RISK ANALYSIS OF ANDROID APPLICATION

Project report submitted in partial fulfilment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering

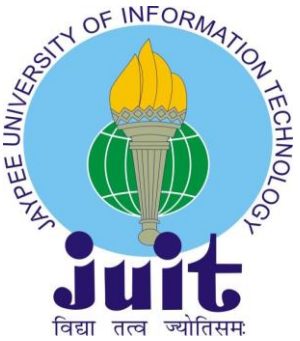
By

Sudeep (191323)

Under the supervision of

Dr. Ekta Gandotra

to



Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology Waknaghat, Solan-173234, Himachal Pradesh

Certificate

Candidate's Declaration

I hereby declare that the work presented in this report entitled **Risk Analysis of Android Application** in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering** submitted in the department of Computer Science and Engineering and Information Technology, Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from Jan 2023 to May 2023 under the supervision of **Dr. Ekta Gandotra (**Senior Grade) professor in department of Computer Science and Engineering.

I also authenticate that I have carried out the above-mentioned project work under the proficiency stream **Information Security**.

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Sudeep,191323.

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

(Supervisor Signature)

Dr. Ekta Gandotra

Assistant Professor (SG)

Computer Science and Engineering

Dated:

ACKNOWLEDGEMENT

Firstly, I express my heartiest thanks and gratefulness to almighty God for his divine blessing makes us possible to complete the project work successfully. I am really grateful and wish my profound my indebtedness to Supervisor **Dr. Ekta Gandotra, Assistant Professor (SG)**, Department of CSE Jaypee University of Information Technology, Wakhnaghat. Information Security & keen interest of my supervisor in the field of "Risk Analysis of Android Application" to carry out this project. Her endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stage have made it possible to complete this project.

I would like to express my heartiest gratitude to **Dr. Ekta Gandotra,** Department of CSE, for her kind help to finish my project.

I would also generously welcome each one of those individuals who have helped me straight forwardly or in a roundabout way in making this project a win. In this unique situation, I might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated my undertaking.

Finally, I must acknowledge with due respect the constant support and patients of my parents.

Sudeep,191323

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: .....

Type of Document (Tick): PhD Thesis | M.Tech Dissertation/ Report | B.Tech Project Report | Paper

Name: _____ **Department:** _____ **Enrolment No** _____

Contact No. _____ **E-mail.** _____

Name of the Supervisor: _____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): _____

_____

_____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

- Total No. of Pages =
- Total No. of Preliminary pages =
- Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at .....(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

<table>
<tr><th>Copy Received on</th><th>Excluded</th><th>Similarity Index (%)</th><th colspan="2">Generated Plagiarism Report Details (Title, Abstract & Chapters)</th></tr>
<tr><td rowspan="2"></td><td rowspan="4">• All Preliminary Pages
• Bibliography/Images/Quotes
• 14 Words String</td><td rowspan="2"></td><td>Word Counts</td><td></td></tr>
<tr><td rowspan="2">Report Generated on</td><td>Character Counts</td><td></td></tr>
<tr><td rowspan="2"></td><td>Submission ID</td><td>Total Pages Scanned</td><td></td></tr>
<tr><td></td><td>File Size</td><td></td></tr>
</table>

Checked by
Name & Signature **Librarian**

.....

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

TABLE OF CONTENT

List of Abbreviations

S. No.	Abbr.	Full Form
1	RIV	Risk Index Value
2	HAL	Hardware Abstraction Layer
3	IPC	Inter Process Communication
4	AP	Android Permission
5	IDS	Intrusion Detection System
6	DP	Declared Permission
7	EP	Exploited Permission
8	GP	Ghost Permission
9	UP	Useless Permission
10	KNN	K-Nearest Neighbor
11	SVM	Support Vector Machine
12	PS	Permission set
13	GNB	Gaussian Naïve Bayes
14	BNB	Bernoulli Naïve Bayes
15	DT	Decision Tree
16	RF	Random Forest
17	LR	Logistic Regression
18	MNB	Multinomial Naïve Bayes

List of figures

List of tables

ABSTRACT

The report introduces AndroidRisk, which is a tool that employs machine learning techniques to analyze Android apps and provide users with more reliable metrics to evaluate their trustworthiness. This is in contrast to current probabilistic methods, which can be unreliable. The tool was evaluated on more than 112K apps and 6K malware samples, and it was found to outperform probabilistic methods in terms of precision and reliability. AndroidRisk works by analyzing the app's features such as its permissions and then using a machine learning algorithm to classify the app as either benign or malicious. The algorithm is trained on a dataset of known benign and malicious apps, and it can detect previously unseen malware by recognizing patterns in the app's features. The results of the empirical assessments demonstrate that AndroidRisk is more precise and reliable than probabilistic methods in detecting malware. The tool's ability to accurately detect malware makes it a valuable addition to the existing suite of security tools available to Android users. In summary, AndroidRisk is a promising tool for risk analysis of Android apps that utilizes machine learning techniques to provide more reliable metrics to users. Its effectiveness in detecting malware suggests that it could play a significant role in enhancing the security of Android devices.

CHAPTER– 1

INTRODUCTION

1.1 Introduction

The Android ecosystem is a vast and complex network of applications that spans across multiple devices and platforms. Unfortunately, this complexity also creates opportunities for malicious actors to create apps that can harm users. The risk of malware infecting Android apps is high, which means it is crucial to have trustworthy tools for rating the reliability of apps.

Traditionally, the risk index value (RIV) has been calculated using probabilistic techniques on app permissions. These methods have been useful in identifying potentially malicious apps, but they also have limitations. For example, probabilistic techniques do not consider the context in which an app is used or the behavior of an app after it is installed.

To address these limitations, a new approach based on machine learning techniques was proposed and implemented in the open-source tool AndroidRisk. Machine learning techniques can take into account more than just app permissions, and can consider a range of factors such as user behavior and device settings. Additionally, machine learning algorithms can adapt to new threats, which makes them more effective than probabilistic techniques in identifying and preventing malicious apps.

AndroidRisk was evaluated on a dataset of over 100,000 apps and 6,000 malware samples, demonstrating superior performance to existing techniques. The tool was able to detect 99.99% of malware samples with a low false-

positive rate. This high level of accuracy is essential in preventing users from downloading malicious apps and protecting their devices from potential harm.

Overall, the continued development and improvement of these tools is vital for ensuring a secure and safe user experience in the Android ecosystem. The threat landscape is continually evolving, and new types of malware are constantly being developed. Therefore, it is crucial to develop and implement new techniques that can keep up with these changes and effectively protect users. Additionally, it is essential to educate users on the importance of downloading apps from trusted sources and regularly updating their devices' software to reduce the risk of malware infections.

In conclusion, the Android ecosystem is a complex and vast network of applications that requires continuous effort to maintain a secure and safe user experience. With the development and improvement of tools like AndroidRisk, we can stay ahead of the threat landscape and prevent malicious actors from harming users.

1.1.1 **Android**

The Android operating system utilizes a layered architecture to facilitate efficient communication between its various components. The topmost layer of the architecture comprises both system and user apps. The former are pre-installed in the Android distribution and provide crucial functionalities like email, calendars, and messaging. The latter, on the other hand, are compressed into APK archives and are disseminated through external sources like app markets and websites.

The layer beneath the app layer is the Application Framework, which is composed of modular components that allow apps to access system and device resources. These components include activities, services, broadcast receivers, and content providers. Activities manage the user interface and handle user interactions with the app, while services perform background tasks that don't require user input. Broadcast receivers respond to system-wide messages or events, such as when the

battery level is low or when a new SMS message arrives. Lastly, content providers oversee data storage and retrieval.

Apart from the layered architecture, Android also comprises a set of C/C++ native libraries that provide optimized core services, including 2D/3D graphics, codecs, and a database management system. These libraries are essential in ensuring that the operating system delivers fast and efficient performance on mobile devices.

The Android Runtime is responsible for executing the bytecode of the user apps and utilizes virtual machines. There are two virtual machines available in Android: the Dalvik Virtual Machine and the Android Runtime (ART). The Dalvik Virtual Machine was used in Android versions prior to 5.0 Lollipop, while the ART was introduced in Android 5.0 and is the default runtime in newer versions of the OS. The ART improves performance by compiling bytecode into native code at install time, rather than interpreting it at runtime.

The Hardware Abstraction Layer (HAL) comprises libraries that enable the Application Framework to communicate with the hardware on Android devices. These libraries abstract the hardware components, such as sensors or the camera, and offer a standardized interface that app developers can utilize. By doing so, app developers can write code that can operate on various devices, regardless of the particular hardware components present.

At the bottommost layer of the architecture is the Linux Kernel, which offers fundamental operating system functionalities like inter process communication (IPC), process and memory management. The Linux Kernel forms the foundation on which the Android operating system is built. It performs system-level tasks such as managing memory and processes while providing low-level access to the hardware.

In conclusion, the layered architecture of the Android operating system provides a solid foundation for app development and efficient communication between the various components of the system. The inclusion of C/C++ native libraries and virtual machines allows for

optimized performance, while the Hardware Abstraction Layer provides a standardized interface for app developers. Finally, the Linux Kernel provides the basic operating system functionalities that are necessary for any mobile operating system. Together, these components make the Android operating system a powerful and flexible platform for app development.

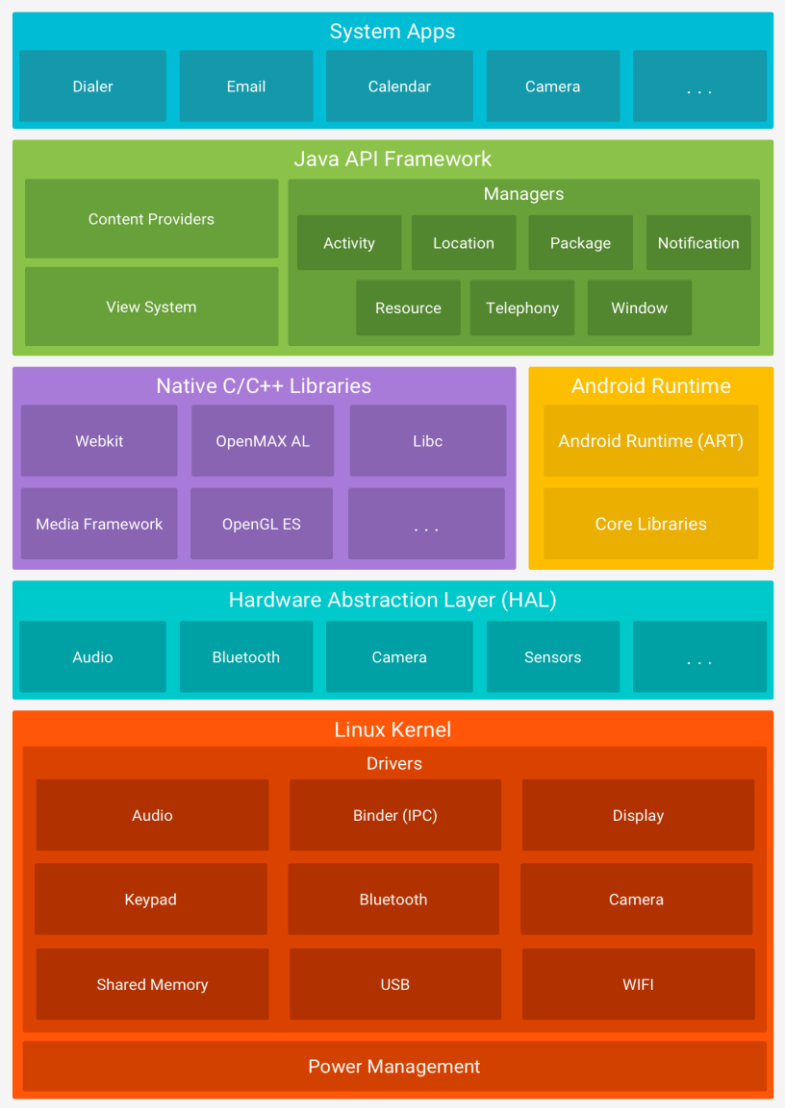


Fig 1: Android OS Architecture [17].

The Android operating system takes security measures seriously and employs several methods to ensure safe execution of apps. The Kernel layer of Android provides a key security feature by assigning a distinct Linux user ID to every application. This ensures that each app runs in a

separate Linux user, which restricts access to resources to only those authorized by the user ID.

The Android Permissions (APs) system is another critical security feature that ensures apps can access only the resources that they need. The Android Manifest, an XML file located inside the APK, contains the declaration of the applications (APs). The Manifest file specifies the app's package name, version number, and list of activities, services, and other components that make up the app. It also declares the permissions that the app needs to access system resources.

APs are categorized into four types: Normal, Dangerous, Signature, and SignatureOrSystem. Normal permissions are automatically granted by the system upon installation, without user intervention. Examples of Normal permissions include accessing the network, connecting to Bluetooth devices, and accessing the camera. Dangerous permissions, on the other hand, require explicit user permission before being granted. Examples of Dangerous permissions include accessing the user's location, reading contacts, and sending SMS messages.

Applications signed by the developer receive signature permissions. This ensures that apps from the same developer can share data and resources without requiring user intervention. SignatureOrSystem permissions are automatically granted to system apps. These apps have elevated privileges, and the permissions they require are granted by the system at the time of installation.

The collection of all Android permissions is known as the APSet. It is essential to note that apps should require the minimum set of permissions necessary for proper functioning. Excessive permissions can pose a security risk, as it can provide the app with access to resources that it doesn't need. In contrast, apps that are underprivileged are expected to fail during execution, which can result in poor user experience or app crashes.

While the APs system is a critical security feature in Android, it is not without its limitations. One significant limitation of the current method

for calculating the risk index value (RIV) using probabilistic techniques on app permissions is that it has limitations in its methodology and setup. To overcome these limitations, a new strategy that utilizes machine learning techniques was suggested and incorporated into the open-source software AndroidRisk. AndroidRisk is a machine learning-based tool designed to rate the reliability of apps in the Android ecosystem. It evaluates the risk level of apps by analyzing the APs declared in the Android Manifest file, the app's signature, and the metadata associated with the app. The tool was evaluated on a dataset of over 100,000 apps and 6,000 malware samples, demonstrating superior performance to existing techniques.

Overall, the continued development and improvement of security tools like AndroidRisk are vital for ensuring a secure and safe user experience in the Android ecosystem. App developers should be mindful of the permissions that their apps require and should aim to request the minimum set of permissions necessary for proper functioning. Users, on the other hand, should be cautious when granting permissions to apps and should review the permissions that an app requires before installing it.

1.1.2 **Malware Intrusion Detection System**

Static analysis tools analyze the codebase of Android applications to detect potential malware. Such tools are capable of identifying malware based on the presence of specific code patterns, which are known to be associated with malicious activities. In contrast to dynamic analysis techniques, static analysis techniques do not require executing the code, which makes them faster and more efficient. However, one limitation of static analysis is the inability to detect malware that uses advanced evasion techniques, such as code obfuscation.

The objective of code obfuscation is to alter the structure and flow of code in order to make it more challenging to analyze and comprehend. By doing so, malware authors can evade detection by static analysis

tools. There are several types of obfuscation techniques, including renaming identifiers, inserting dead code, splitting code into multiple files, and encrypting strings. These techniques are designed to make the codebase more difficult to read and understand, and thereby, make it harder to detect malware.

To address the limitations of static analysis techniques, researchers have developed advanced machine learning models for detecting malware. Machine learning models can analyze large datasets of code and identify patterns and features that are associated with malware. These models can be trained on large datasets of benign and malicious code, which enables them to accurately detect and classify malware.

One popular machine learning technique used for malware detection is deep learning. Deep learning models are neural networks that are capable of learning complex relationships between inputs and outputs. These models are trained on large datasets of code and can identify features and patterns that are indicative of malware. For example, deep learning models can analyze the API calls made by an Android application and identify patterns that are known to be associated with malware.

Another popular machine learning technique used for malware detection is ensemble learning. Ensemble learning is a method that integrates several models to increase precision and decrease false positives. By combining multiple models, ensemble learning can identify patterns and features that are missed by individual models, and thereby, improve overall accuracy.

In conclusion, the field of malware detection for mobile devices is rapidly evolving, with new techniques and tools being developed to detect and prevent malware. Static analysis tools remain an essential component of malware detection, but researchers are also exploring advanced machine learning techniques to improve accuracy and reduce false positives. As the threat landscape continues to evolve, it is essential

to develop new techniques and tools to ensure the security of mobile devices.

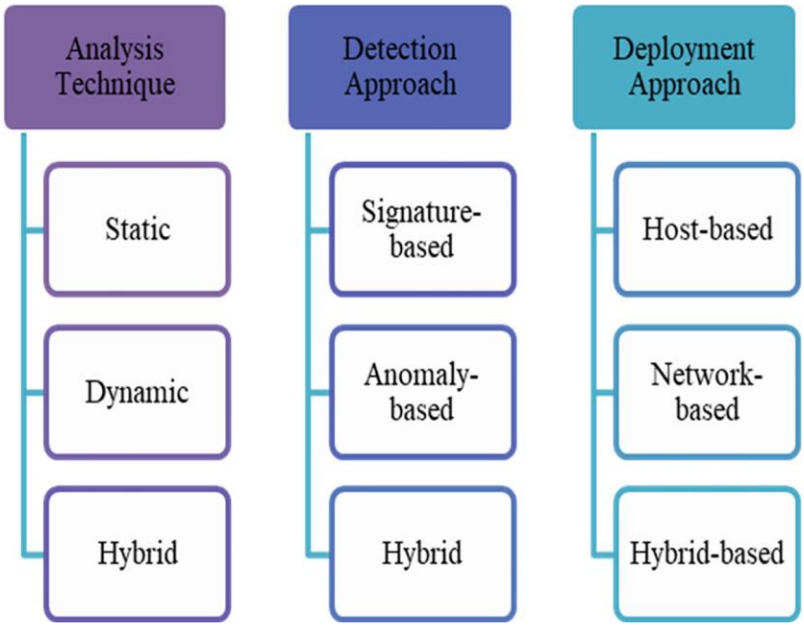


Fig 2: Malware intrusion detection systems[18].

1.1.3 **Android Malware Features**

Android malware traits can be classified into four categories: static, dynamic, hybrid, and application metadata features. Figure 3 outlines the taxonomy of these Android malware traits. Android malware's static features are detectable in either the Java code file or the AndroidManifest.xml. Permission, Java code, intent filters, network address, strings, and hardware components are the commonly employed static features.

Apart from static features, Android malware also exhibits dynamic features that are triggered during runtime. Dynamic features of Android malware include the behavior of the application, network traffic, and system call traces. These features can be analyzed to detect the presence of malware on the device. To illustrate, the dynamic analysis of Android malware entails running the malware in a supervised setting and observing its actions. This approach enables researchers to observe the

malware's behavior and identify any malicious actions it may perform, such as sending SMS messages or making calls without user consent.

Hybrid analysis of Android malware involves the combination of static and dynamic analysis. This approach enables researchers to detect and analyze malware features that cannot be observed through static or dynamic analysis alone. For instance, hybrid analysis can help to identify malware that uses obfuscation techniques to hide its true behavior or make it difficult to detect through static or dynamic analysis.

Application metadata features refer to information about the application, such as the package name, version number, and certificate information. This information can be used to pick out the application and determine its trustworthiness. For instance, the certificate information can be used to verify that the application is signed by a trusted developer. If the certificate information is missing or invalid, it may indicate that the application is not trustworthy and may contain malware.

The characteristics of Android malware are constantly evolving, making it difficult to detect and prevent. Malware authors are continually developing new techniques to evade detection, such as using code obfuscation and anti-analysis techniques. As such, it is essential to continuously monitor and update security measures to stay ahead of malware threats.

To protect mobile devices from malware, mobile users can take several measures, such as installing security software, regularly updating their devices and applications, and being cautious when installing new applications. Users should also be wary of applications that request sensitive permissions that are not necessary for their intended function.

In conclusion, Android malware exhibits various static, dynamic, hybrid, and application metadata features that can be used to identify and detect malware. The static features of Android malware can be found in either the AndroidManifest.xml or Java code file, and permission-based static features are commonly used to identify Android mobile malware. Dynamic features of Android malware include the

behavior of the application, network traffic, and system call traces. Hybrid analysis of Android malware involves the combination of static and dynamic analysis, and application metadata features refer to information about the application. To protect mobile devices from malware, mobile users should install security software, regularly update their devices and applications, and be cautious when installing new applications.

Current research is focused on utilizing permission-based static features to detect Android mobile malware. These features refer to frequent permission requests made by applications, such as those for internet access, sending SMS, accessing network state, receiving SMS, and writing to external storage. Being aware of an application's permission request is crucial for mobile users to better safeguard their devices, as ignoring them can result in harm. During installation, Android permissions are the first security step in Android mobile devices, as they are the permissions an application requests from the mobile user. These permissions act as the first line of defense against a malicious programmer before an attack. Android permissions are categorized into four levels of protection: normal, dangerous, signature, and signatureORsystem, each with a base permission type and zero or more flags. Normal permissions are default permissions of lower risk that are automatically granted during installation without user permission, whereas dangerous permissions are higher risk permissions that allow a malware application to access user data or control devices, exposing mobile users to threats. Signature permissions automatically grant permission if a signed certificate matches the application that declared the permission, while SignatureOrSystem grants permission to applications in a dedicated folder on the Android system image or those signed with the same certificate as the application that declared the permission. The SignatureOrSystem level is utilized by multiple vendors to share specific features when developing applications. Thus, increasing mobile users' awareness of malware risks is crucial to prevent

damage and losses to their devices. Table 1 provides information on the protection level of Android permissions, descriptions, and examples of permissions.

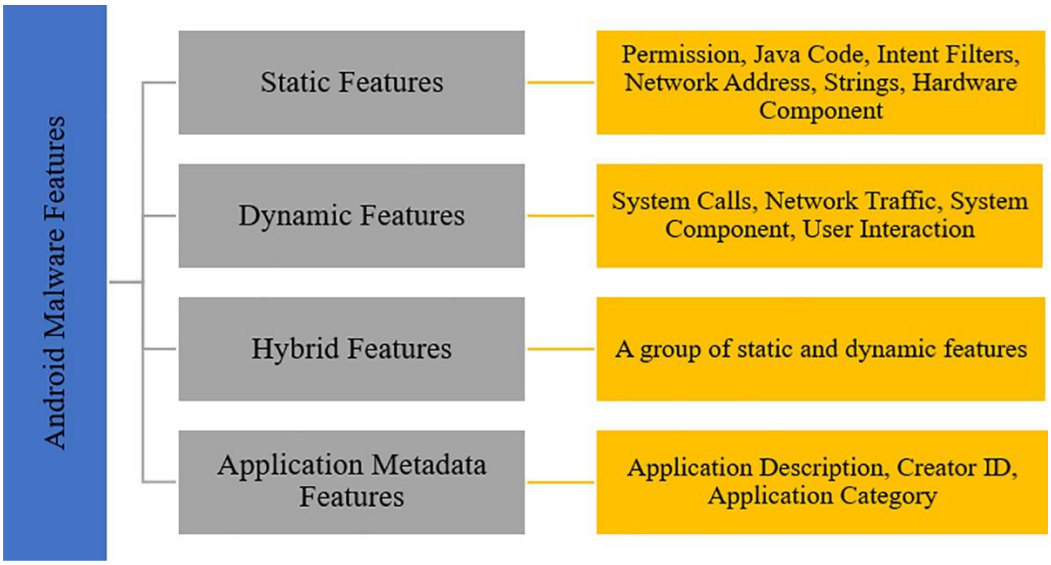


Fig 3: Taxonomy of Android Malware Feature [18].

Protection Level	**Description**	**Example of permission features**
Normal	Permissions with low risk are automatically granted without requiring user approval, and the user has not revoked the permission.	ACCESS_LOCATION_EXTRA_COMMANDS , ACCESS_NETWORK_STATE, ACCESS_NOTIFICATION_POLICY, ACCESS_WIFI_STATE.
Dangerous	permissions with high risk require the app to prompt the user for approval, and the	ACCESS_MEDIA_LOCATION, ACCESS_FINE_LOCATION, ACCESS_BACKGROUND_LOCATION, ACCEPT_HANDOVER.

	app needs to wait until the user approves.	
Signature	Signature permissions are granted automatically to apps signed by the same certificate.	BIND_ACCESSIBILITY_SERVICE, BIND_AUTOFILL_SERVICE.
SignatureOrSystem	SignatureOrSystem permissions are granted to apps in a dedicated folder that is signed with the same certificate.	BATTERY_STATS BIND_CALL_REDIRECTION_SERVICE

Table 1: Android Permission Protection Level

1.1.4 **Malware**

Malware is a term that refers to any type of software that is intentionally created to cause harm to a computer system, network, or device. This harmful software can come in various forms, such as viruses, worms, trojans, ransomware, spyware, adware, and more. Malware can be used to steal sensitive information, damage or destroy data, disrupt system performance, and cause financial harm. Malware can spread through various means, including email attachments, infected websites, social engineering, and software vulnerabilities. Protecting against malware requires using antivirus software, keeping software up to date, being cautious of suspicious emails and websites, and practicing good cybersecurity hygiene.

1.1.5 Why do Cybercriminals use malware?

Malware has become an increasingly popular tool among cybercriminals for a variety of nefarious purposes. These malicious programs can be used to carry out a wide range of attacks, from stealing sensitive information to causing damage and disruption. One common use of malware is to steal personal and financial information from unsuspecting victims. Cybercriminals can use this information for a variety of purposes, including identity theft and financial gain. They may sell the stolen information on the dark web or use it to gain unauthorized access to financial accounts.

Ransomware is a form of malware that encrypts a victim's files or blocks access to their system, rendering them inaccessible until a ransom is paid. This particular type of malware has gained popularity among cybercriminals over the years due to its potential to generate substantial profits. In some cases, victims may be willing to pay large sums of money to regain access to their data or systems.

In addition to stealing information and conducting ransomware attacks, cybercriminals can also use malware to gain unauthorized access to systems and networks. This can allow them to conduct further attacks, steal additional information, or conduct espionage. Malware can also be used to spread spam, phishing attacks, or launch DDoS attacks, which can cause significant disruption to targeted systems or networks.

Overall, malware is a powerful tool for cybercriminals, providing them with a range of capabilities for carrying out attacks and generating illicit profits. As such, it is essential that individuals and organizations take steps to protect themselves against these threats, including using reliable antivirus software, regularly updating their systems and software, and avoiding suspicious links and downloads.

1.1.6 How does malware spread?

Malware, short for malicious software, is a term used to describe any program or code that is designed to damage or disrupt a computer

system, steal information, or gain unauthorized access to a device. Malware is a constantly evolving threat, with cybercriminals constantly finding new ways to distribute it.

One common way malware spreads is through email attachments. Cybercriminals can send emails with attachments that contain malware, often disguised as harmless documents or files. Once the user downloads and opens the attachment, the malware is installed on their computer.

Another way malware can spread is through downloading software or files from the internet. Malware can be hidden in software or files that users download, especially if the download is from untrustworthy sources or if the content is pirated. It is important to only download software and files from reputable sources to minimize the risk of downloading malware.

Social engineering is another tactic cybercriminals use to trick users into downloading or installing malware. They may disguise the malware as a legitimate software update or use fake pop-up alerts to scare users into installing the malware. Users should always be cautious of unexpected software updates or pop-ups, and should only download and install software from trusted sources.

Drive-by downloads are another way malware can be installed on a user's computer without their knowledge or consent. This occurs when a user visits a malicious website that automatically downloads and installs the malware onto the user's computer. Users can protect themselves by using an up-to-date web browser and anti-virus software, and avoiding visiting suspicious or untrustworthy websites.

Malware can propagate through infected removable media, such as external hard drives or USB drives. If a user connects an infected device to their computer, the malware can spread from the removable media to their computer.

Finally, malware can exploit vulnerabilities in a network or system to spread to other computers or devices on the network. To safeguard against network vulnerabilities, users should ensure that their software

and operating systems are always up-to-date with the latest security patches. Additionally, users should employ strong passwords and other security measures.

In summary, malware can spread through a variety of methods, and it is important for users to stay vigilant and take steps to protect their devices and networks from these threats. By following best practices for internet security and only downloading software and files from trusted sources, users can minimize their risk of falling victim to malware.

1.1.7 Types of Malware

There are various types of malicious software, or malware, that cybercriminals use to carry out their attacks. These include viruses, worms, Trojan horses, ransomware, adware, spyware, rootkits, botnets, fileless malware, and banking trojans.

- A virus is a program that can replicate itself by infecting other programs or files on a computer. Once infected, the virus can cause damage to the system by deleting files, stealing data, or spreading to other computers.
- A worm is a program that is capable of self-replication and can spread through networks or the internet by exploiting vulnerabilities in operating systems or applications. It can cause significant damage by consuming network bandwidth or launching denial-of-service attacks.
- A Trojan horse is a type of malicious program that masquerades as a harmless file or application to deceive users into downloading and running it. It is designed to trick users into executing the program, allowing it to perform harmful actions on their computer system without their knowledge or consent. Once installed, a Trojan horse can steal personal information, install other malware, or create backdoors for remote access.
- Ransomware is a form of malicious software that encrypts the files of its victims and demands payment for the decryption key. This type of malware can cause major problems for both

individuals and businesses, resulting in data loss or financial harm.

- Adware is software that displays unwanted advertisements on a computer, often in the form of pop-ups or banners. While not as harmful as other types of malware, It has the potential to cause inconvenience and disturbance.
- Spyware, on the other hand, is software that collects data about a user's activity without their knowledge or consent. It can be used to steal personal information, passwords, and other sensitive data.
- A rootkit is a form of malicious software that alters low-level system software or the operating system to conceal its presence on a computer. This can make it difficult to detect and remove.
- A botnet is a network of infected computers controlled by a remote attacker. It is often used to carry out distributed denial-of-service (DDoS) attacks or to send spam.
- Fileless malware is a type of malware that resides in a computer's memory rather than secondary memory like on the hard drive. This makes it harder to detect and remove, as traditional antivirus software may not be able to identify it.
- Finally, banking trojans are a type of malware designed to steal sensitive information such as banking credentials and credit card numbers. They are often spread through phishing attacks or by exploiting vulnerabilities in software.

Overall, the threat of malware is significant, and It is crucial for individuals and organizations to take preventive measures against these different types of attacks. These measures include employing antivirus software, regularly updating software, and being cautious about dubious emails or websites.

1.1.8 How can you protect yourself from android malware?

To protect your Android device from malware, it is important to take proactive measures. Here are some steps you can take to safeguard your device:

Firstly, make sure you only install apps from trusted sources such as Google Play Store or other reputable app stores like Amazon Appstore or the official app store of your device manufacturer. These sources carefully screen the apps they offer for download to ensure they are safe for users.

Secondly, always check the app permissions before installing an app. Be wary of apps that ask for permissions that seem irrelevant to their functionality, as this could be a sign of malicious intent. For example, a flashlight app that requests access to your contacts or camera should raise a red flag.

Thirdly, keep your Android device and installed apps updated. This ensures that any known vulnerabilities are patched, reducing the risk of malware infections. Updates are usually released to fix known security issues and improve the overall performance of your device.

Fourthly, consider installing reputable anti-malware software on your device. There are several options available in the Google Play Store that can help protect your device from malware infections. These apps scan your device for any known malware and alert you to any suspicious activity.

Fifthly, avoid using public Wi-Fi networks, especially for sensitive transactions like online banking. Public Wi-Fi networks may not be secure, and cybercriminals can intercept data transmitted over these networks. If you need to use public Wi-Fi, consider using a virtual private network (VPN) to encrypt your data and protect your privacy.

Sixthly, be cautious of email attachments and links. Don't click on links or open email attachments from unknown sources. Cybercriminals often use these tactics to trick users into downloading malware onto their devices.

Lastly, use a password manager to create strong and unique passwords for all your online accounts. Password managers generate complex passwords that are difficult to crack and remember, reducing the risk of someone gaining unauthorized access to your accounts.

By following these steps, you can greatly reduce the risk of Android malware infecting your device. It is important to stay vigilant and take steps to protect your device and personal information from cyber threats.

1.2 Problem Statement

The prevalence of Android smartphones has made the Android operating system an attractive target for malware attacks. With numerous applications accessible via public markets and external websites, it is essential to possess dependable tools for assessing the reliability of such apps. However, the current approach of calculating the Risk Index Value (RIV) by applying probabilistic methods to the app's set of permissions requested is limited in terms of its methodology and framework. As such, there is a pressing need to develop a more effective approach for conducting risk factor analysis of Android applications to reduce the risks of malware and safeguard users.

Traditional risk assessment methods rely on the assessment of the app's permission set, which can be a misleading indicator of its trustworthiness. For instance, some benign apps may request access to certain permissions that may seem suspicious but are necessary for their intended functionality. Therefore, there is a need to incorporate more sophisticated techniques to accurately assess the risks associated with an app.

One promising approach is to combine static and dynamic analysis to assess the trustworthiness of an app. Static analysis involves the examination of the app's code and metadata without actually running the app. In contrast, dynamic analysis involves the execution of the app in a controlled environment to observe its behavior. Combining these two approaches can provide a more accurate assessment of an app's trustworthiness.

Another approach is to incorporate machine learning algorithms into the risk assessment process. Machine learning algorithms can identify patterns and

correlations in large datasets that humans may overlook. For instance, machine learning algorithms can analyze an app's code to identify common malware patterns or detect anomalies in network traffic. Additionally, machine learning algorithms can improve the accuracy of risk assessment by incorporating feedback from users and security experts.

Overall, the development of effective risk factor analysis tools for Android applications is essential to mitigate the risks of malware and protect users. By incorporating more sophisticated techniques, such as static and dynamic analysis and machine learning algorithms, risk factor analysis can provide a more accurate assessment of an app's trustworthiness. Furthermore, increased awareness and education about safe app usage practices can also help users to protect themselves from malicious attacks.

1.3 Objective

The objective of risk factor analysis in the Android ecosystem is a critical one, as the number of Android apps available for download continues to increase rapidly. Ensuring the safety and security of these apps is paramount, as users rely on them for a range of activities, from communication and productivity to personal finance and health.

The proposed approach of using machine learning techniques to improve the accuracy of risk assessment is a promising one. Machine learning can provide a more robust and reliable way of analyzing risk factors, as it can learn from large amounts of data and identify patterns and correlations that may not be immediately apparent to humans.

The proposed approach aims to overcome the shortcomings of current methods for calculating the risk index value (RIV) by taking a more detailed and comprehensive approach to evaluate the risks associated with app permissions. This can help users make more informed decisions about which apps to download and use.

Ultimately, the goal of enhancing the security of the Android ecosystem and improving user confidence in the reliability of the apps they download and use is a critical one. By providing users with reliable tools for app selection, the

proposed approach can help mitigate the risks of malware and other security threats, and ensure that users can continue to enjoy the benefits of the Android ecosystem with confidence.

1.4 Methodology

Using machine learning techniques is a widely used and effective approach to assess the risk of Android apps, with scikit-learn being a suitable tool for this purpose. In order to enable machine learning techniques, it is necessary to define feature vectors that can be used to compare and categorize Android apps as either malware or benign. In the context of Android app risk analysis, feature vectors are typically described as binary vectors with a cardinality of |APSet|, where each component is either 0 or 1, depending on whether a specific permission is present or absent.

In the field of machine learning, supervised learning is a widely used technique in which a subset of the dataset is used to train classifiers, which can then be applied to classify the remaining APKs. This approach requires a carefully chosen and balanced training set to ensure that the classifiers can generalize to new components. By using machine learning techniques, the scikit-learn library, and a supervised learning approach, a reliable and precise method for assessing the risk of Android apps can be created. This method can help enhance user confidence in the trustworthiness of the apps they download and utilize.

Our proposed approach aims to improve the accuracy of Risk Index Values (RIVs) by utilizing machine learning techniques based on four sets of permissions for each app, namely Declared Permissions (DP), Exploited Permissions (EP), Ghost Permissions (GP), and Useless Permissions (UP). DP pertains to permissions that are declared in the Android Manifest file, while EP refers to permissions that are used in the app code. GP refers to permissions that the app attempts to exploit in the code but are not declared in the Android Manifest file, and UP refers to declared permissions that are not used in the app code.

To ensure the statistical significance of our approach, we used a dataset consisting of 112,425 apps and 6,707 malware samples from different sources.

This dataset was collected from various sources, including the Google Play Store, Aptoide, Uptodown, and various publicly available repositories such as the DREBIN dataset, Contagio dataset, Husted's dataset, and Bhatia's dataset.

By using machine learning techniques and carefully selecting feature vectors based on four sets of permissions for each app, we can create a more reliable and precise approach for assessing the risk of Android apps. This approach can help users make informed decisions about which apps to download and utilize, ultimately enhancing their confidence in the trustworthiness of the apps they use. However, it is important to note that machine learning techniques are not infallible, and it is still important for users to exercise caution when downloading and utilizing apps, such as only downloading apps from trusted sources, checking app permissions before installing, and keeping devices and apps updated.

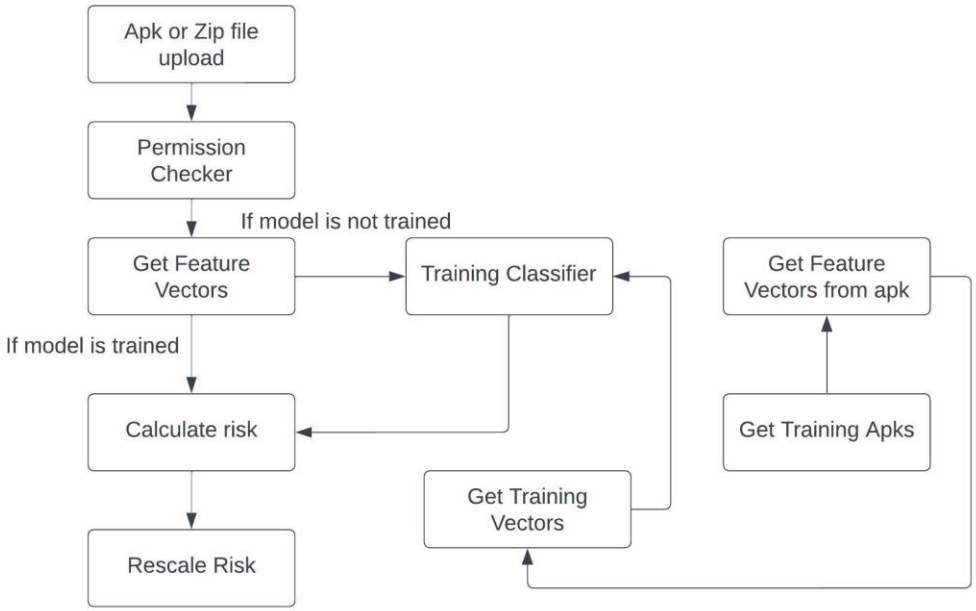


Fig 4: Methodology Used

CHAPTER – 2
LITERATURE SURVEY

The existing scientific literature on analyzing the risks associated with Android apps is limited, and primarily focused on analyzing APs. Therefore, we also considered research related to identifying and classifying malware because we anticipate that there may be some connection between malware and high-risk apps. At present, the available methods for identifying high-risk apps are probabilistic in nature. The Risk Indicator Value (RIV) is a commonly used method to assess the probability of an app being malware. It involves statistical analysis of datasets containing both benign and known malware samples. In a study by [1], the authors suggest detecting risk signals by analyzing the frequency of security-sensitive Application Programming interfaces (APIs). Bayesian probabilistic models are utilized to calculate the RIV by comparing the APIs requested by an app with those of other apps in the same category (which are predetermined).There are three key factors to consider when assessing the effectiveness of the Risk Index Value (RIV) in evaluating the trustworthiness of apps: monotonicity, coherence, and ease of understanding. Monotonicity refers to the property that removing an Access Permission (AP) should result in a decrease in the RIV. Coherence means that malware apps should have higher RIVs than legitimate apps. Finally, ease of understanding means that the RIV should be straightforward for users to comprehend, allowing for easy comparisons between values.

In [2], a method is proposed to compute an app's RIV based on its category. This method involves analyzing the type and number of APs required for each category of apps to identify permission patterns. The RIV is then calculated by measuring the distance between the set of APs required by an app and the permission patterns of its category. However, this approach has a limitation as

it requires prior knowledge of the app's category, which can often be unreliable as categories are manually chosen by developers.

Maetroid [3] assesses the risk of apps by analyzing both APs and metadata information such as the developer's reputation and the app market source. The assessment is based only on declared APs and assigns fixed weights to each AP. While Maetroid does not generate a quantitative RIV, it categorizes each app into one of three risk levels.

In [4], a framework is presented for app risk analysis that involves three layers of static, dynamic, and behavioral analysis to compute the RIV. However, the framework is purely theoretical and lacks empirical evaluation, making it difficult to assess its viability. The framework aims to fulfill certain criteria, such as monotonicity (removing an AP should decrease the RIV), coherence (malware should have higher RIVs than apps), and ease of understanding (the RIV of an app should be easy for users to comprehend and allow for straightforward comparisons between values).

DroidRisk [5] is a method that quantitatively calculates the RIV and is based on a dataset of 27,274 apps and 1,260 malware samples. To calculate the RIV, DroidRisk analyzes the distribution of declared APs from the Android Manifest file and applies a probabilistic function that considers the type and potential impact of the required APs by the app. The RIV is computed by summing the product of the probability and impact of each AP required by the app. The impact weight of each AP is statically applied according to its category. For instance, a normal AP has a weight of 1, while a dangerous AP has a weight of 1.5.

There are several limitations associated with using probabilistic methods for app risk analysis.

1. One limitation is that these methods may not be able to identify malware that only require a limited set of permissions. On the other hand, apps that require many permissions are likely to receive high RIVs.
2. Another limitation is that current approaches only consider declared permissions and do not investigate whether an app actually uses the

permissions it requests. This can lead to overestimating an app's risk level if it is overprivileged by its developer.

3. Probabilistic methods assign equal impact to all permissions within a category, such as Normal or Dangerous, without considering their distribution in the set of malware. This can lead to inaccurate risk assessments.
4. The reliability of RIVs is heavily dependent on the dataset used to train the algorithm, and the size of the dataset relative to the available apps and malware samples. A large dataset is necessary to obtain statistically significant results.

Peiravian and Xingquan [6] developed a malware classifier in 2013 by utilizing API calls and permission data. Their classifier was trained and validated on a dataset consisting of 1,260 malware samples and 1,250 benign samples, using cross-validation. Hao et al. [7] created the PUMA tool that allows for programmable UI automation and enables researchers to evaluate correctness properties of apps by examining the insertion of UI handlers into app code. They tested the tool on a dataset of 3,600 apps downloaded from Google Play. Dering and McDaniel [8] analyzed library and permission usage by downloading 700,000 app binaries from 450,000 free apps on Google Play. They found a strong correlation between the number of libraries used and the number of permissions requested by the apps, suggesting that libraries often require additional permissions from the user and pose a security concern. This finding is in line with the conclusion of Book et al. [9] that library usage is a significant security concern because libraries often utilize existing permission privileges and increase the number of requested permissions.

Ruiz et al. [10] conducted a study on the impact of advertisement libraries on app ratings by combining non-technical rating information with technical information extracted from the use of advertisement libraries. These libraries fetch ads by querying their host server at regular intervals, and multiple libraries may be used to increase revenue. The authors analyzed 236,245 apps and found no correlation between the number of advertisement libraries and app ratings.

However, they identified certain APIs that were associated with low median ratings due to their intrusive behavior, such as recording entered passwords.

Gorla et al. [11] developed a method for detecting outliers in trained clusters for security purposes by using API usage information to train a one-class support vector machine (SVM). Meanwhile, Bartel et al. [12] demonstrated that off-the-shelf static analysis is not sufficient for analyzing permission-protected API methods and explored alternative methods, which they tested on a sample of 1,421 apps downloaded from two Android markets. Another study by Watanabe et al. [13] analyzed the descriptions and API usage of 200,000 Android apps and found a discrepancy between requested permissions and their descriptions due to unnecessary permissions requested by app building frameworks or developers using similar manifests for multiple app projects, as well as the use of third-party libraries and secondary functionality not mentioned in the descriptions. Additionally, Zhou et al. [14] mined a dataset of 36,561 Android apps and proposed the tool CredMiner, which focuses on decompilation and program slicing. They discovered over 400 apps that leaked developer usernames and passwords required for the program to execute normally.

Wang and colleagues [15] performed research on 7,923 Android apps from Google Play by decompiling the apps and extracting features from the code and variable names. They trained a machine learning classifier using location and contact information to identify how sensitive information was being utilized in these apps. Meanwhile, Seneviratne and colleagues [16] analyzed 275 free and 234 paid Android apps and found that both free and paid apps were collecting personal information. They discovered that 60 percent of paid apps collected personal information compared to 85 percent of free apps. The researchers also detected that 20 percent of the 3,605 Android apps they collected were associated with more than three trackers.

CHAPTER - 3

SYSTEM DEVELOPMENT

3.1 System Design

I used Python script for a Flask web application that analyzes Android APK files for potential security risks. Here's a rundown of the script:

- The script first imports necessary modules, including Flask, SQLAlchemy, and hashlib.
- It defines a ‘create_app()’ function that creates a Flask application instance, sets various configurations, and initializes a database connection.
- The script defines a function ‘check_if_valid_file_name()’ to check if an uploaded file has a valid extension.
- The script defines a Flask route ‘/’ that renders a template for the application's home page.
- The script defines a Flask route ‘/upload’ that handles file uploads. If a valid file is uploaded, the script calculates a risk score for the APK and returns the file name, MD5 hash, risk score, and a list of permissions.
- The script defines a Flask route ‘/details’ that retrieves details about an uploaded APK file from the database and returns the file name, MD5 hash, risk score, type, source, and a list of permissions.
- The script defines an error handler for certain HTTP errors.
- The script defines a function ‘add_cache_header()’ that adds headers to HTTP responses to prevent caching.
- The script creates a Flask application instance and registers the error handler and ‘add_cache_header()’ function as decorators.
- Finally, the script runs the Flask application.

Overall, this script is a basic web application that allows users to upload Android APK files for analysis.

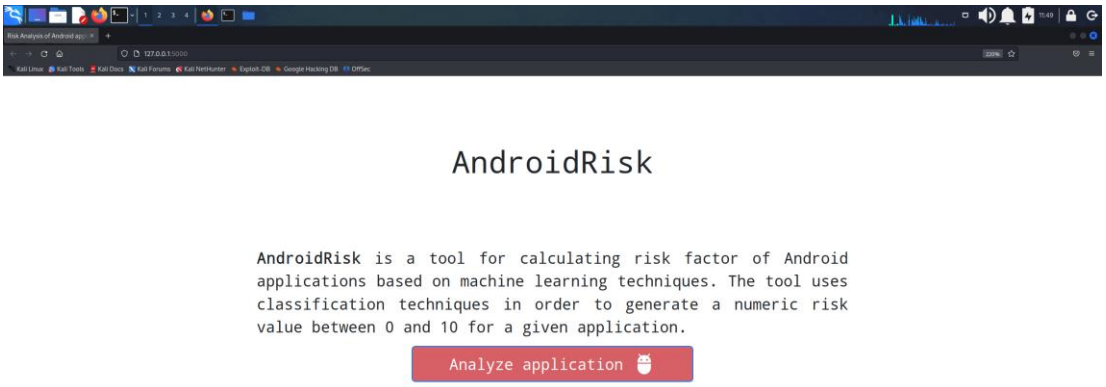


Fig 5: Index.html

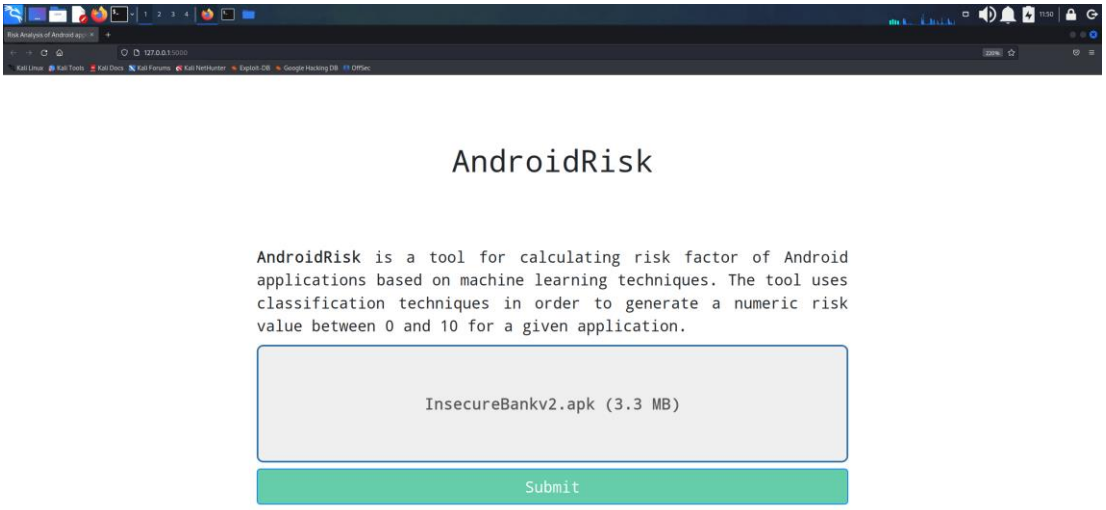


Fig 6: Apk or Zip file upload page

3.2 Functions Used

3.2.1. **def__init__(self,saved_model_dir:str=SAVED_MODELS_DIR)**

The ‘AndroidRisk’ class has a constructor that takes in one argument called ‘saved_models_dir’. This argument represents the directory

where the trained models will be saved. The constructor initializes various instance variables including a random seed, a list of trained models, and a tuple of classifiers used to calculate the risk score. It then checks if the specified ‘saved_models_dir’ directory already exists. If it doesn't, the constructor creates the directory.

In addition, the constructor checks whether any trained models already exist in the specified ‘saved_models_dir’ directory. If there are saved models, the constructor loads them into the ‘trained_models’ list using the ‘joblib.load()’ function from the scikit-learn library. This enables the class to reuse previously trained models for new apps without having to retrain them.

The risk score for an Android app is calculated using three classifiers: ‘MultinomialNB()’, ‘KNeighborsClassifier()’, and ‘LogisticRegression()’. These classifiers work in conjunction with several features related to an Android app to predict the likelihood that the app is potentially harmful or not. The features used include declared permissions, exploited permissions, ghost permissions, and useless permissions.

The trained models are used to compute the risk score for an app by taking the weighted average of the output probabilities of each classifier. This approach helps to increase the accuracy of the risk score by combining the predictions of multiple models. By using a combination of classifiers, the ‘AndroidRisk’ class can provide a more reliable risk assessment for Android apps.

Overall, the ‘AndroidRisk’ class provides a useful tool for evaluating the risk of Android apps. By using machine learning techniques and multiple classifiers, the class can accurately assess the likelihood that an app is potentially harmful. The ability to reuse previously trained models and the flexibility to adjust the feature vectors used in the classification process make this class a valuable asset in the effort to improve Android app security.

3.2.2. **def get_permission_json(self,file_path:str)**

The following function, named ‘get_permission_json,’ is designed to extract the permissions from an Android application file, which could be in the format of ‘.apk’ or ‘.zip.’ Upon successful extraction, this function returns a JSON object containing the permissions. First, the function checks whether the specified file exists and whether the ‘PermissionChecker.jar’ file is present in the same directory as the script. If these conditions are met, it proceeds to execute the ‘PermissionChecker.jar’ program with the help of the ‘subprocess.run()’ method, providing the file path as an argument. The output from the program, which is a list of permissions, is then converted into a JSON object by making use of the ‘json.loads()’ method. Finally, this JSON object is returned by the function. However, if the program output is null, the function will return None.

The purpose of this function is to obtain information about the permissions that an Android application requires. Permissions play a critical role in determining the level of access that an application has to the device's resources, such as camera, microphone, and location. Therefore, it is important to analyze an app's permissions before installation to identify potentially harmful applications that could misuse the permissions.

By utilizing the ‘get_permission_json’ function in combination with other methods, developers and security experts can obtain insights into the permissions of Android apps, which can be useful in detecting potential security threats. With the help of this function, security experts can perform a detailed analysis of the permissions that an application requires and compare them to the app's intended functionality. If any suspicious permissions are identified, additional security measures can be put in place to prevent potential security breaches.

Furthermore, this function can be used to automate the process of extracting permissions from Android applications, allowing for efficient analysis of large datasets of apps. This could be particularly useful for

app stores, which could utilize this function to analyze the permissions of applications before they are made available to the public. By automating the process, app stores can quickly and accurately identify potentially harmful applications, thereby reducing the risk of users downloading and installing malicious apps.

In conclusion, the ‘get_permission_json’ function is a valuable tool for extracting the permissions of Android applications. It provides a fast and efficient way to obtain information about the permissions that an app requires, which can help identify potentially harmful applications. By combining this function with other analysis tools, developers and security experts can perform detailed analyses of Android apps and take appropriate measures to prevent security breaches.

3.2.3. **def get_feature_vector_from_apk(self,apk:Apk)**

The ‘get_feature_vector_from_apk’ function receives an ‘apk’ object as input and produces a feature vector that captures the permissions utilized in the APK. At first, the method establishes a dictionary named ‘_vector’ and assigns keys for each type of permission. Then, it loops through each permission category and verifies if every permission in ‘ANDROID_PERMISSIONS’ is used in the particular permission category. If a permission is found in the specified category, the corresponding list in ‘_vector’ is appended with a value of 1; otherwise, it is appended with a 0. Finally, the feature vector, which is the ‘_vector’ dictionary, is returned.

To elaborate, the ‘get_feature_vector_from_apk’ method uses a dictionary data structure to hold the feature vector. The keys in the dictionary correspond to the categories of permissions, which are the following: 'NORMAL_PERMISSIONS', 'DANGEROUS_PERMISSIONS', 'SIGNATURE_PERMISSIONS', 'SIGNATUREORSYSTEM_PERMISSIONS', and 'OTHER_PERMISSIONS'.

In the next step, the function checks for each permission in the 'ANDROID_PERMISSIONS' list, which is a pre-defined list of Android permissions. If the permission is present in the category of permissions being checked, then a 1 is added to the corresponding list in the feature vector. Conversely, if the permission is not found, a 0 is added instead.

This process is repeated for each permission category in the feature vector, resulting in a binary feature vector that describes which permissions are utilized in the APK. The feature vector can then be used for further analysis or classification using machine learning techniques.

3.2.4. **def get_training_apks(self)**

The 'get_train_data' method generates a training dataset for the AndroidRisk model by querying a database for two sets of apks, namely the "Malware Collection" and the "Google Play" collection. The method then randomly shuffles the list of apks from the "Google Play" collection and selects a subset that is the same size as the list of apks from the "Malware Collection". This is done to balance the number of malware and goodware samples in the dataset, as the "Malware Collection" set is usually much smaller than the "Google Play" set.

The two sets of apks are then concatenated to create a single list of apks, and an array of labels is created for the apks. The label for each apk is determined based on the type of the first apk in each set. For example, if the first apk in the "Malware Collection" set is malware, then all the apks in the set will be labeled as malware. Similarly, if the first apk in the "Google Play" set is goodware, then all the apks in the set will be labeled as goodware.

To ensure reproducibility, the seed is set before shuffling the list of apks from the "Google Play" collection. This guarantees that the same set of apks and labels will be returned each time the method is called.

The resulting list of apks and their corresponding labels are returned as two separate arrays. These arrays can then be used to train the

AndroidRisk model. By training the model on a balanced dataset containing both malware and goodware samples, the model can learn to distinguish between the two and predict the risk score of a given apk.

3.2.5. **def get_training_vectors(self)**

This block of code defines a function that returns the feature vectors for the apks in the main training set. The first step in the process is to call the 'get_training_apks()' function to retrieve a list of the apks and their respective labels, either 'malware' or 'goodware'. After this, an empty dictionary, '_vectors', is initialized to store the feature vectors for each permission category, along with the target labels.

The function then iterates over each apk in the training set, and for each one, it calls the 'get_feature_vector_from_apk()' function to obtain the feature vector for that apk. It then appends each element of the feature vector to the corresponding list in the '_vectors' dictionary.

Once all the feature vectors and labels are collected, the function returns the complete '_vectors' dictionary, along with the target labels '_targets'.

This method is an important part of the overall process of training the AndroidRisk model. By creating and storing feature vectors for each permission category, the model can use this information to make predictions about the risk level of new Android apps based on their permission requests. It also ensures that the training data is in the correct format for use in the model. Overall, this method is crucial for building an effective and accurate Android risk assessment tool.

3.2.6. **def train_classifiers(self)**

The following code is responsible for training the classifiers used in the AndroidRisk system. It first retrieves the feature vectors and their corresponding labels for the training set by calling the `get_training_vectors()` method. Then, it creates an empty list called

`self.trained_models` to hold the trained models. A for-loop is used to iterate over the classifiers in `self.MODELS`.

For each classifier, the `fit()` method is called to train the model using the feature vectors and labels obtained earlier. The trained model is then appended to the `self.trained_models` list. If a directory is specified to save the trained models, then the `joblib.dump()` method is used to save the trained model in that directory. Finally, the list of trained models is returned.

This implementation allows for easy integration of additional classifiers, as they can simply be added to the `self.MODELS` list. Additionally, the trained models can be saved for future use, which can save time and resources when training the system in the future.

It is important to note that the quality of the trained models depends on the quality and size of the training set. Therefore, it is crucial to carefully select and balance the datasets used for training to ensure the best possible results.

3.2.7. **def rescale_risk(self,original_risk:float)**

The function 'rescale_risk()' is designed to adjust the risk value calculated by the AndroidRisk system from a range of 0 to 1 to a range of 0 to 10. The purpose of this function is to make the risk value more easily understandable and interpretable by users. The rescaling is done using a logarithmic function, which helps avoid probabilities that are too close to either 0 or 1. This is because probabilities that are very close to either end of the range can be difficult to interpret and may not reflect the true risk associated with the app.

The logarithmic function used for rescaling is based on the formula '10 * log10(1 + (risk_value * 99))'. The input to the function is the risk value, which ranges from 0 to 1. The function first multiplies the risk value by 99 to scale it up to a range of 0 to 99. It then adds 1 to the result, which ensures that the minimum risk value is 1, avoiding a log value of

0. The logarithmic function is then applied to the result, and the output is multiplied by 10 to scale it up to the desired range of 0 to 10.

By using a logarithmic function to rescale the risk value, the resulting risk score is more evenly distributed across the entire range of 0 to 10. This helps to make the risk score more informative and actionable for users. It also helps to avoid situations where the risk score is skewed towards the extremes of the range, which can be misleading and may not accurately reflect the true risk associated with the app. Overall, the 'rescale_risk()' function is an important component of the AndroidRisk system, as it helps to ensure that the risk scores generated by the system are accurate, informative, and easy to interpret.

3.2.8. **def calculate_risk(self,feature_vector:dict)**

The purpose of the following code is to calculate the risk score of an Android app using its feature vector. The code uses machine learning classifiers to train on a set of malware and benign apps to generate a probability of whether the app under test is malicious or not. The risk value is then rescaled to be between 0 and 10 for better interpretability.

The 'calculate_risk()' function takes a feature vector as input and checks whether the feature vector is empty. If the feature vector is empty, the function returns 'None'. If the feature vector is not empty, the function proceeds to train the classifiers if they have not been trained already. The function then computes the probability of the app under test being malicious for every classifier in the list and obtains the mean probability generated by the classifiers. Finally, the function rescales the risk value using the 'rescale_risk()' function, which uses a logarithmic function to avoid probabilities too close to 0 or 1.

In summary, this code provides a way to calculate the risk score of an Android app based on its feature vector. By training machine learning classifiers on a set of known malicious and benign apps, the code is able to generate a probability of whether an app under test is malicious or not. The risk score is then rescaled to be between 0 and 10, making it

easier to interpret and compare. Overall, this code can be a useful tool for evaluating the security of Android apps.

3.2.9. **def get_training_apks_3_sets(self)**

This Python method is designed to generate three different sets of training APKs and their corresponding labels for evaluating the performance of classifiers. The method starts by collecting all apps from the Malware Collection and Google Play Store collection. Next, it shuffles the apps from the Google Play Store collection and splits them into three subsets with equal sizes. These subsets are then concatenated with the entire set of apps from the Malware Collection, creating three different sets of training APKs.

To label the training APKs, the method creates a target array, which is comprised of the type of the first app in the Malware Collection and the first app in the first subset of Google Play Store apps. This target array is the same for all three sets of training APKs.

Finally, the method returns the three sets of training APKs and their corresponding labels as tuples. This enables the generated training sets to be easily used in classifier training and evaluation.

3.2.10. **def get_training_vectors_3_sets(self)**

This function is designed to generate training feature vectors and their corresponding labels, which are necessary for training classifiers. It begins by calling the 'get_training_apks_3_sets()' function to obtain three sets of training APKs. Then, it creates three dictionaries, one for each set of training vectors, to store the feature vectors for each type of permission along with their corresponding labels (malware or goodware). The permission types to be used are determined by the 'PERMISSION_TYPES' list.

Using a loop, the function iterates over each APK in the training sets. For each APK, it calls the 'get_feature_vector_from_apk()' function to extract the feature vector. Each element of the feature vector

corresponding to each permission type is appended to the corresponding dictionary, and the label for the APK (malware or goodware) is appended to the “target” dictionary.

Finally, the function returns a tuple of tuples, where each tuple contains the feature vectors and their labels for each set of training vectors. These tuples can be used for training the classifiers in the AndroidRisk system.

In summary, this function provides a streamlined way to generate the necessary training data for the classifiers to work accurately. By calling the ‘get_feature_vector_from_apk()’ function to extract the feature vectors and then appending them to the corresponding dictionary, the function creates a comprehensive dataset that can be used to train machine learning models.

3.2.11. **def performance_analysis(self)**

This method evaluates the performance of several classifiers on a given set of training data using 10-fold cross-validation. The training data is divided into three subsets, and the classifiers are trained on each subset separately. The output of the method includes the accuracy, mean, and standard deviation for each classifier and for each subset of the training data.

The classifiers being evaluated include SVM, GaussianNB, MultinomialNB, DecisionTreeClassifier, RandomForestClassifier, LogisticRegression, LogisticRegressionCV, KNeighborsClassifier, and BernoulliNB. All these classifiers are widely used in machine learning and are implemented using the scikit-learn library in Python.

The aim of the classifiers is to classify Android applications as either malware or goodware based on a given set of permissions. The input to the classifiers is a set of feature vectors containing information about the permissions requested by each application.

The code performs binary classification, where the output is either 1 (indicating malware) or 0 (indicating goodware). The performance of

each classifier is evaluated based on its accuracy in predicting the correct output for a given input.

The 10-fold cross-validation used in the method is a widely used technique in machine learning that helps prevent overfitting. It involves dividing the training data into 10 subsets, and then training the classifier on 9 of the subsets and testing it on the remaining subset. This process is repeated 10 times, with each subset serving as the test set once, and the results are averaged to obtain a more accurate estimate of the classifier's performance.

The output of the method is a table showing the accuracy, mean, and standard deviation for each classifier and for each subset of the training data. This information can be used to select the best-performing classifier for a given problem and to assess the generalizability of the classifiers across different subsets of the training data.

Overall, this method provides a useful tool for evaluating the performance of classifiers on a given set of training data and can be used to inform decisions about which classifier to use for a particular problem.

3.2.12. **def calculate_set_accuracy(self)**

This code is part of the AndroidRisk tool and aims to evaluate the accuracy of the classifiers used in the model. It employs a 10-fold cross-validation approach to train and test the classifiers on the training set. The training set APKs and their targets are obtained by calling the ‘get_training_vectors()’ method.

The code then proceeds to train the classifiers and predict the class labels for the test data using the fit and predict methods from the scikit-learn library. The accuracy, mean, and standard deviation are then computed for the malware and goodware scores using the scikit-learn metrics module.

The ‘rescale_risk()’ method, which rescales the risk value to a range of 0 to 1, is assumed to be called within the classifier implementation. The use of cross-validation helps to prevent overfitting and ensures that the model is able to generalize well to new data.

The evaluation of the model’s accuracy is a crucial step in ensuring the effectiveness of the AndroidRisk tool in detecting malware in Android apps. By assessing the model’s performance on the training set, it is possible to identify any issues or limitations with the classifiers used and make necessary adjustments.

In addition, the use of cross-validation provides a more reliable estimate of the model’s accuracy by reducing the variance of the performance metrics. This is because the 10-fold cross-validation approach ensures that all instances in the dataset are used for both training and testing, and the performance metrics are averaged over the 10 folds.

Overall, this code plays an important role in evaluating the effectiveness of the AndroidRisk tool and ensuring that it can accurately detect malware in Android apps.

3.3 Classifiers Used

3.3.1. **K-Nearest Neighbors**

The k-nearest neighbors (KNN or k-NN) algorithm is a supervised learning classifier that is non-parametric and uses proximity to group individual data points for making predictions or classifications. Although it can be used for both regression and classification tasks, it is mainly employed as a classification algorithm, relying on the assumption that similar points are located close to one another.

To categorize data points, the algorithm uses a technique called "majority vote," which involves selecting the label that appears most frequently in the vicinity of the data point. Although this technique is technically known as "plurality voting," the term "majority vote" is more commonly used in the literature. However, it should be noted that

"majority voting" requires a majority of more than 50%, which is appropriate for only two categories. When dealing with multiple categories, such as four, a label can be assigned with a vote greater than 25%, and the term "majority vote" is still used.

To solve regression problems using the k-nearest neighbors (KNN) algorithm, the approach is similar to that used for classification problems, with the key difference being that instead of assigning a class label through majority voting, the algorithm calculates the average of the k-nearest neighbors to make a prediction. While classification involves discrete values, regression deals with continuous ones. To perform classification, a distance measure between data points needs to be defined, with Euclidean distance being the most commonly used measure. The KNN algorithm is also referred to as "lazy learning" because it only stores the training dataset and computes predictions only when necessary. Because of its reliance on memory to store all the training data, it is also known as an instance-based or memory-based learning method.

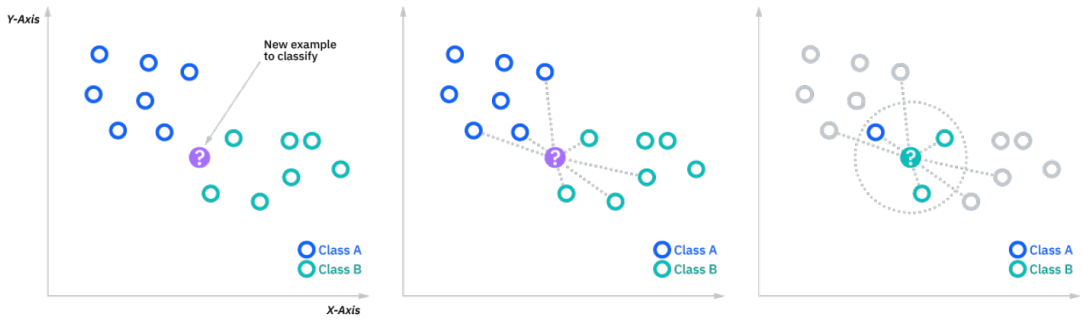


Fig 7: KNN [20].

To solve regression problems using the k-nearest neighbors (KNN) algorithm, the approach is similar to that used for classification problems, with the key difference being that instead of assigning a class label through majority voting, the algorithm calculates the average of the k-nearest neighbors to make a prediction. While classification involves discrete values, regression deals with continuous ones. To

perform classification, a distance measure between data points needs to be defined, with Euclidean distance being the most commonly used measure. The KNN algorithm is also referred to as "lazy learning" because it only stores the training dataset and computes predictions only when necessary. Because of its reliance on memory to store all the training data, it is also known as an instance-based or memory-based learning method.

The KNN algorithm can be used for risk factor analysis of Android applications based on their permissions. In this case, the data points would represent individual Android applications, and the features would be the permissions requested by each app. The algorithm would then use the proximity of apps in terms of their requested permissions to make predictions about their risk factor. For example, the algorithm could be trained on a dataset of known malicious and benign Android apps, with their respective permission sets as features. Then, when a new app is analysed, the algorithm would calculate the distance between the new app's permission set and those of the known apps, and assign a risk factor based on the classes of the k-nearest neighbors. This approach can be useful for identifying potential security risks in Android apps, and can be incorporated into larger security systems for mobile devices.

3.3.2. **Multinomial Naïve Bayes**

The Multinomial Naive Bayes algorithm is a statistical approach that is often used for text classification tasks, such as sentiment analysis or spam detection. However, it can also be used in the context of Android application risk factor analysis based on requested permissions. This algorithm works by utilizing Bayes' theorem to calculate the probability of an app being malware given its requested permissions.

The basic idea behind the Multinomial Naive Bayes algorithm is to model the conditional probability of each feature (i.e., permission) given each class (i.e., malware or benign app). In other words, the algorithm calculates the likelihood of each permission being associated with either malware or benign apps based on a training set of known examples. The

algorithm then combines these probabilities using Bayes' theorem to compute the overall probability of an app being malware given its permission requests.

One of the key assumptions of the Naive Bayes algorithm is that the features are conditionally independent given the class label. This assumption simplifies the computation of probabilities and makes the algorithm computationally efficient and scalable. However, in reality, the features may not be entirely independent, and this can sometimes lead to inaccurate predictions.

To apply the Multinomial Naive Bayes algorithm to Android application risk factor analysis, a training set of known malware and benign apps is needed. The algorithm then calculates the probability of each permission being associated with malware or benign apps, based on the observed frequencies in the training set. These probabilities are then used to calculate the overall probability of an app being malware given its permission requests.

In practice, the Multinomial Naive Bayes algorithm can be a powerful tool for Android application risk factor analysis. It is relatively simple to implement and can provide useful insights into the potential risks associated with a given app. However, it is important to note that the accuracy of the algorithm will depend on the quality of the training data and the assumptions made about the independence of the features. Additionally, the algorithm may not be effective against more sophisticated types of malware that are designed to evade detection by traditional methods.

Overall, the Multinomial Naive Bayes algorithm is a widely used approach for Android application risk factor analysis based on permission requests. By modeling the conditional probabilities of permissions given the class label, the algorithm can provide valuable information about the potential risks associated with a given app. However, it is important to consider the limitations of the algorithm and

to use it in conjunction with other methods for more comprehensive threat detection.

3.3.3. Logistic Regression

The logistic regression algorithm is a popular statistical model that can be used for risk factor analysis in Android applications based on their requested permissions. It is a binary classification method, meaning that it predicts whether an application is "malware" or "benign" based on the permissions it requests.

The logistic regression algorithm uses a training dataset of known malware and benign applications to create a model that can predict the probability of an unknown application being malware based on its permission requests. The model uses the logistic function to transform the input features (i.e., permission requests) into a probability score between 0 and 1. This score represents the likelihood that the application is malware, with a score closer to 1 indicating a higher likelihood of being malware.

The logistic regression algorithm is versatile and can be used with various types of input features. For Android application risk factor analysis, the input features are typically the permissions requested by the application. The algorithm calculates the probabilities of each permission being associated with malware or benign applications, and then combines these probabilities to calculate the overall probability of an app being malware based on its permission requests.

One of the advantages of the logistic regression algorithm is its ability to handle complex and non-linear relationships between the input features and the output variable. This makes it a powerful tool for identifying potentially malicious applications and assisting in making informed decisions about their use. Additionally, the algorithm can be used to identify the specific permissions that are most strongly associated with malware, providing valuable insights into potential security risks.

The threshold value used to classify an application as either "malware" or "benign" can be adjusted to meet the needs of the specific use case. For example, a more conservative threshold value may be used in high-security environments to reduce the risk of false positives (i.e., classifying a benign application as malware). In contrast, a less conservative threshold value may be used in less critical environments to avoid false negatives (i.e., failing to identify a malware application). Overall, the logistic regression algorithm is a powerful and versatile tool for Android application risk factor analysis based on permission requests. Its ability to handle complex relationships between the input features and output variable makes it an effective approach for identifying potentially malicious applications and making informed decisions about their use.

3.4 Selection of Classifiers

Scikit-learn is a library that offers a range of machine learning algorithms designed for classification tasks, including text classification. The library includes 9 supervised classifiers that have the ability to estimate probabilities, which is a useful feature as it allows for the generation of probability values for each classification outcome. This feature can be applied to algorithms like SVM and Decision Trees that typically do not provide probabilities.

To evaluate the performance of these classifiers for Android application risk factor analysis, a study was conducted using three randomly extracted datasets, each containing an equal number of malware and benign samples (6,707 each) and using only the DAP permission set. The study aimed to select the most reliable classifiers based on three empirical rules:

1. A minimum accuracy of 90% was required to eliminate less reliable classifiers.

2. Binary classifiers were avoided by selecting only those with average scores between 4% and 95%.

3. Classifiers with standard deviation less than 5% were excluded, as they had very narrow distributions within the range of possible scores.

The default parameters from the scikit-learn library were used to evaluate the classifiers, utilizing the K-fold cross-validation method with K=10. This technique entails partitioning the dataset into K independent sets, each with an equivalent number of elements. During each iteration, one fold is set aside for testing, while the remaining k-1 folds are utilized as the training set to establish the model. The accuracy of the model is measured by the number of correctly classified samples in the testing set.

The advantage of employing the K-fold cross-validation approach is that it ensures all instances are utilized for both training and testing, minimizing the likelihood of overfitting, where a model may exhibit high performance on the training data but poor performance on the test data. This method allows for a more accurate evaluation of the performance of the classifiers, and the use of default parameters provided by the scikit-learn library ensures a fair comparison between the classifiers.

According to the research results, the Multinomial Naive Bayes, Bernoulli Naive Bayes, Complement Naive Bayes, and Ridge Classifier with default settings were the classifiers that satisfied the three empirical rules. These classifiers showed high accuracy rates ranging from 91% to 93%, indicating that they are reliable for Android application risk factor analysis based on permission requests.

In summary, the scikit-learn library provides a range of machine learning algorithms that can be used for Android application risk factor analysis based on permission requests. The K-fold cross-validation method is a useful technique for evaluating the performance of these classifiers, and the study found that the Multinomial Naive Bayes, Bernoulli Naive Bayes, Complement Naive Bayes, and Ridge Classifier are reliable classifiers for this task. The use of these classifiers can assist in identifying potentially malicious applications and making informed decisions about their use.

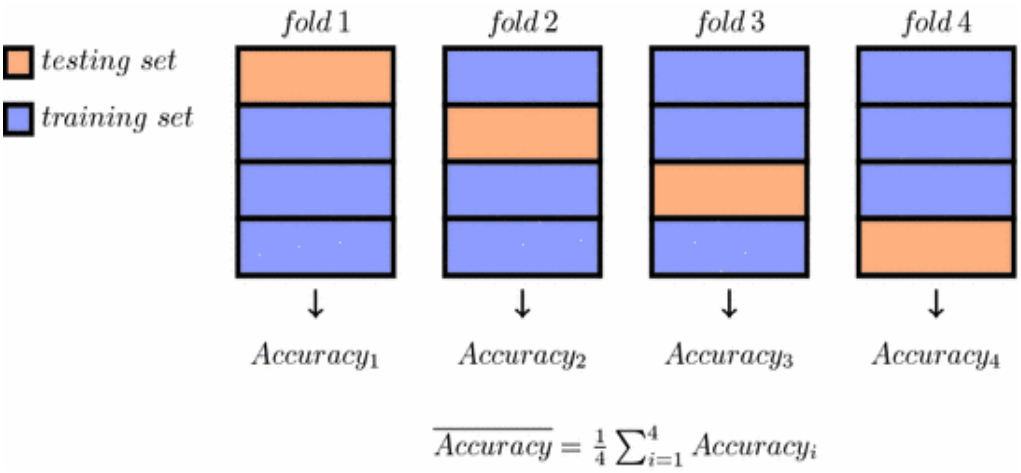


Fig 8: Example for 4-fold cross validation[19].

3.5 Statistical Analysis

To conduct a systematic analysis of the large dataset comprising of over 100,000 Android apps and malware samples, the researchers used a tool called the Permission Checker. This tool was developed to extract relevant information on four different permission sets, namely DAP, EAP, GAP, and UAP. The researchers relied on reverse engineering to retrieve app bytecode and identify permission sets required to execute each method invocation in the bytecode. The Permission Checker then built a set called PS, which contained all the permission sets used in the bytecode. Using PS as a foundation, the tool built the other permission sets in the following manner: EAP comprised of permissions that belonged to both DAP and PS, while GAP contained permissions that belonged to PS but not to DAP. Lastly, UAP contained permissions that belonged to DAP but not to PS.

The researchers identified the disjoint union of all single app permission sets in the dataset as DAP, EAP, GAP, and UAP, which were abbreviated as DAPA, EAPA, GAPA, and UAPA, respectively. This enabled them to perform a comprehensive analysis of each app's permission set and evaluate the effectiveness of probabilistic risk index methods used to calculate RIV. However, while the probabilistic methods had some limitations, the researchers proposed a new approach based on machine learning to address these issues. They created a tool called AndroidRisk, which implemented this methodology

and evaluated it empirically. The researchers also planned to expand the feature set to include suspicious API calls and URLs, which could be identified through static analysis of the bytecode used to construct permission sets.

In summary, the researchers used the Permission Checker tool to analyze a large dataset of Android apps and malware samples, extracting information on four sets of permissions. They then proposed a new approach based on machine learning to improve the limitations of probabilistic risk index methods and developed the AndroidRisk tool to implement this methodology. The researchers also planned to expand the feature set to include additional information, such as suspicious API calls and URLs, to further improve the risk factor analysis of Android applications.

AP Set	MALWARE			APPS		
	MAX AP	AVG AP	Std. dev.	MAX AP	AVG AP	Std. dev.
DAP	87	10.67	5.76	96	5.84	4.39
EAP	15	4.25	3.19	24	3.81	2.40
GAP	9	1.15	1.26	23	2.9	2.11
UAP	84	6.42	4.58	91	2.03	2.78

Table 2: Statistics on APs on the dataset

In Table 2, there are overall statistics provided for the four AP sets. The data suggests that, on average, malware declare more APs than apps (10.67 vs. 5.84), but they use very few of them (4.25). Additionally, malware rarely attempt to use undeclared APs (AVG_{GAP}=1.15), in contrast to apps (AVG_{GAP}=2.9).

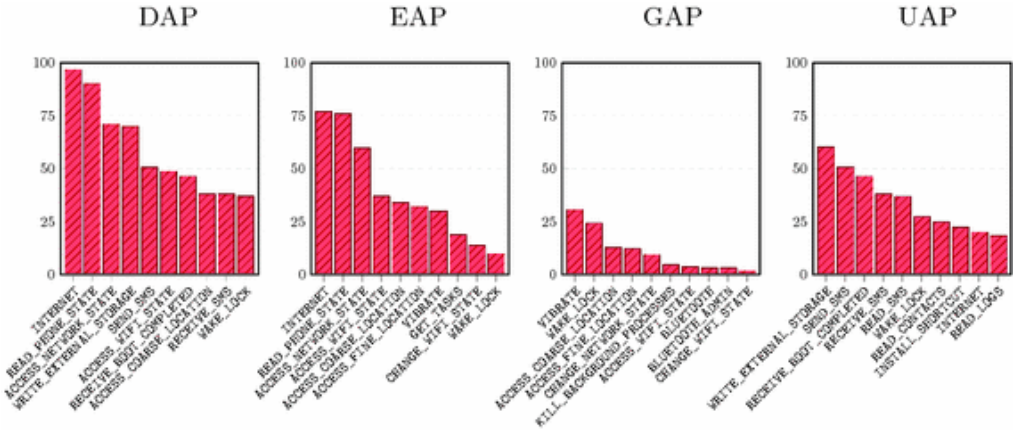


Fig 9: Top 10 APs for malware[19].

Figures 8 and 9 display the top ten access points (APs) used by malware and apps, respectively. The y-axis of each graph indicates the percentage of malware or apps that use the specific AP.

Some APs related to networking are commonly used by both malware and apps, such as INTERNET, ACCESS_NETWORK_STATE, and ACCESS_WIFI_STATE. These APs are necessary for apps that require Internet connectivity, which makes assessing their risk challenging. However, other APs are required more frequently by malware than apps, such as READ_PHONE_STATE, RECEIVE_BOOT_COMPLETED, and READ_CONTACTS, which pose a potential threat. The most significant difference between malware and apps is in relation to SMS APs. As seen in Figure 8, the DAP plot shows that 2 out of 10 APs are related to SMS (SEND_SMS and RECEIVE_SMS), while no SMS-related APs appear in the DAP plot of Figure 9. Although almost half of the malware require SEND_SMS and over 40% require RECEIVE_SMS, they seldom use them, as evidenced by their absence in the corresponding EAP set.

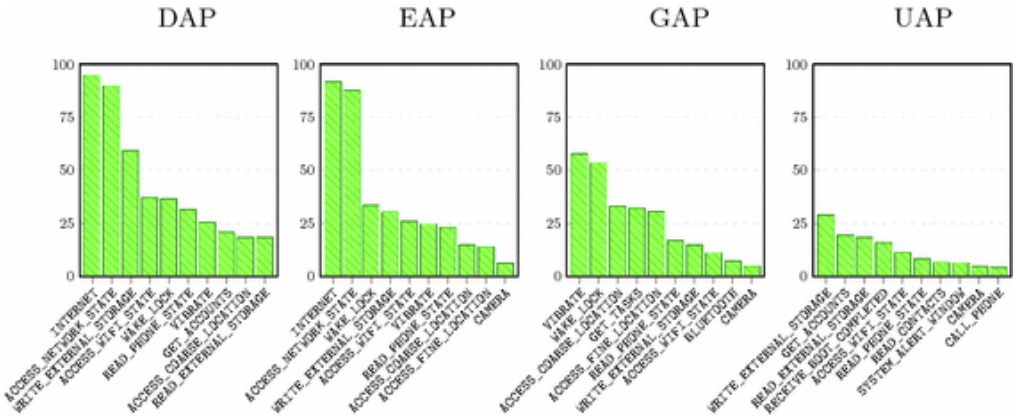


Fig 10: Top 10 APs for apps[19].

CHAPTER - 4

PERFORMANCE ANALYSIS

The study evaluates nine supervised classifiers with probability estimation available in the scikit-learn library, which provide probability values for classification results even for algorithms like SVM and Decision Trees that do not usually provide probabilities. The evaluation process involves training each classifier on a training set of approximately 1342 elements, which includes 671 apps and 671 malware samples. The remaining nine sets are used to test the classifier, with a score of 50% or more considered malware and a score of less than 50% considered non-malware. To determine the accuracy of each classifier, the number of correctly classified elements is divided by the total number of classified elements.

The study applies three empirical rules to select the most reliable classifiers. First, to eliminate less reliable classifiers, a minimum accuracy of 90% is required. Second, only classifiers with average scores between 4% and 95% are selected, avoiding binary classifiers. Third, classifiers with a standard deviation of less than 5% are excluded, as they have very narrow distributions within the range of possible scores.

The evaluation of the classifiers is carried out through the K-fold cross-validation technique, where K is set to 10. This approach involves dividing the dataset into K subsets or "folds", with approximately equal sizes. During each iteration, one fold is used for testing, while the other K-1 folds are utilized for training the model. The benefit of this method is that all samples are used both for training and testing, reducing the risk of overfitting.

The evaluation results are summarized in Table 3, which reports the average value of each metric since all classifiers behaved similarly on all three sets. Support Vector Machines had the highest AVG Accuracy of 94.89%, with an AVG Score of 94.83% for malware and 7.42% for apps. Decision Tree had the highest AVG Score for malware, at 99.68%, and an AVG Accuracy of 95.68%. Random Forest had the highest AVG Score for apps, at 8.87%, and an AVG Accuracy of 96.73%. Gaussian Naïve Bayes had the lowest AVG Accuracy of 84.64%, while Multinomial Naïve Bayes had an AVG Accuracy of 90.69%, and Bernoulli Naïve Bayes had an AVG Accuracy of 89.97%. Logistic Regression had an AVG Accuracy of 94.96%, while Logistic Regression CV had an AVG Accuracy of 94.93%. K-Nearest Neighbors had an AVG Accuracy of 94.29%.

Overall, the evaluation process reveals that the scikit-learn library's classifiers are effective at identifying malware with high accuracy, particularly Support Vector Machines, Decision Tree, and Random Forest classifiers. However, different classifiers may have varying strengths and weaknesses depending on the specific dataset and context. Therefore, it is essential to consider the appropriate classifier based on the dataset's characteristics and the intended application to achieve the best results.

Classifier	AVG Accuracy	Malware		Apps	
		AVG Score	σ	AVG Score	σ
Support Vector Machines	94.89	94.83	7.42	4.73	8.34
Gaussian Naïve Bayes	84.64	99.87	1.82	0.05	1.11
Multinomial Naïve Bayes	90.69	94.88	7.65	4.89	6.29
Bernoulli Naïve bayes	89.97	99.07	4.87	0.69	4.19
Decision Tree	95.68	99.68	3.29	0.73	3.62
Random Forest	96.73	97.31	8.19	4.09	8.87
Logistic Regression	94.96	93.36	8.23	4.85	9.38
Logistic Regression CV	94.93	96.41	8.21	4.71	9.21
K-Nearest Neighbors	94.29	98.69	6.22	4.82	11.34

Table 3: Empirical evaluation of Classifiers in the scikit-learn library

Among the classifiers evaluated, GNB and BNB had low accuracy, while DT, RF, LR-CV, and K-NN had too high average scores for apps. Thus, only KNN, MNB, and LR were found to meet all the requirements for the classification process.

In light of these results, I developed a Python-based tool called AndroidRisk, which incorporates the three chosen classifiers. The tool computes the RIV (Risk Indicator Value) for each app by merging the feature vectors of all four APs (Android Permissions) sets (DAP, EAP, GAP, and UAP) into a single vector. The average score of all three classifiers is used to compute the RIV. Each classifier in AndroidRisk is trained by conducting a 10-fold cross-validation on one of the three sets used for evaluating the classifiers. Additionally, they experimented with using all four APs sets to determine whether it could improve accuracy.

Overall, the use of three classifiers in AndroidRisk provides a promising approach to identifying and classifying malware on Android devices. By computing the RIV for each app, the tool can quickly assess the likelihood of an app being malware and flag it for further investigation if necessary. Additionally, I experiment with using all four APs sets could provide further insights into the potential benefits of incorporating more features in the classification process. However, further research is needed to determine the tool's effectiveness and potential limitations in real-world settings.

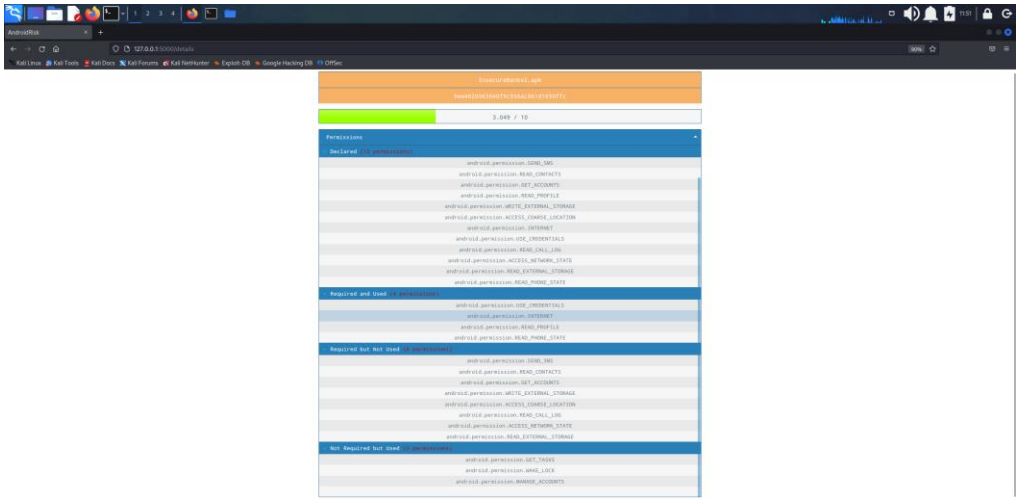


Fig 11: RIV after analysis of android app

CHAPTER - 5

CONCLUSIONS

Our study aimed to assess the practical performance of probabilistic risk index methods for Android applications and to propose an innovative machine learning-based approach to overcome the shortcomings of these techniques. To achieve this goal, we developed a tool named AndroidRisk, which implements our new methodology. We conducted an empirical evaluation of AndroidRisk and found it to be more effective than traditional probabilistic risk index methods.

In our future work, we plan to expand the feature set of AndroidRisk to include suspicious API calls and URLs. This extension will be achieved through static analysis of the bytecode used to construct permission sets. By analyzing the app's code, we can identify API calls and URLs that may be associated with malware and add them to our risk assessment. This approach will enhance the accuracy of our risk assessment and provide users with more comprehensive protection against malicious apps.

Our approach based on machine learning is particularly promising because it can learn from past data and improve over time. As more data becomes available, the accuracy of our risk assessment will continue to improve. Furthermore, machine learning algorithms can also incorporate user feedback and expert knowledge to further enhance the accuracy of our risk assessment.

To sum up, our research has shed light on the shortcomings of conventional probabilistic risk index methods for Android applications and has suggested a novel machine learning-based approach as a solution. Our tool, AndroidRisk, is a step towards providing users with more accurate and comprehensive protection against malicious apps. In the future, we plan to continue developing and improving our approach to better protect users and stay ahead of emerging threats in the Android app ecosystem.

REFERENCES

[1] C. S. Gates et al., "Generating Summary Risk Scores for Mobile Applications," in IEEE Transactions on Dependable and Secure Computing, vol. 11, no. 3, pp. 238-251, May-June 2014, doi: 10.1109/TDSC.2014.2302293.

[2] H. Hao, Z. Li and H. Yu, "An Effective Approach to Measuring and Assessing the Risk of Android Application," 2015 International Symposium on Theoretical Aspects of Software Engineering, Nanjing, China, 2015, pp. 31-38, doi: 10.1109/TASE.2015.16.

[3] M. Li, H. Elahi, and S. Chen. 2020. A Risk Analysis of Android Children's Apps. In Web Information Systems and Applications: 17th International Conference, WISA 2020, Guangzhou, China, September 23–25, 2020, Proceedings. Springer-Verlag, Berlin, Heidelberg, 539–546. https://doi.org/10.1007/978-3-030-60029-7_48

[4] S. Li, T. Tryfonas, G. Russell and P. Andriotis, "Risk Assessment for Mobile Systems Through a Multilayered Hierarchical Bayesian Network," in IEEE Transactions on Cybernetics, vol. 46, no. 8, pp. 1749-1759, Aug. 2016, doi: 10.1109/TCYB.2016.2537649.

[5] Y. Wang, J. Zheng, C. Sun, S. Mukkamala: Quantitative security risk assessment of android permissions and applications. In: Wang, L., Shafiq, B. (eds.) DBSec 2013. LNCS, vol. 7964, pp. 226–241. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39256-6_15

[6] N. Peiravian and X. Zhu, "Machine learning for Android Malware detection using permission and API calls," in Proc. IEEE 25th Int. Conf. Tools Artif. Intell., 2013, pp. 300–305.

[7] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in Proc. 12th Int. Conf. Mobile Syst. Appl. Services, 2014, pp. 204–217.

[8] M. L. Dering and P. McDaniel, "Android Market reconstruction and analysis," in Proc. IEEE Military Commun. Conf., 2014, pp. 300–305.

[9] T. Book, A. Pridgen, and D. S. Wallach, “Longitudinal analysis of Android ad library permissions,” Computing Research Repository CoRR, vol. abs/1303.0857, pp. 1–9, 2013.

[10] J. M. Ruiz, M. Nagappan, B. Adams, T. Berger, S. Dienst, and A. E. Hassan, “Impact of ad libraries on ratings of Android mobile apps,” IEEE Softw., vol. 31, no. 6, pp. 86–92, Nov./Dec. 2014.

[11] Gorla, I. Tavecchia, F. Gross, and A. Zeller, “Checking app behavior against app descriptions,” in Proc. Int. Conf. Softw. Eng., 2014, pp. 292–302.

[12] Bartel, J. Klein, M. Monperrus, and Y. Le Traon, “Static analysis for extracting permission checks of a large scale framework: The challenges and solutions for analyzing Android,” IEEE Trans. Softw. Eng., vol. 40, no. 6, pp. 617–632, Jun. 2014.

[13] T. Watanabe, M. Akiyama, T. Sakai, H. Washizaki, and T. Mori, “Understanding the inconsistencies between text descriptions and the use of privacy-sensitive resources of mobile apps,” in Proc. 11th Symp. Usable Privacy Secur., 2015, pp. 241–255.

[14] Y. Zhou, L. Wu, Z. Wang, and X. Jiang, “Harvesting developer credentials in Android apps,” in Proc. 8th ACM Conf. Secur. Privacy Wireless Mobile Netw., 2015, pp. 23:1–23:12.

[15] H. Wang, J. Hong, and Y. Guo, “Using text mining to infer the purpose of permission use in mobile apps,” in Proc. ACM Int. Joint Conf. Pervasive Ubiquitous Comput., 2015, pp. 1107–1118.

[16] S. Seneviratne, H. Kolamunna, and A. Seneviratne, “A measurement study of tracking in paid mobile applications,” in Proc. 8th ACM Conf. Secur. Privacy Wireless Mobile Netw., 2015, pp. 7:1–7:6.

[17] Platform Architecture, Online available at: https://developer.android.com/guide/platform

[18] M. Arif J, A. Razak MF, S. Awang, T. Mat SR, Ismail NSN, Firdaus A. A static analysis approach for Android permission-based malware detection systems. PLoS One. 2021 Sep 30;16(9):e0257968. doi:

10.1371/journal.pone.0257968. PMID: 34591930; PMCID: PMC8483345.

[19] A. Merlo, G.C. Georgiu, (2017). RiskInDroid: Machine Learning-Based Risk Analysis on Android. In: De Capitani di Vimercati, S., Martinelli, F. (eds) ICT Systems Security and Privacy Protection. SEC 2017. IFIP Advances in Information and Communication Technology, vol 502. Springer, Cham. https://doi.org/10.1007/978-3-319-58469-0_36

[20] K-Nearest Neighbors Algorithm, Online Available at: https://www.ibm.com/topics/knn

S

ORIGINALITY REPORT

5%	5%	5%	0%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	vdocuments.mx Internet Source	5%

Exclude quotes	On	Exclude matches	< 14 words
Exclude bibliography	On		