methodology, repackaging, is the most common methodology malware developers employ to piggyback malicious code into Android applications. In order to achieve this, malware developers take an Android application(bundled APK file), dismantle them via reverse engineering, malicious code implant inside them, rebuild it, and yield the new mutant of Android malicious application for getting published to an official or alternative Android application market. Victims of these applications get fascinated to download and install these repackaged applications.

These malwares are employed with various tactics for stealing the private and confidential information, using the paid service like SMS bombarding from compromised device, performing banking frauds and etc. Now a days, mobile devices are of higher computing configuration, this attracts the perpetrators for crypto currency mining. Figure 4 depicts various type of malwares that are present in android ecosystem. These types are described as:

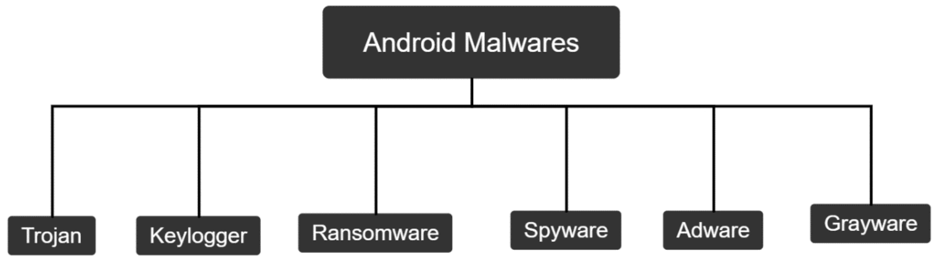


Figure 4: Types of Android Malware

(A) **Trojans**

Android malware disguised as a legitimate application and having the capabilities to spy on our activity, collect our sensitive data, gain access to our device, delete files, download secondary payload, and many more are classified as Trojan. Android applications are distributed via various stores; among them, the Google play store is the most trusted and secured one. These Trojan android malwares are distributed via these stores.

Stagefright is a media library in the Android framework that is widely exploited for sending text messages employed with malicious payload to any number. Trojans are also distributed via SMS or MMS. SMS Trojans racked up users' phone bills and wreak financial havoc by sending SMS messages to premium-rate numbers worldwide. The reason behind the popularity of this vulnerability is users get exploited even though if they didn't open or acknowledge the text message, the malware receives deployed, and attackers may get root access to the victim android device. The vulnerability gets patched very quickly but still proves the text message-based exploitation.

(B) **Keyloggers**

Android malwares which are capable of recording keystrokes or the information typed on mobile devices are categorized as Keyloggers. These are quickly obtainable to the common public and listed in the search engines is moderately surprising and annoying. Some developers are notoriously strengthening the surveillance of our colleagues and partners, while some are masquerading these apps as parental control solutions.

(C) **Ransomware**

Ransomware is popular among malwares designed for PCs. They encrypt the user information like photos, videos, documents and demand payment to be paid to the perpetrator of the attack. This ransom amount is usually paid via virtual currency like Bitcoin. Such malwares are also infecting the android device by encrypting files, locking the device screen also. A message gets promoted demanding payment in recompense for decrypting our Android device.

(D) **Spyware**

Spyware is the most generous malware found on mobile devices. The malware empowers attackers to obtain all the data on our phone, including calls, texts, contacts, and other delicate information, including a hijack of our camera and microphone. They also monitor our activity, records location, and lifts critical information, like usernames and passwords for e-commerce sites or email accounts. They are packaged with other apparently benign applications and unostentatiously collect information in the background.

(E) **Adware**

Annoying advertisement pop-ups and user interest data collection is done through a malicious android application called Adware. In an adware-infected device, vexatious advertisements are displayed frequently. These advertisements come upon the devices' entire screen even though users are not using internet browsers or add-enabled applications. Adware developers generate revenue based on the add click frequency. Some clicks force the user to download and install another malicious application, leading to complete compromise, including root access.

(F) **Grayware**

Grayware occupies the vague middle ground between benign and malicious behavior. These applications harvest data from Android devices with the objective of marketing. They intend not to harm users but annoy them. However, grayware may not execute activities that can be defined as entirely illegal or malicious, but its actions may still negatively impact the user regarding user efficiency, performance, and privacy.

2.6 Android Malware Analysis

Android malware analysis is the mechanism used by an analyst for flagging an android application as benign or malicious. Android malware analysis can be generalized into three main categories: Static Analysis, Dynamic Analysis, and Hybrid Analysis. Android suspicious applications are transformed in the form of Java bytecode, which can be obtained by disassembling an application during static analysis. Index of android application is maintained the manifest file that is also a significant aspect that needs to be considered for static analysis. Static analysis is obscured towards analyzing the portion of the code which gets downloaded and executed at runtime and is regarded as the more concerning disadvantage of static analysis that leads to failing the results during analysis. Dynamic analysis is advantageous in judging suspicious application code that is really performed by an application. An analyst uses typically Emulated environments for dynamic analysis. Hybrid Analysis is the unification of both dynamic and static methodology. For flagging benign or malicious with high accuracy, most of the analysts follow a hybrid approach.

3 RELATED WORK

[16] implemented the two stage fuzzy strategy to classify android malware family which have polymorphic variants. They used the regular expressions for identifying the callbacks to determine the behavior of Android malware. For classification 1-NN classifier and distance between the regular expression is considered. [2] classified the android malwares by an adaptive neuro-fuzzy inference system (ANFIS) with FCM. They generate the optimal number of clusters using FCM which is used to develop ANFIS classifier. The maximum achieved accuracy was 91% with considering their permission as feature vector.

[10] transformed the DEX file into images based on their sectional structure and extract the texture, color moment and string based features for predicting android malware family. Authors utilized the multiple kernel learning SVM algorithm for classification. [3] transformed the executables of various platforms to a grayscale image rather than extracting and analyzing the texture features. Various hash values Average Hash (AHash), Perception Hash (PHash), and Difference Hash (DHash) of the image were deliberated in the form of features. Neural networks are applied for classification malware. This method was good for dealing with high volume of malware but suffers with a poor accuracy due to loss of information in grayscale image. [11] transformed PE files into RGB images according to the structural architecture of PE files. Texture features via the GLCM algorithm and color features are extracted from the converted image. Among various classification algorithms Random forest (RF) exhibits the most effective classification accuracy of 97.4%. Although, the PE file structure and the APK file structure are quite different from each other and hence this method cannot be implemented directly for classifying the Android malware family.

[12] proposed the fuzzy consensus clustering based model for attributing the Advanced Persistent Threat (APT). They generate the multiple views based on the static and dynamic features which is further merged and used to train a decision tree model. The achieved accuracy for attributing APT group

is 95.2%. [7] utilized the fuzzy pattern tree classifier for the classification of IOT malware. They utilized the opcodes as the feature vector to classify the android malware.

[8] methodology Faldroid is based on constructing a frequent subgraph based on the dalvik opcode sequence which represents the common behavior of android malware. The implemented method is experimented with 8407 malwares with 36 families and 94.2% is the maximum achieved accuracy.

[20] presents an android malware classification based on fuzzy classification algorithms. Authors classified the 5000 android malware samples taken from Drebin dataset in the classes as: Botnet, Rootkit, SMS Trojan, Spyware, Installer and Ransomware. The considered fuzzy classification algorithms are: NN, OWANN, VQNN, FURIA, FuzzyRoughNN, FuzzyOwnershipNN, DiscernibilityClassifier, MultiObjectiveEvolutionaryClassifier.

4 METHODOLOGY

In the previous work, the authors did their research to find malicious behavior of android applications based on permissions, Dalvik opcodes, transformed images, DEX file's hashes, their Dalvik code flow graph and even used Fuzzy logic for classification.

The proposed methodology considers the multiple aspects of Android malware samples to generate feature vectors. These aspects are termed as views and are used to train FPT and FCM models individually. The developed fuzzy classifier model and clustering models are ensembles, and a novel feature vector is generated, which is further used to train the Decision Tree classifier to classify the android malware family. Figure 5 depicts the overall methodology in which firstly DEX file is extracted from the provided android APK, which is fed into the view generation module. The view generation module generates the six views based on multiple aspects: permission, transformed image, Dalvik opcode (frequency and tf-idf), and bytecode (frequency and tf-idf). Each view is used to train an FPT classifier and Clustered using the FCM model, and their results are combined and used as a feature vector to train the Decision Tree Classifier to classify the android malware family.

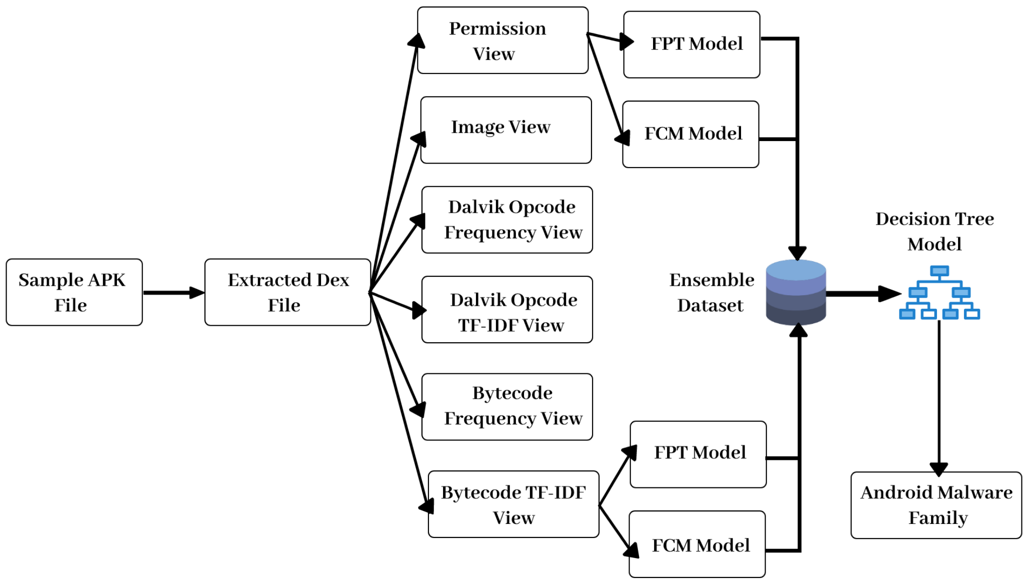


Figure 5: Overall Methodology for Classifying Android Malware Family

4.1 View Generation

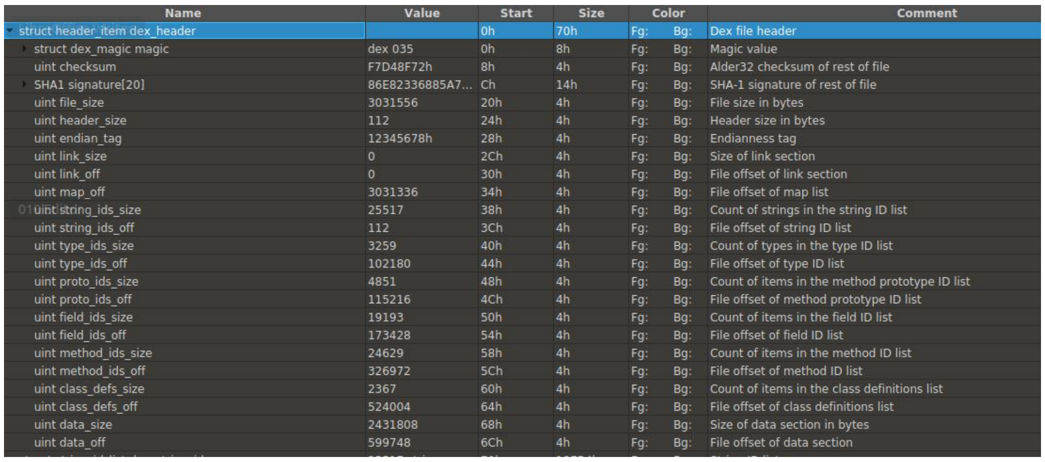
4.1.1 Permission View

Android permission-based framework is developed to restrict unwanted applications to access critical information like SMS, call logs, and other vital and sensitive data stored on the device. Android malware gets these permissions by tricking a user into incorporating attack vectors. A dictionary of permissions is created for generating permission view, which contains the unique permission set within all android malware datasets used in this research. Each android malware is transformed to a vector-based on its permission list present in its manifest file. Dimension of the permission view vector is equal to the length of the dictionary and generated using equation 1.

$$Perm_{apk} = \{x_i = 1 if D_{per}[i] \in SampleAPK_{per}\} \tag{1}$$

4.1.2 Image View

Android applications are compressed into APK files which are used for distribution. APK files are ZIP files such as JAR files which use Java libraries. APK file contains app code in the form of DEX file format, native libraries, resource files, configuration files, digital signatures, etc. Header, Index, and Data portion are the three portions of the DEX file. Figure 6 depicts snap of 010 Editor (hex editor) for representing the sections of DEX file. Basic information of the DEX file is present in the header section with the Index and offset values of other sections. The index portion of the DEX file consists size and offset of string index, proto Index, type index, method index, and field Index sections. The data portion consists size and offset of class definition sections and data sections. Therefore, in combining all these three portions, the DEX file is divided into eight sections.



Name	Value	Start	Size	Color	Comment
struct header_item dex_header		0h	70h	Fg: Bg:	Dex file header
struct dex_magic magic	dex 035	0h	8h	Fg: Bg:	Magic value
uint checksum	F7D48F72h	8h	4h	Fg: Bg:	Alder32 checksum of rest of file
SHA1 signature[20]	86E82336885A7...	Ch	14h	Fg: Bg:	SHA-1 signature of rest of file
uint file_size	3031556	20h	4h	Fg: Bg:	File size in bytes
uint header_size	112	24h	4h	Fg: Bg:	Header size in bytes
uint endian_tag	12345678h	28h	4h	Fg: Bg:	Endianness tag
uint link_size	0	2Ch	4h	Fg: Bg:	Size of link section
uint link_off	0	30h	4h	Fg: Bg:	File offset of link section
uint map_off	3031336	34h	4h	Fg: Bg:	File offset of map list
uint string_ids_size	25517	38h	4h	Fg: Bg:	Count of strings in the string ID list
uint string_ids_off	112	3Ch	4h	Fg: Bg:	File offset of string ID list
uint type_ids_size	3259	40h	4h	Fg: Bg:	Count of types in the type ID list
uint type_ids_off	102180	44h	4h	Fg: Bg:	File offset of type ID list
uint proto_ids_size	4851	48h	4h	Fg: Bg:	Count of items in the method prototype ID list
uint proto_ids_off	115216	4Ch	4h	Fg: Bg:	File offset of method prototype ID list
uint field_ids_size	19193	50h	4h	Fg: Bg:	Count of items in the field ID list
uint field_ids_off	173428	54h	4h	Fg: Bg:	File offset of field ID list
uint method_ids_size	24629	58h	4h	Fg: Bg:	Count of items in the method ID list
uint method_ids_off	326972	5Ch	4h	Fg: Bg:	File offset of method ID list
uint class_defs_size	2367	60h	4h	Fg: Bg:	Count of items in the class definitions list
uint class_defs_off	524004	64h	4h	Fg: Bg:	File offset of class definitions list
uint data_size	2431808	68h	4h	Fg: Bg:	Size of data section in bytes
uint data_off	599748	6Ch	4h	Fg: Bg:	File offset of data section

Figure 6: DEX file sections

For the transformation of the DEX file to RGB image size, entropy, bytecode, and proportions of sections are mapped to the size, red color channel, green color channel, and blue color channel of RGB image. Transformation and visualization are divided into five steps: Parsing of DEX file, Matrix Creation, Computation, Merging and Conversion.

DEX file Parsing: All eight sections of DEX file is parsed according to the DEX header as shown in Figure 6.

Matrix Creation: Each section of the DEX file is read byte and transformed into a byte matrix corresponding to each section. For deciding the width of matrix [10] proposed determining criteria according to DEX file size.

Computation: Calculation of the entropy matrix and the proportion matrix is done with the help of the bytecode matrix. Every section has a single value of entropy and proportion; hence these values remain the same for every section. *Entropy computation:* The entropy represents the stability of the byte sequence. Therefore, the entropy of each section is calculated by using the equation 2.

$$Entropy = \sum_{i=0}^{255} [p(c_i) \log_2 p(c_i)] \tag{2}$$

Where c_i = frequency of byte i, $p(c_i)$ =probability of frequency of byte i, The value of $p(c_i) \log_2 p(c_i)$ is defined as 0 if the number of bytes is 0. Since range of entropy lie between [0, 8] and pixel value lie between [0, 255] hence entropy value was extended non-linearly. Value of R channel is calculated as shown in equation 3.

$$R = (Entropy^2 \mod 8) \times 255/8 \tag{3}$$

Proportion Computation: Every section of the DEX file has a different size because every section has many methods, variables, and classes. Therefore the proportion of each section may vary. The Blue channel of RGB images is mapped with the proportion of each section. Equation 4 depicts the calculation formula for the proportion.

$$Proportion = (SectionSize)/(FileSize) \tag{4}$$

The range of proportion is [0, 1]; hence it gets mapped with a range of pixel [0, 255] according to the equation 5.

$$B = Proportion \times 255 \tag{5}$$

Merging and Conversion: After computation of R channel as entropy, G channel as byte code matrix, and B channel as proportion matrix is merged and RGB tuple matrix is obtained. This RGB tuple matrix is finally converted into the RGB image. Figure 7 illustrates the overall steps of transformation, and the GIST algorithm for feature extraction is applied as:

Step 1: Gabor filter bank shown in expression 6 is utilized to filter the grayscale Image. A Gaussian envelope modulates a sinusoidal plane with fixed direction and frequency in Gabor function with 2-D mode. Gabor filters are selective in terms of direction and frequency. Multiple groups of Gabor filters can be constructed by modifying the direction and frequency parameter.

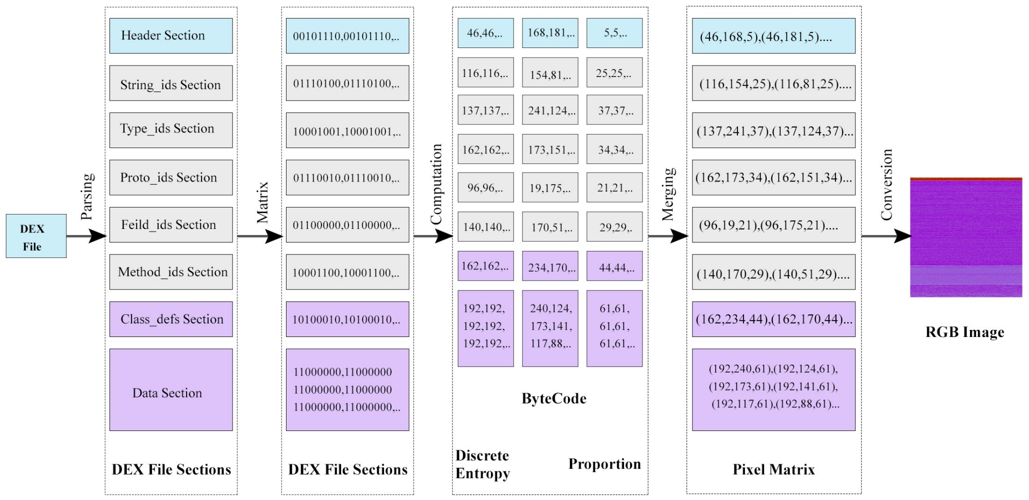


Figure 7: DEX to RGB transformation

$$\begin{gathered}
g_{pq}(x,y) = a^{(-p)} g(x^{'},y^{'})(a>1) \\
x^{'} = a^{(-p)}(x\cos\theta + y\sin\theta) \\
y^{'} = a^{(-p)}(-x\cos\theta + y\sin\theta) \\
\theta = q\pi/(q+1)
\end{gathered} \tag{6}$$

where θ = direction of filter $a^{(-p)}$ = scaling factor of wavelet expansion, $a = (U_h/U_l)^{(1/(p-1))}$ U_h and U_l are the lower and upper value of interest of frequencies. Gabor filter bank shown in expression 6 is used for generating the f number of Gabor filters by modifying the value of p and q. So the number of the filter is $f = p * q$. During this research, p=4 and q=6 are considered, and a total of 32 Gabor kernels are generated for filtering the obtained grayscale image.

Step 2: A Gabor filter bank with f filter channels is used to convolute the gray image f(x, y) to obtain the f number of the filtered image. A grid of size $n_b \times n_b$ is obtained from the filtered image. The mean value of grids is considered as a feature vector. Hence a vector of length $f \times n_b \times n_b$ is obtained as a feature vector. During this research, the considered size of the grid is 4 × 4, and a total of the 512-dimensional feature vector as image view is generated.

4.1.3 Dalvik Opcode frequency View

Android APK file consists of a DEX file executable in Android Runtime Environment (ART). Dalvik opcodes are the intermediate opcodes that get executed in ART. These opcodes are responsible for achieving the objective of the Android Application. To consider it as a feature vector, firstly, a dictionary of Dalvik opcode is created. The dictionary is the unique Dalvik opcodes that are present in the complete dataset. The resulted dictionary is used as feature columns in the Dalvik opcode view.

Frequency view is the vector representation of the count of each opcode that is present in the dictionary in the respective Android Malware Sample. The vectorization of this view is done according to equation 7.

$$Opcodefrequency_{apk} = \{x_i = (Opcodecount_{apk}[i]/|SampleAPK_{op}|)\} \tag{7}$$

4.1.4 Dalvik Opcode TF-IDF View

TF-IDF value of a Dalvik opcode represents its relevance among all the datasets. Every DEX file is vectorized according to equation 8 to generate this view. Again the length of the resulted feature vector is equal to the length of the Dalvik opcode dictionary.

$$\begin{gathered} OpcodeTFIDF = tf \times idf \\ tf = Opcodefrequency_{apk} \\ idf = \log\left[|TotalSampleAPK|/|SampleAPK_{op} \in SampleAPK|\right] \end{gathered} \tag{8}$$

4.1.5 Bytecode frequency and TF-IDF View

Since all the DEX is the sequence of bytecode hence the frequency and tf-idf views are generated similar to Dalvik opcode views. For this view, all possible Bytecode values are considered a dictionary of bytecode (Dict_ByteCode = 0,1,2 ......, 255). Hence the length of each vector is 256 in Bytecode view.

4.2 Fuzzification

Crisp Set: Crisp represent the collection of items having identical properties like finiteness and countability. A crisp set is based on Boolean logic which means the item is the member of set or not. A crisp set 'S' is defined over the universal set 'U' as per the given equation .

$$S = \{x : x \in \text{U and x have the identical properties P}\} \tag{9}$$

Union, intersection, compliment and difference are the most common operation performed of crisp sets. The properties which remains hold on the crisp set are associativity, distributivity, commutativity, idempotency, identity, involution and transitivity. Crisp logic is the traditional way of knowledge representation which does not deal with the methodology of interpreting non-categorical and imprecise data.

Fuzzy Set: Fuzzy logic represent the partial logic of truth or vagueness of reasoning. It takes the knowledge or data in a very similar way that our brain takes in. The transformation crisp values for the fuzzy inference engine (fuzzy set) is called fuzzification. Fuzzification is the assignment of membership function, which can represent the linguistic notion of a crisp set.

All the generated views represent the crisp value which must be fuzzified before implementing any fuzzy logic algorithm for classification and clustering. [1] proposed triangular and trapezoidal membership function $\mu(x)$ for fuzzifying the dataset. Some other membership functions are Gaussian, Generalized bell, and Sigmoid membership function.

In this research triangular fuzzifier is used to generate the fuzzy membership value $\mu(x)$.The definition of $\mu(x)$ is shown in equation 10 and depicted in Figure 8. In equation 10, a is the lower limit, b is the upper limit and c is the actual value. The considered value of a is $min(F)–[max(F)–min(F)]^2$, b is $min(F)$ and c is $max(F)$ where F refers the corresponding Fuzzy Set.

$$\mu(x) = \begin{cases} 0 & \text{if } x \leq a \\ [(x-a)/(b-a)] & \text{if } a \leq x \leq b \\ [(c-x)/(c-b)] & \text{if } b \leq x \leq c \\ 0 & \text{if } x \geq c \end{cases} \tag{10}$$

4.3 Fuzzy Pattern Tree Classifier

FPT [26] is a fuzzy-based classification algorithm introduced recently. This research considers the bottom-up approach for learning. It is a hierarchal tree-like structure whose inner nodes are associated with fuzzy-based logical and arithmetic operators. The most commonly used fuzzy operators in

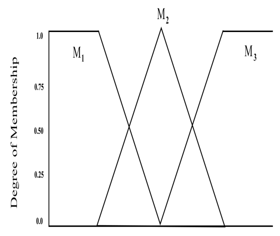


Figure 8: Degree of Membership

constructions of FPT are t-norms, t-conorms, weighted average (WA), and ordered weighted average (OWA) [14, 25, 32]. Input to the tree is provided at the leaf node. An instance of an FPT model is depicted in Figure 9. An FPT model is the collection of pattern trees, and each pattern tree is associated with a specific class. For classification, test data is given to the FPT model, and the highest score resulting pattern tree represents the predicted class.

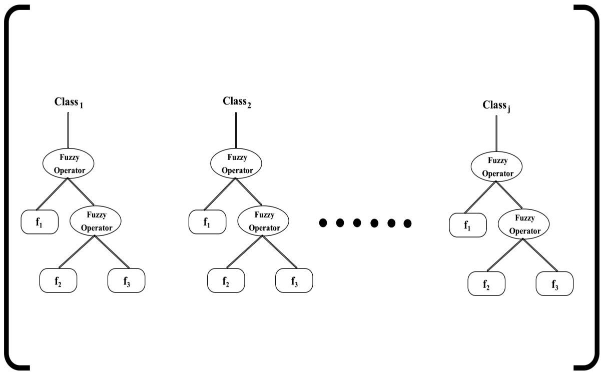


Figure 9: An instance of FPT model

Dataset $X(x_1, x_2, x_3 \ldots, x_i)$ corresponding to each view is the input for algorithm 1. Initially, three sets of basic partition tree PT, C^0 candidate tree, and M^* FPT model are initialized. PT is the collection of primitive fuzzy pattern trees F_{ij} where i represents the feature vector, and j refers to the corresponding class. A candidate pattern tree C^0 is initialized based on the root mean square error(RMSE) as a loss function. C^0 is the collection of N best trees among the PT, i.e., C^0 is a subset of PT. M^* contains the

N best trees of C^0 as the initial FPT model, which get tuned and improved as the final FPT model. Now iteratively, each pattern tree from M* is taken and optimized with fuzzy operator and trees from PT as mentioned in algorithm 1. This research considers the maximum iteration as five, and in every iteration loss function, RMSE is used to optimize M^*. Finally, the M^* model is used to predict the confidence value corresponding to each class.

Algorithm 1 Fuzzy Pattern Tree Classifier

```
 1: $PT = \{F_{i,j} | \forall i \in inputfeatures, and \forall j \in classes\}$
 2: $S = PT$
 3: $C^0 = argmin^{NB}_{F \in PT} \left[\sum_{(x,y) \in T} RMSE(y, F(x))\right]$
 4: $M^* = argmin\,[error(C^0)]$
 5: $t_{max} = 5$
 6: $t = 0$
 7: $MaximumDepth = 10$
 8: while $t \leq t_{max}$ do
 9:     $t = t + 1$
10:     $C^t = C^{t+1}$
11:     for all $L \in leafs(M^*)$ do
12:         if $Depth(L) \leq MaximumDepth$ then
13:             for all $\psi \in FuzzyOperators$ do
14:                 for all $p \in PT$ do
15:                     $C^t = C^{t-1} \cup replaceleaf(M^*, L, \psi, p)$
16:                 end for
17:             end for
18:         end if
19:     end for
20:     $M^* = argmin^{n}_{F \in C^t} \left[\sum_{(x,y) \in T} RMSE(y, F(x))\right]$
21: end while
22: return $M^*$
```

4.4 Fuzzy C-means Clustering

There are mainly two types of clustering: Hard Clustering (Figure 10) and Soft Clustering(Figure 11). In hard clustering, each data point belongs to a particular fixed cluster number. In soft clustering, instead of defining a specific cluster number, a probability is defined for each data point regarding each cluster number.

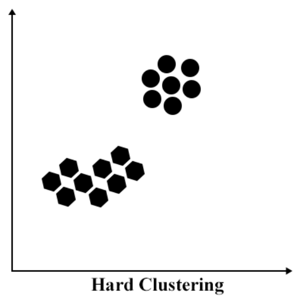


Figure 10: Hard Clustering

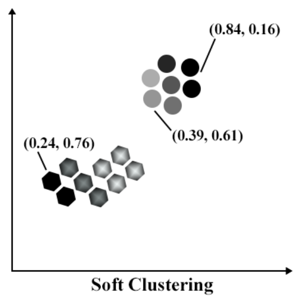


Figure 11: Soft Clustering

To generate fuzzy clusters, this research implements the Fuzzy C-means Clustering [22]. The technique is mentioned in algorithm 2. In FCM, C refers to the maximum cluster number, which is taken 200, and fuzziness parameter m is taken 1.75. The training set of each view is clustered, and a trained partition matrix U is returned corresponding to each view. Since each row refers to the probability distribution of data points, the sum of each row of the partition matrix should always be one.

4.5 Ensemble FPT and FCM vectors with Decision Tree

The trained models of FPT and FCM are used to generate a new dataset on each view. The dataset of each view is fitted on FPT and FCM models for

Algorithm 2 Fuzzy C Means Clustering

```
1: $C = 200$ /* No of Clusters */
2: $m = 1.75$ /*Fuzziness Parameter */
3: $U^* = [u_{ij}]matrix, U^0$ /*Initialize Partition Matrix */
4: $k = 0$
5: **repeat**
6:     $C^{(k)} = \left[\sum_{i=1}^{N} u_{ij}^m x_i / \sum_{i=1}^{N} u_{ij}^m\right]$
7:     $U^{(k+1)} = \left[\sum_{k=1}^{c} (x_i - c_j / x_i - c_k)^{2/m-1}\right]^{-1}$
8: **until** $|U^{(k+1)}| - |U^{(k)}| \leq StoppingCriteria$
9: **return** $U$
```

producing the new dataset as mentioned in algorithm 3. Now the aggregated dataset X is combined with the corresponding class Y. Here Y represents the actual class of the android malware family.

This module of the proposed methodology is the final decision maker on the aggregated dataset of FPT and FCM models. $[X, Y]$ the new aggregated dataset is used to train a decision tree model that decides the Android malware family class. The optimal depth of the decision tree is considered ten during the research, which is obtained by the GridCVSearch method.

Algorithm 3 Ensemble FPT and FCM Models

```
1: $X_{FPT} = FPT_{view}(X_{view})$
2: $X_{FCM} = FCM_{view}(X_{view})$
3: $X = [X_{FPT}, X_{FCM}]$
4: **return** $X$
```

5 RESULTS AND DISCUSSION

5.1 Dataset

The samples utilized in this research are based on the RmvDroid Android malware dataset. [31] collects the android applications of Google play in 4 years and then uses Virus Total to label the application. They also observe the removal of the application from the play store to label the application.

This dataset contains 9,133 android malware samples that belong to 56 families. These samples are randomly chosen from the dataset and used to generate the respective views. After analyzing the dataset, it may confer that it contains the imbalance number of malware samples in their families. Hence for getting the effective results, 11 families and 150 samples per family are taken as shown in Table 1.

S No.	Family name	Sample Count
1.	AIRPUSH	2872
2.	MECOR	993
3.	PLANKTON	8022
4.	ADWO	690
5.	YOUMI	597
6.	GAPPUSIN	441
7.	MOBIDASH	344
8.	VISER	291
9.	DOWGIN	279
10.	LEADBOLT	179
11.	KUGUO	168

Table 1: Android Malware Family Dataset

5.2 Performance Evaluation

The discussed methodology is used to classify the Android malware families. The dataset of RmvDroid is fed into the framework, and the following six different feature vector is generated based the multiple views. These views are based on permission, image, Dalvik opcode, and DEX bytecode. Every single view is firstly used to train, and Fuzzy based FPT classifier. There are six different FPT classifiers based on each view. Similarly, these views are clustered based on the soft clustering technique Fuzzy C-means. The clustering algorithm considers the number of clusters c=200 and fuzziness parameter m=1.75. A partition matrix based on each view is generated as a result. These FPT and FCM models are aggregated and fed into a Decision Tree with its actual class, which yields a trained model with an accuracy of 95.75%.

The performance of machine learning algorithms is evaluated based on four core metrics True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FN). Based on these metrics, precision, recall, and F1 Score is calculated to assess the proposed methodology. The confusion matrix visualizes the performance of any machine learning algorithm. The predicted class is represented by the row of the confusion matrix and the actual class by column. The accuracy of the trained model for each Android malware family is depicted through diagonal cells.

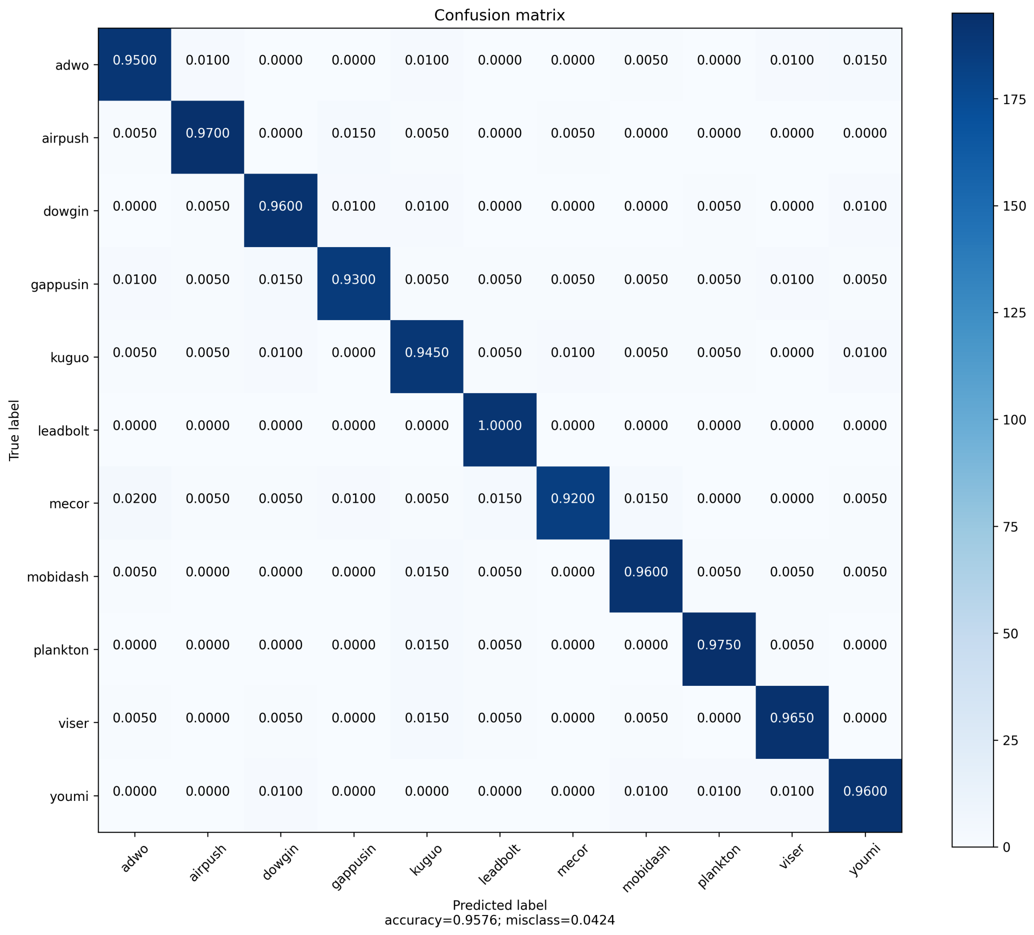
View	**F1 Score**	**Accuracy**	**Precision**	**Recall**
Permission View	0.748592	0.690205	0.690205	0.921700
Image View	0.443169	0.492027	0.492027	0.445675
Count_Opcode View	0.602364	0.665148	0.665148	0.628156
TF-IDF_Opcode View	0.708458	0.724374	0.724374	0.737401
Count_Bytecode View	0.429397	0.448747	0.448747	0.508666
TF-IDF_Bytecode View	0.517084	0.517084	0.4779082	0.51708
Combined View	0.957495	0.957554	0.957764	0.957554

Table 2: Performance Metrics

The performance metrics of every single view are represented in Table 2 and compared with the performance metrics of the combined view. The results obtained based on a single view need a significant improvement to classify the android malware family. In order to achieve this improvement, when these views are aggregated, then the performance metrics improved significantly. Combined view F1 Score is achieved up to 95.74%. Figure 12 represents the confusion matrix of the integrated view. Finally, it can be justified that classifying the android malware family based on multiple views and combining fuzzy logic algorithms for decision trees yield classification accuracy up to 95.74%. The consideration of multiple aspects ensures that any single view cannot perform, balanced by the other views.

5.3 Discussion

This section discusses the methodology adopted in this paper with similar work done for the Android malware family classification. [10] methodology is based on computer vision domain. The authors transformed the DEX file into images and applied an SVM multi-kernel for the classification of Android



	adwo	airpush	dowgin	gappusin	kuguo	leadbolt	mecor	mobidash	plankton	viser	youmi
adwo	0.9500	0.0100	0.0000	0.0000	0.0100	0.0000	0.0000	0.0050	0.0000	0.0100	0.0150
airpush	0.0050	0.9700	0.0000	0.0150	0.0050	0.0000	0.0050	0.0000	0.0000	0.0000	0.0000
dowgin	0.0000	0.0050	0.9600	0.0100	0.0100	0.0000	0.0000	0.0000	0.0050	0.0000	0.0100
gappusin	0.0100	0.0050	0.0150	0.9300	0.0050	0.0050	0.0050	0.0050	0.0050	0.0100	0.0050
kuguo	0.0050	0.0050	0.0100	0.0000	0.9450	0.0050	0.0100	0.0050	0.0050	0.0000	0.0100
leadbolt	0.0000	0.0000	0.0000	0.0000	0.0000	1.0000	0.0000	0.0000	0.0000	0.0000	0.0000
mecor	0.0200	0.0050	0.0050	0.0100	0.0050	0.0150	0.9200	0.0150	0.0000	0.0000	0.0050
mobidash	0.0050	0.0000	0.0000	0.0000	0.0150	0.0050	0.0000	0.9600	0.0050	0.0050	0.0050
plankton	0.0000	0.0000	0.0000	0.0000	0.0150	0.0050	0.0000	0.0000	0.9750	0.0050	0.0000
viser	0.0050	0.0000	0.0050	0.0000	0.0150	0.0050	0.0000	0.0050	0.0000	0.9650	0.0000
youmi	0.0000	0.0000	0.0100	0.0000	0.0000	0.0000	0.0000	0.0100	0.0100	0.0100	0.9600

Figure 12: Confusion Matrix on Combined View

malware classes. [12] mechanism is based on multiple views for attributing the Advanced persistent threat payloads and achieved accuracy is 95.2% on five APT groups. [7] utilizes FPT for classifying IoT malware as 0/1 classification. [20] done android malware classification based on fuzzy classification algorithms into the following classes: Botnet, Rootkit, SMS Trojan, Spyware, Installer, and Ransomware. The proposed methodology uses the concept of ensemble learning and implements it on Android Malware Family classification. The methodology in this research is based on multiple views of Android applications, which includes permission, image, Dalvik Opcode, and Bytecode. These views adopt ensemble learning of fuzzy-based classification and clustering algorithms to generate the input for the Decision Tree.

6 CONCLUSION AND FUTURE WORK

Android malware family classification is one of the most demanding tasks in the android malware threat domain. This research is done based on multiple aspects of an Android application which is termed as view. The views considered for attaining the classification problems are permission, image representation of DEX file, underlying Dalvik opcodes, and bytecode of the DEX file. These multiple views are vectorized and fuzzified using triangular fuzzification. The fuzzified vectors are used to train the FPT classifier and soft-clustered using the FCM algorithm. Finally, the FPT and FCM vectors are combined, and a decision tree model is used to final classification of android malware families. From the future point, the views considered during this research are based on static analysis, but also these views may be based on dynamic analysis.

Appendices

A Raw Feature Extraction

```
from androguard.misc import AnalyzeAPK
from collections import defaultdict
from operator import itemgetter
import hashlib
import os
import json
basepath = '/home/rax/Desktop/rmvdroid'
outputpath = '/home/rax/Desktop/rmvdroid_output'
result = {}
for root, dirs, files in os.walk(basepath):
    for fil in files:
        filepath = os.path.join(root, fil)
        dirpath = outputpath
        if not os.path.exists(dirpath):
            os.makedirs(dirpath)
        filepathlist = filepath.split('/')
        filename = filepathlist[-1]
        familyname = filepathlist[-2]
        dirpath = os.path.join(dirpath, familyname)
        if not os.path.exists(dirpath):
            os.makedirs(dirpath)
        print(filepath, dirpath)
        try:
            a,d,dx = AnalyzeAPK(filepath)
            permissions = a.get_permissions()
            activities = a.get_activities()
            services = a.get_services()
            recievres = a.get_receivers()
            package = a.get_package()
            apkname = a.get_app_name()
            andr_ver_code = a.get_androidversion_code()
            and_ver_name = a.get_androidversion_name()
```

```
min_sdk = a.get_min_sdk_version()
max_sdk = a.get_max_sdk_version()
target_sdk = a.get_target_sdk_version()
eff_target_sdk = a.get_effective_target_sdk_version()
result.update(filepath=filepath,filename=filename,
familyname=familyname,apkname=apkname,
package=package,andr_ver_code=andr_ver_code,
min_sdk=min_sdk,max_sdk=max_sdk,
eff_target_sdk=eff_target_sdk,
permission=permissions,
activities=activities,services=services)
dexcount = 0
dexnames = []
dexpathnames = []
dexpath = dirpath
if not os.path.exists(dexpath):
    os.makedirs(dexpath)
for x in a.get_dex_names():
    dexnames.append(x)
    dexfilename = filename + '_'+ x
    dexpath = os.path.join(dexpath,dexfilename)
    dexpathnames.append(dexpath)
    '''f = open(dexpath,'wb')
    f.write(a.get_file(x))
    f.close()'''
dexcount = len(dexnames)
if dexcount>1:
    multidex = 1
else:
    multidex = 0
opcode_value = []
opcode_mnemonics = []
for method in dx.get_methods():
    if method.is_external():
        continue
    m = method.get_method()
    if m.get_code():
        byte_code = m.get_code()
```

```
                    if byte_code != None:
                        byte_code = byte_code.get_bc()
                        idx = 0
                        for i in byte_code.get_instructions():
                            opcode_value.append(i.get_op_value())
                            opcode_mnemonics.append(i.get_name())
                            #print (i.get_op_value(), i.get_name())
                            idx += i.get_length()
            result.update(opcode_value=opcode_value,
            opcode_mnemonics=opcode_mnemonics,
            multidex=multidex,
            dexpaths=dexpathnames)
            outputfile = os.path.join(dirpath,filename
                                     +'_'+'result.json')
            with open(outputfile, "w") as outfile:
                json.dump(result, outfile)
        except:
            pass
```

B Permission and Opcode View Generation

```
from collections import defaultdict
from operator import itemgetter
import hashlib
import os
import json
import glob
from collections import Counter
import csv
import time
basepath = '/home/rax/Desktop/rmvdroid_output'
andr_mal_class = ["airpush","mecor","plankton","adwo",
"youmi","gappusin","kuguo","mobidash","viser","dowgin",
"leadbolt","smsreg","admogo","domob","secapk","kyview"]

def clean_dict(filepath,typ):
    opcode_dict = open(filepath,"r").read()
    opcode_dict = opcode_dict.split(',')
    opcode_dict[0] = opcode_dict[0].replace('{', '')
    opcode_dict[-1] = opcode_dict[-1].replace('}', '')
    opcode_dict_clean = []
    for op in opcode_dict:
        opcode_dict_clean.append(op.strip('\' '))
    if typ=='op':
        op_dict = {}
        for i in opcode_dict_clean:
            #print(i)
            op_dict.update({i:0})
        return op_dict
    elif typ=='per':
        per_clean = {}
        for p in opcode_dict_clean:
            p = p.split('.')[-1]
            per_clean.update({p:0})
        return per_clean

opmen_dict = clean_dict('opcode_menomonics_dict.txt','op')
per_dict = clean_dict('permission_dict.txt','per')

def createdata(filepath):
    with open(filepath) as f:
        data = json.load(f)
        perm = data['permission']
        perm1 = []
        for p in perm:
            p = p.split('.')[-1]
            perm1.append(p)
        opcode = data['opcode_value']
        opcode_mnemonics = data['opcode_mnemonics']
        opcount = dict(Counter(opcode_mnemonics))
        percount = dict(Counter(perm1))
        f_opmen_dict = opmen_dict
        f_per_dict = per_dict
        f_opmen_dict.update(opcount)
```

```
            f_per_dict.update(percount)
        return f_opmen_dict,f_per_dict

    def createcsv(oplist,perlist):
        csv_column1 = opmen_dict.keys()
        csv_column2 = per_dict.keys()
        csvfile1  = "opcode.csv"
        csvfile2 = "per.csv"
        try:
            with open(csvfile1, 'w') as csvfile1:
                writer1 = csv.DictWriter(csvfile1, fieldnames=csv_column1)
                writer1.writeheader()
                for data in oplist:
                    writer1.writerow(data)
            with open(csvfile2, 'w') as csvfile2:
                writer2 = csv.DictWriter(csvfile2, fieldnames=csv_column2)
                writer2.writeheader()
                for data1 in perlist:
                    writer2.writerow(data1)
        except IOError:
            print("I/O error")

    oplist = []
    perlist = []
    for cl in andr_mal_class:
        dirpath = os.path.join(basepath,cl)
        #print(dirpath)
        filelist = os.listdir(dirpath)
        for fi in filelist:
            filepath = os.path.join(dirpath,fi)
            if '.json' in filepath:
                op_data = 0
                per_data = 0
                op_data, per_data = createdata(filepath)
                op_data.update(clas=cl)
                per_data.update(clas=cl)
                oplist.append(op_data.copy())
                perlist.append(per_data.copy())
```

```
createcsv(oplist, perlist)
```

C Image View Generation

```
import numpy as np
from cv2 import cv2
import os
from decimal import *
def cal_color_moment(img):
    img_h, img_w = img.shape
    img_mean = np.sum(img)/(img_h*img_w)
    img_deviation = (img-img_mean)**2
    img_variance_t = (np.sum(img_deviation))/(img_h*img_w)
    img_variance  = np.sqrt(img_variance_t)
    img_deviation = (img-img_mean)**3
    img_skewness_t = (np.sum(img_deviation))/(img_h*img_w)
    img_skewness = np.cbrt(img_skewness_t)
    img_mean = format(img_mean, '.5f')
    img_variance = format(img_variance, '.5f')
    img_skewness = format(img_skewness, '.5f')
    return img_mean, img_variance, img_skewness

def build_filters():
    filters = []
    ksize = 21
    for theta in np.arange(0, np.pi, np.pi / 8):
        params = {'ksize':(ksize, ksize),
        'sigma':8.0, 'theta':theta, 'lambd':10.0,
        'gamma':0.5, 'psi':0, 'ktype':cv2.CV_32F}
        kern = cv2.getGaborKernel(**params)
        for i in range(1,5):
            kern *= i
        #filters.append((kern,params))
            filters.append(kern)
    return filters
tp = ["droidkungfu","fakeinsta","dowgin",
"youmi", "bankbot", "airpush", "minimob",
"lotoor","kuguo","boqx","jisut"]
for tt in tp:
    target_path = "../pyDexparser/"+tt+"_gabor_sig/"
    sample_dir_path = "../pyDexparser/"+tt+"_images/"
    if not os.path.exists(target_path):
            os.makedirs(target_path)
            print(target_path)
    for root, dir, files in os.walk(sample_dir_path):
        for filename in files:
            h_index = filename.find("_")
            if (h_index) != -1:
                file_hash = filename[0:h_index]
            sample_files_path = os.path.join(sample_dir_path, filename)

            if "gray" in sample_files_path:
                #print(filename, file_hash)
                img = cv2.imread(sample_files_path)
                img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
                #print img.shape
                #print len(img)
                filters = build_filters()
                #print(len(filters))
                #meand of the grid image(4*4)
                texture_feature = []
                for filter in filters:
                    filtered_img =
                    cv2.filter2D(img, cv2.CV_8UC3, filter)
                    size_x = filtered_img.shape[1]
                    size_y = filtered_img.shape[0]
                    for i in range(0, 4):
                      for j in range(0,4):
                        roi = filtered_img[int(i*size_y/4):
                        int(i*size_y/4 + size_y/4),
                        int(j*size_x/4):int(j*size_x/4+ size_x/4)]
                        texture_feature.append(
                        format(np.mean(roi),'.5f'))
                target_file_name = target_path + "texture/"
                if not os.path.exists(target_file_name):
                    os.makedirs(target_file_name)
                target_file_name = os.path.join
                (target_file_name, file_hash)
                with open(target_file_name, "w") as outfile:
                    outfile.write(","
                    .join(str(v) for v in texture_feature))
                print(texture_feature)
```

D FPT, FCM and Decision tree Models

```
import numpy as np
import pandas as pd
import sys
import matplotlib.pyplot as plt
import argparse
import pickle
from fuzzycmaster.fcmeans.fcm import FCM
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from fylearn.fpt import FuzzyPatternTreeTopDownClassifier
import matplotlib.pyplot as plt
from fylearn.fpt import FuzzyPatternTreeClassifier
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import classification_report
from sklearn.metrics import silhouette_score
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import f1_score,accuracy_score
from sklearn.metrics import precision_score,recall_score
from sklearn.metrics import plot_confusion_matrix
import seaborn as sns
from sklearn.metrics import confusion_matrix
###Reads the file and returns the label and data
def read(file):
    df = pd.read_csv(file)
    #df1 = df.head(10)
    x = df.iloc[:,:-1]
    x = x.apply(lambda x:x.fillna(x.mean()),axis = 0)
    y = df.iloc[:,-1]
    k = df.columns[-1]
    z = list(df[k].unique())
    return x,y,z

## Creates fpt Model and returns classification report
def create_fpt_model(x,y,file,unique_vals):
    le = LabelEncoder()
    print(y)
    z = y.unique()
    le.fit(y)
    y = le.transform(y)
    x_train,x_test,y_train,y_test=
    train_test_split(x,y,test_size =0.2)
    mod = FuzzyPatternTreeClassifier(max_depth=10)
    mod.fit(x_train,y_train)
    #z = file.split('.')
    k = file.split('/')
    z=k[-1].split('.')[0]

    with open(z+'fpt','wb') as f:
        pickle.dump(mod,f)
    _,y_pred = mod.predict(x_test)
    fpt_r,_ = mod.predict(x_train)

    with open(z+'preds_fpt','wb') as f:
        pickle.dump(fpt_r,f)
    cm = confusion_matrix(y_test,y_pred)
    print(cm)
    sns_plt = sns.heatmap(cm/np.sum(cm),
    xticklabels =unique_vals,
    yticklabels=unique_vals,
    annot = True,fmt = '.2%',
    cmap = 'Blues')
    plt.savefig('fpt'+z+'confusion_matrix.png')
    #graph = plot_confusion_matrix(mod,x_test,y_test)
    #plt.savefig('output.png'+z)
    print(x_test)
    print(y_test)

    print(f1_score(y_test,y_pred,average = 'micro'))
    print(accuracy_score(y_test,y_pred))
    print(recall_score(y_test,y_pred,average = 'micro'))
    return f1_score(y_test,y_pred,average = 'weighted'),
    accuracy_score(y_test,y_pred),
    recall_score(y_test,y_pred,average = 'weighted'),
    precision_score(y_test,y_pred,average = 'weighted')

##creates fcm model and returns sillohte coefficient
def create_fcm_model(x,y,file):
    x = np.array(x)
    x_train,x_test,y_train,y_test =
    train_test_split(x,y,test_size=0.2)
    fcm = FCM(n_clusters=200)
    fcm.fit(x_train)
    print(x_train.shape)
    #print(file[0])
    k = file.split('/')
    z=k[-1].split('.')[0]
    with open(z+'fcm','wb') as f:
        pickle.dump(fcm,f)
    with open(z+'fcm','rb') as f:
        fcm = pickle.load(f)
    preds=fcm.predict(x_train)
    with open(z+'preds_fcm','wb') as f:
        pickle.dump(preds,f)
### generates csv and creates a dataframe
def create_dataframe(y,file):
    k = file.split('/')
    p=k[-1].split('.')[0]
    with open(p+'preds_fpt','rb') as f:
        fpt = pickle.load(f)
    with open(p+'preds_fcm','rb') as f:
        fcm = pickle.load(f)
    res = []
    for i in range(len(fpt)):
        x1 = fpt[i]
        y1 = fcm[i]
        z = np.concatenate([x1,y1])
        res.append(z)
    df = pd.DataFrame(res)
    df['output'] = y
    df.to_csv(p+'data',header = False,index = False)
    return df


####creates Decision Tree and generates classification report
def create_decision(df):
    clf = DecisionTreeClassifier()
    x = df.iloc[:,:-1]
    y = df.iloc[:,-1]
    print(x)
    x_train,x_test,y_train,y_test =
    train_test_split(x,y,test_size=0.2)
    clf.fit(x_train,y_train)
    with open('dt_new','wb') as f:
        pickle.dump(clf,f)
    y_pred = clf.predict(x_test)
    print(x_test)
    print(y_test)
    print(y_pred)
    cm = confusion_matrix(y_test,y_pred)
    sns_plt = sns.heatmap
    (cm/np.sum(cm),
    annot = True,fmt = '.2%',
    cmap = 'Blues')
    plt.savefig('final'+'confusion_matrix.png')
    return accuracy_score(y_test,y_pred),
    f1_score(y_test,y_pred,average='weighted'),
    recall_score(y_test,y_pred,average = 'weighted'),
    precision_score(y_test,y_pred,average='weighted')
```

```
def create(file):
    x,y,z = read(file)
    f1,acc,recall,precision =create_fpt_model(x,y,file,z)
    create_fcm_model(x,y,file)
    df = create_dataframe(y,file)
    return df,f1,acc,recall,precision,z

def save_fig(df,name=''):
    fig = plt.figure(figsize=(15,8))
    ax = fig.add_axes([0,0,1,1])
    df.plot.bar(rot=0, ax=ax)
    plt.savefig("comparison of performance
    metric"+name,
    bbox_inches = "tight",
    facecolor = "white")

if __name__=='__main__':
    df = pd.DataFrame()
    df_old = pd.DataFrame()
    n = len(sys.argv)
    print(n)
    flag = 0
    lis = []
    k = []
    k.append('final')
    f1 = []
    acc = []
    precision = []
    recall = []
    res_df = pd.DataFrame()
    df_final = pd.DataFrame()
    for i in range(1,n):
        file = sys.argv[i]
        print(file)
        k = file.rfind('/')
        if (k == -1):
            z = file.rfind('.')
            lis.append(file[:z])
        else:
            z = file.rfind('.')
            lis.append(file[k+1:z])
        df,a,b,c,d,unique_vals = create(file)
        print(a)
        print(b)
        #res_dic
        f1.append(a)
        acc.append(b)
        precision.append(c)
        recall.append(d)
        if(flag == 1):
            df_old = pd.concat([df,df_old])
            flag = 2
        elif(flag==0):
            flag = 1
            df_old = df
        elif(flag==2):
            df_old = pd.concat([df,df_old])
    print(df_old.info())
    res_df['f1_score'] = f1
    res_df['accuracy'] = acc
    res_df['precision'] = precision
    res_df['recall'] = recall
    res_df.index = lis
    save_fig(res_df)
    print(res_df.head())
    print(df_old)
    df_old.fillna(value = 0,inplace = True)
    acc,f1,recall,precision = create_decision(df_old)
    df_final = pd.DataFrame([[acc,f1,recall,precision]],
    columns = ['accuracy','f1_score','recall','precision'])
    print(df_final)
    save_fig(df_final,'final_output')
```

References

[1] Saeid Abbasbandy and T Hajjari. “A new approach for ranking of trapezoidal fuzzy numbers”. In: *Computers & Mathematics with Applications* 57.3 (2009), pp. 413–419.

[2] Altyeb Altaher and Omar BaRukab. “Android malware classification based on ANFIS with fuzzy c-means clustering using significant application permissions”. In: *Turkish Journal of Electrical Engineering & Computer Sciences* 25.3 (2017), pp. 2232–2242.

[3] Mehran Arefkhani and Mohsen Soryani. “Malware clustering using image processing hashes”. In: *2015 9th Iranian Conference on Machine Vision and Image Processing (MVIP)*. IEEE. 2015, pp. 214–218.

[4] Saba Arshad et al. “Samadroid: a novel 3-level hybrid malware detection model for android operating system”. In: *IEEE Access* 6 (2018), pp. 4321–4339.

[5] Balaji Baskaran and Anca Ralescu. “A study of android malware detection techniques and machine learning”. In: (2016).

[6] Taniya Bhatia and Rishabh Kaushal. “Malware detection in android based on dynamic analysis”. In: *2017 International Conference on Cyber Security And Protection Of Digital Services (Cyber Security)*. IEEE. 2017, pp. 1–6.

[7] Ensieh Modiri Dovom et al. “Fuzzy pattern tree for edge malware detection and categorization in IoT”. In: *Journal of Systems Architecture* 97 (2019), pp. 1–7.

[8] Ming Fan et al. “Android malware familial classification and representative sample selection via frequent subgraph analysis”. In: *IEEE Transactions on Information Forensics and Security* 13.8 (2018), pp. 1890–1905.

[9] Ming Fan et al. “Dapasa: detecting android piggybacked apps through sensitive subgraph analysis”. In: *IEEE Transactions on Information Forensics and Security* 12.8 (2017), pp. 1772–1785.

[10] Yong Fang et al. “Android malware familial classification based on DEX file section features”. In: *IEEE Access* 8 (2020), pp. 10614–10627.

[11] Jianwen Fu et al. “Malware visualization for fine-grained classification”. In: *IEEE Access* 6 (2018), pp. 14510–14523.

[12] Hamed Haddadpajouh et al. "Mvfcc: A multi-view fuzzy consensus clustering model for malware threat attribution". In: *IEEE Access* 8 (2020), pp. 139188–139198.

[13] Jaemin Jung et al. "Android malware detection based on useful API calls and machine learning". In: *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. IEEE. 2018, pp. 175–178.

[14] Erich Peter Klement, Radko Mesiar, and Endre Pap. "On the order of triangular norms—comments on "A triangular norm hierarchy" by E. Cretu". In: *Fuzzy Sets and Systems* 131.3 (2002), pp. 409–413.

[15] Koodous. *Koodous*. Apr. 2021. URL: `https://koodous.com/`.

[16] Xiaojian Liu et al. "Multifamily Classification of Android Malware With a Fuzzy Strategy to Resist Polymorphic Familial Variants". In: *IEEE Access* 8 (2020), pp. 156900–156914.

[17] Zhuo Ma et al. "A combination method for android malware detection based on control flow graphs and machine learning algorithms". In: *IEEE access* 7 (2019), pp. 21235–21245.

[18] Arvind Mahindru and Paramvir Singh. "Dynamic permissions based android malware detection using machine learning techniques". In: *Proceedings of the 10th innovations in software engineering conference*. 2017, pp. 202–210.

[19] Ignacio Martı´n et al. "Android malware characterization using metadata and machine learning techniques". In: *Security and Communication Networks* 2018 (2018).

[20] Francesco Mercaldo and Andrea Saracino. "Not so Crisp, Malware! Fuzzy Classification of Android Malware Classes". In: *2018 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE)*. IEEE. 2018, pp. 1–7.

[21] Grayson Milbourne and Armando Orozco. "Android malware exposed". In: *Virus Bulletin Coference. Available: http://www. webroot. com/shared/pdf/Android-Malware-Exposed. pdf* (2012).

[22] Nikhil R Pal and James C Bezdek. "On cluster validity for the fuzzy c-means model". In: *IEEE Transactions on Fuzzy systems* 3.3 (1995), pp. 370–379.

[23] Justin Sahs and Latifur Khan. "A machine learning approach to android malware detection". In: *2012 European Intelligence and Security Informatics Conference*. IEEE. 2012, pp. 141–147.

[24] A-D Schmidt et al. "Static analysis of executables for collaborative malware detection on android". In: *2009 IEEE International Conference on Communications*. IEEE. 2009, pp. 1–5.

[25] Berthold Schweizer and Abe Sklar. *Probabilistic metric spaces*. Courier Corporation, 2011.

[26] Robin Senge and Eyke Hüllermeier. "Top-down induction of fuzzy pattern trees". In: *IEEE Transactions on Fuzzy Systems* 19.2 (2010), pp. 241–252.

[27] Asaf Shabtai et al. ""Andromaly": a behavioral malware detection framework for android devices". In: *Journal of Intelligent Information Systems* 38.1 (2012), pp. 161–190.

[28] Tayssir Touili et al. "Extracting Android malicious behaviors". In: *International Workshop on FORmal methods for Security Engineering*. Vol. 2. SCITEPRESS. 2017, pp. 714–723.

[29] Anil Utku, Ibrahim Alper Dogru, and M Ali Akcayol. "Permission based android malware detection with multilayer perceptron". In: *2018 26th Signal Processing and Communications Applications Conference (SIU)*. IEEE. 2018, pp. 1–4.

[30] VirusTotal. *Virus Total*. Apr. 2021. URL: `https://www.virustotal.com/gui/`.

[31] Haoyu Wang et al. "Rmvdroid: towards a reliable android malware dataset with app metadata". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 404–408.

[32] Ronald R Yager. "On ordered weighted averaging aggregation operators in multicriteria decisionmaking". In: *IEEE Transactions on systems, Man, and Cybernetics* 18.1 (1988), pp. 183–190.

[33] Win Zaw Zarni Aung. "Permission-based android malware detection". In: *International Journal of Scientific & Technology Research* 2.3 (2013), pp. 228–234.

[34] Wu Zhou et al. “Detecting repackaged smartphone applications in third-party android marketplaces”. In: *Proceedings of the second ACM conference on Data and Application Security and Privacy.* 2012, pp. 317–326.

[35] Yajin Zhou and Xuxian Jiang. “Dissecting android malware: Characterization and evolution”. In: *2012 IEEE symposium on security and privacy.* IEEE. 2012, pp. 95–109.

[36] Hui-Juan Zhu et al. “DroidDet: effective and robust detection of android malware using static analysis along with rotation forest model”. In: *Neurocomputing* 272 (2018), pp. 638–646.

Publications

1. The manuscript entitled ”Android Malware Family Classification using ensembling of FPT and FCM with decision tree” is accepted for Publication in ”Turkish Online Journal of Qualitative Inquiry; E-ISSN: 1309-6591; Indexing: SCOPUS (https://www.scopus.com/sourceid/21101019739)

2. The manuscript entitled ”Analysis of Android Malware Detection Techniques in Deep Learning” is published in SKIT RESEARCH JOURNAL; ISSN (PRINT) 2278-2508; ISSN (ONLINE) 2454-9673 in Volume 10 Issue 2 Year 2020 with following DOI: 10.47904/IJSKIT.10.2.2020.21-26. Online URL: https://ijskit.org/abstract.php?article=296