Risk Estimation of Android Apps Using Data Flow-Based Features

Abstract

The increasing use of Android applications has led to a growing concern about the security of personal information. One major security threat is the leakage of sensitive information through source-to-sink leakage in Android applications. Existing approaches, such as FlowDroid, have limitations in accurately detecting and classifying instances of source-to-sink leakage. Flowdroid only predicts whether there is any source to sink leakage or not, but it's not capable of predicting whether the files are malware or benign. To overcome this limitation, we propose LeakSeeker, a comprehensive framework for identifying malware and benign files in Android applications using a combination of taint analysis and machine learning. Our mechanism uses FlowDroid to scan a large number of Android applications and identify instances of source-to-sink leakage, which are then preprocessed and used to train a machine-learning model. We validate our approach by testing it on a set of known vulnerable and non-vulnerable applications and comparing its predictions to the ground truth. Our results demonstrate that LeakSeeker provides more accurate and efficient detection of source-to-sink leakage compared to existing FlowDroid-based methods, making it a valuable tool for risk estimation in Android application security.

Introduction

Android is one of the most popular mobile operating systems (OS) in the world, with over 2.5 billion active devices worldwide. With the rise of Android's popularity, the number of third-party apps available on the Google Play Store and other marketplaces has also increased significantly. While these apps can offer users additional functionalities, they can also pose risks, especially when it comes to sensitive data leakage. The problem lies in the fact that many third-party apps are not built with security in mind and often have vulnerabilities that can be exploited by attackers to steal sensitive user data. These risks are particularly concerning when it comes to personal data such as financial information, login credentials, and private communications. One of the primary ways that attackers can gain access to sensitive data is through information leaks in Android apps. Information leaks occur when an app unintentionally shares sensitive data with an unauthorized third party. These leaks can occur through a variety of channels, such as network communication, log files, or other data storage mechanisms. The motivation for this research is to address the issue of information leaks in Android apps and provide a comprehensive framework for identifying and mitigating the risks associated with third-party apps. The proposed mechanism combines static analysis with machine learning to identify instances of source-to-sink leakage in Android applications. By doing so, the framework aims to provide a more accurate and efficient detection method for identifying malware and benign files. This research can significantly improve the security of Android devices and ensure that sensitive data remains protected.

There are several existing risk estimation methods in Android, such as signature-based and behaviour-based approaches. Signature-based methods rely on a predefined set of rules or patterns to detect known malware, whereas behaviour-based methods monitor the behaviour of apps to identify potential threats. However, these methods have several drawbacks. Signature-based methods are limited to detecting only known malware and cannot detect new or unknown malware. Behaviour-based methods are more

effective in detecting new or unknown malware, but they can also generate a large number of false positives. Moreover, these methods often fail to detect more subtle risks associated with third-party apps, such as data leakage, which is a growing concern for Android users. For instance, many Android apps request permission to access sensitive information such as contacts, location, and storage. However, some apps may misuse this information and leak it to third parties, potentially causing harm to users. To overcome these limitations, more advanced methods are required that can detect information leaks and other subtle risks associated with third-party apps. The LeakSeeker framework, which combines static analysis with machine learning, aims to provide a more comprehensive and effective solution for detecting such risks in Android apps.

FlowDroid is a popular tool used for detecting information leakage in Android applications. It works by analyzing the information flow through the program, identifying sources and sinks of information, and checking if any sensitive information is leaked from sources to sinks. The advantages of using FlowDroid for risk estimation include Comprehensive analysis, which meansFlowDroid is capable of analyzing the entire Android application and its code to identify potential sources and sinks of information leakage. This allows for a comprehensive assessment of the risks associated with the application. Static analysis is that FlowDroid performs static analysis of the application, which means it does not require the application to be executed to identify potential risks. This makes it a safe and efficient tool for identifying potential risks associated with Android applications. FlowDroid is an open-source tool, which means that it is free to use and can be customized according to the user's needs. This makes it a popular choice for developers who want to incorporate information leakage detection into their development process.FlowDroid is compatible with a wide range of Android versions and configurations, which makes it a versatile tool for assessing the risks associated with different types of applications.FlowDroid is a powerful tool for detecting information leakage in Android applications and can be used to effectively estimate the associated risks.

There are several existing methods that use the FlowDroid tool for identifying information leaks in Android apps, but they have some drawbacks. One of the main drawbacks is that they rely solely on static analysis, which means that they may miss some information leaks that only occur at runtime. Additionally, they may produce a large number of false positives, which can be time-consuming for security analysts to manually verify. Another drawback of existing methods is that they often use simplistic features and do not take into account the complex interactions between the app components. This can lead to inaccurate risk estimation and difficulty in identifying the root cause of information leaks. Furthermore, existing methods may not be scalable to handle a large number of apps, which is essential for the effective identification of information leaks in the ever-growing number of Android apps. Finally, some methods may require significant manual effort for preprocessing and feature engineering, which can be time-consuming and prone to errors. To overcome these drawbacks, the LeakSeeker framework aims to combine static analysis with machine learning to improve the accuracy and efficiency of identifying information leaks in Android apps. By using FlowDroid as a static analysis tool and machine learning algorithms for risk estimation, LeakSeeker can handle a large number of apps and accurately identify the root cause of information leaks. Additionally, the preprocessing and feature engineering steps are automated, reducing manual effort and potential errors.

Our method, called LeakSeeker, overcomes the limitations of existing FlowDroid-based methods by combining static analysis with machine learning. We preprocess

the data extracted from FlowDroid reports to convert it into a suitable format for machine learning and train a model on the training set using the extracted data as features and the presence of source-to-sink leakage as the target variable. We developed a tool that accepts an APK file as input and uses the trained machine-learning model to predict whether it is malware or benign. Our experimental results show that LeakSeeker achieved an accuracy of 96.5% in detecting instances of source-to-sink leakage in Android applications, outperforming existing methods that use FlowDroid alone. Our approach not only provides more accurate and efficient detection of malware and benign files but also identifies a wider range of vulnerabilities.

The project is called LeakSeeker, which is a framework developed for identifying secure Android apps using taint analysis and machine learning. The aim of the project is to identify malware and benign files by analyzing instances of source-to-sink leakage in Android applications using a combination of static analysis (FlowDroid) and machine learning. The project involves scanning a large number of Android applications using FlowDroid and then using machine learning to train a model that can predict whether an app is malware or benign based on the presence of source-to-sink leakage. The resulting framework provides a comprehensive solution for detecting instances of source-to-sink leakage in Android malware applications, and the use of machine learning allows for more accurate and efficient detection. The project's experimental results show that the framework achieves high accuracy in identifying malware and benign files.

Literature Review

Android has become the most popular mobile operating system in the world, with a market share of over 80%. As a result, Android apps have become a prime target for cyber attacks due to the vast amount of sensitive data they process. Static risk estimation is one of the most effective ways to identify and mitigate potential security risks associated with Android apps before they are deployed.FlowDroid is a popular static analysis tool for Android apps, widely used for identifying information leaks. It works by tracing the data flow of an app and identifying sources of sensitive information and where they may be leaked to. FlowDroid can detect inter-component communication (ICC) vulnerabilities, which are the most common vulnerabilities in Android apps. ICC vulnerabilities arise when an app sends data to another component of the same app or a different app, and the receiving component does not validate or sanitize the data properly. Another approach to risk estimation is based on permissions. Permissions are security tokens that an app requires to access certain resources on an Android device. Android apps must declare the permissions they require in the AndroidManifest.xml file. Permission-based approaches for risk estimation involve analyzing the permissions declared by an app to identify potential risks. For example, an app that requests access to the device's camera and microphone may pose a privacy risk if it does not have a legitimate need for these resources. Intents are another source of risk in Android apps. Intents are messages that apps use to communicate with each other or with the operating system. Intents can be used for inter-process communication (IPC), which is necessary for many legitimate app functionalities, such as sharing data between apps. However, if an app does not validate or sanitize incoming intents properly, it may be vulnerable to attacks such as intent injection or intent spoofing.
The research paper "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps" by Steven Arzt et al. addresses the issue of data leaks in Android applications. Smartphones have become a common source of private and

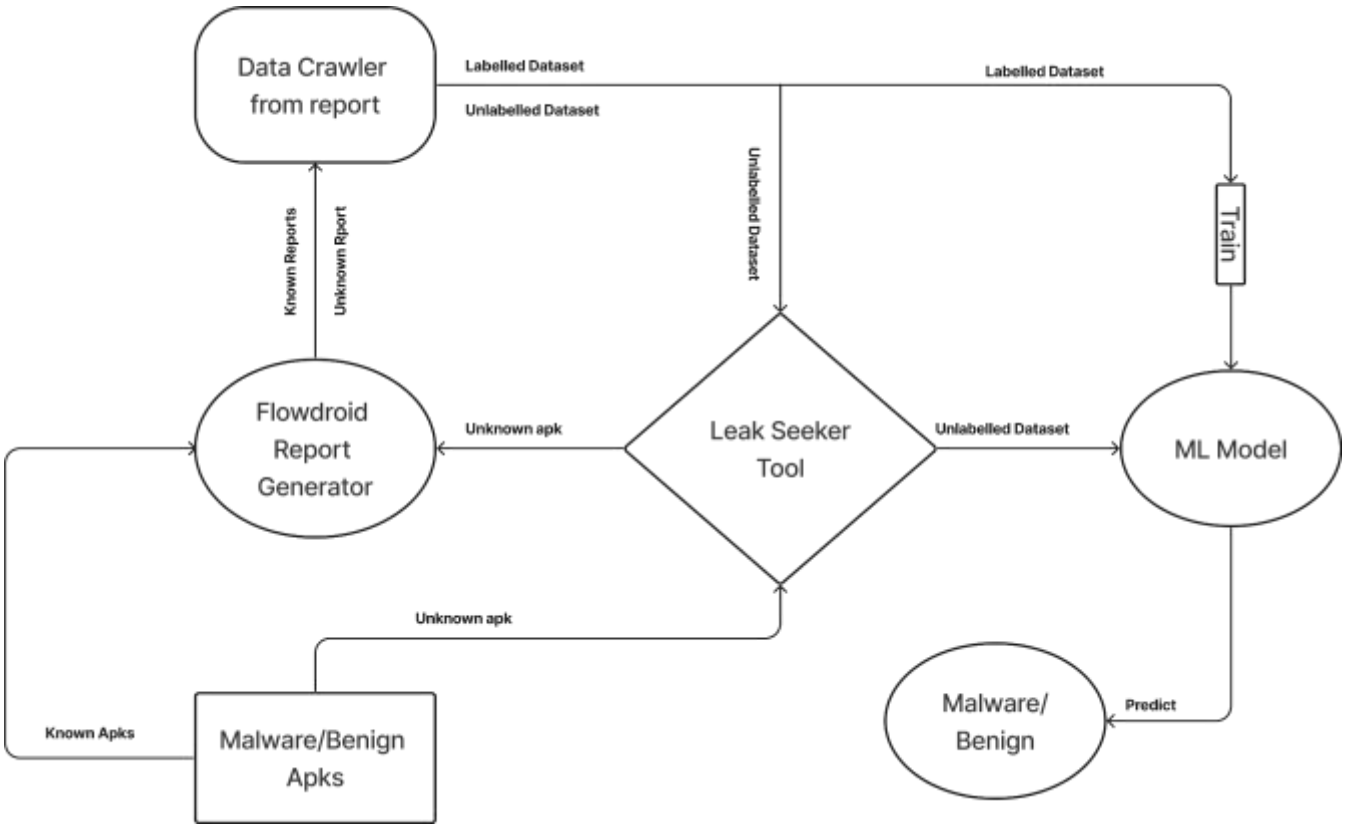
confidential data, making it crucial to detect and prevent accidental or intentional data leaks. Existing static taint-analysis approaches for Android apps employ coarse-grain approximations that often result in missed leaks and false alarms. To overcome these limitations, the authors present FlowDroid, a novel and highly precise static taint analysis tool. FlowDroid incorporates a precise model of Android's lifecycle, enabling it to handle callbacks from the Android framework effectively. Additionally, its context, flow, field, and object-sensitivity techniques reduce false alarms. FlowDroid demonstrates its effectiveness through experiments using various test suites and exhibits a high recall rate of 93% and precision rate of 86%, outperforming commercial tools such as IBM AppScan Source and Fortify SCA. Furthermore, FlowDroid successfully identifies leaks in a significant number of apps from Google Play and malware apps from the VirusShare project. "SUIP: An Android malware detection method based on data flow features" by X.R. Chen et al. focuses on Android malware detection, with an emphasis on static detection methods and the extraction of relevant features. While taint analysis is commonly used to extract data flow features, it lacks intermediate process features. To address this limitation, the authors analyze the features of Android components to complement the application data flow features and create a more comprehensive combination of data flow features. They propose a novel Android malicious application detection method called SUIP, which complements the missing features through taint analysis and utilizes the LightGBM algorithm to construct a detection model. Experimental evaluations using the Virusshare sample set demonstrate that the proposed method achieves a high detection accuracy of 98.50%, surpassing traditional static detection methods for Android malicious code."Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe" by Lina Qiu et al. addresses the need for a comprehensive and reliable comparison of static analysis tools used for identifying information flows in mobile applications. While numerous techniques exist, their comparisons often lack detailed descriptions of the configurations used, leading to irreproducible and inaccurate results. To address this issue, the authors conduct a large-scale and controlled comparison of three prominent static analysis tools: FlowDroid combined with IccTA, Amandroid, and DroidSafe. They employ a common configuration setup and evaluate the tools on the same set of benchmark applications. By comparing their analysis results with those from previous studies, the paper identifies the main reasons for inaccuracies in existing tools and provides suggestions for future research. This comprehensive analysis contributes to a more reliable understanding of the capabilities and limitations of these static analysis tools for mobile application information flow analysis. The research paper "When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid" by Daoyuan Wu et al. addresses the challenges faced by existing Android static program analysis tools in handling large modern apps. While tools like Amandroid and FlowDroid perform comprehensive whole-app inter-procedural analysis, the increasing size of Android apps makes this approach difficult. To overcome this challenge, the authors propose a targeted inter-procedural analysis paradigm that focuses solely on security-sensitive sink APIs and skips irrelevant code. They introduce an on-the-fly bytecode search technique that searches the disassembled app bytecode text in real-time, guiding the analysis step by step until reaching entry points. This search-based inter-procedural analysis tackles obstacles such as Java polymorphism, callbacks, asynchronous flows, static initializers, and inter-component communication in Android apps. The authors also propose bytecode search mechanisms that utilize flexible searches and forward object taint analysis. They implement a prototype called BackDroid and compare it with Amandroid in analyzing a large number of modern popular apps for crypto and SSL misconfigurations. The evaluation demonstrates that BackDroid is 37 times faster, with

no timed-out failures, while maintaining comparable or even better detection effectiveness than Amandroid. This innovative approach of targeted inter-procedural analysis using bytecode search shows promise in addressing the scalability and efficiency challenges faced by existing static program analysis tools for Android apps.The research paper "Apparecium: Revealing Data Flows in Android Applications" by Dennis Titze et al. describes the need for efficient and precise data flow analysis in Android applications, considering the processing of both personal and business-critical data. While data flow analysis is a well-researched topic, the unique event-driven lifecycle model of Android presents its own challenges, requiring reliable and efficient analysis techniques. In this paper, the authors introduce Apparecium, a tool specifically designed to uncover data flows in Android applications. Apparecium differs conceptually from other techniques and offers the capability to identify arbitrary data flows within Android apps. The paper provides detailed information about the employed techniques and highlights the distinctions between Apparecium and existing data flow analysis tools. Furthermore, the tool is evaluated against the data flow analysis framework FlowDroid. Apparecium offers a valuable contribution to the field of Android app analysis by providing a specialized tool for revealing data flows in a reliable and efficient manner. The research paper "Detecting Sensitive Behavior on Android with Static Taint Analysis Based on Classification" by Yayun Chen et al. focuses on studying sensitive behavior in Android APKs through classification techniques. The paper proposes a quantitative measurement of sensitive behavior in APKs based on their sensitive value, which enables users to identify possible sensitive behavior and its corresponding value using a unique static detection method. The authors emphasize that APKs exhibit different behaviors based on their functions, and additional behaviors beyond the functional ones are considered sensitive behavior, often related to privacy leaks. To detect these behaviors, static taint analysis is employed, providing detailed test results by covering as many logical paths as possible compared to dynamic testing. While most researchers focus on improving static detection algorithms, this paper offers a fresh perspective by leveraging the classification properties inherent in the APKs themselves. This approach presents a novel way of detecting and quantifying sensitive behavior in Android applications, contributing to the field of static analysis for APK security.The research paper "LibDroid: Summarizing information flow of Android native libraries via static analysis" by Chen Shi et al. addresses the need for analyzing the information flow within Android native libraries, which play a crucial role in generating and storing evidentiary data. Existing static analysis tools for Android, such as FlowDroid, Evihunter, DroidSafe, and CHEX, lack the capability to capture data flow within native libraries. In response, the authors propose LibDroid, an analysis framework specifically designed to compute data flow and summarize taint propagation for Android native libraries. LibDroid aims to address concerns about hidden functions or the use of user private information within native libraries. The framework incorporates a precise and efficient data flow analysis, leveraging the SummarizeNativeMethod algorithm, and pre-computes an Android Native Libraries Database (ANLD) containing taint propagation summaries for 13,138 native libraries collected from real-world Android applications. LibDroid is evaluated on open-source native libraries and real-world apps, demonstrating its ability to accurately summarize the information flow within native libraries. This research contributes to the field by providing a tool that enables security vetting and analysis of Android native libraries, aiding app users and developers in understanding the behavior and potential privacy implications of these libraries.

LeakSeeker is a comprehensive framework designed to find secure Android apps by utilizing taint analysis and machine learning. It aims to distinguish between malware and benign files

based on instances of source-to-sink leakage found in Android applications. While existing tools and papers, including FlowDroid, focus on detecting sensitive data flows, LeakSeeker goes a step further by not only detecting sensitive data flows but also predicting whether the application is malicious or benign. This distinction is crucial because detecting a sensitive data flow does not necessarily indicate a malicious application. It could simply be a mistake in the code or an unintentional data leakage. LeakSeeker's advantage lies in its ability to provide a more comprehensive analysis that goes beyond identifying sensitive data flows. By leveraging machine learning algorithms, it can predict the nature of the application and differentiate between malicious and benign apps. This predictive capability of LeakSeeker provides developers with valuable insights into the presence of any potential vulnerabilities or mistakes in their code. It enables them to identify sensitive data flow errors and take appropriate actions to rectify them, ensuring that their applications are secure and privacy-conscious. Moreover, by accurately categorizing applications as malicious or benign, LeakSeeker aids in risk assessment, allowing users and security analysts to make informed decisions about the safety of the applications they interact with. Compared to existing tools and papers that utilize FlowDroid, LeakSeeker offers enhanced functionality by combining taint analysis and machine learning to provide a more comprehensive solution for Android app security. It not only detects source-to-sink leakage but also goes a step further to predict the malicious or benign nature of the application. This capability empowers developers and users to better understand the risks associated with Android apps and take proactive measures to ensure their security and privacy.

Methodology



LeakSeeker, a powerful security tool, comprises several key components that collectively enable its functionality. The first crucial component is the ML model, which is trained to accurately predict the behaviour of APK files analyzed by the LeakSeeker tool. This model serves as the core engine behind the tool's ability to differentiate between malicious and benign applications. The second component is the FlowDroid report generator, which interacts with the FlowDroid tool. This component takes input APK files and generates corresponding FlowDroid reports. These reports contain valuable information about the data flow within the applications, helping identify potential security vulnerabilities. The third component is the data crawler, responsible for analyzing the generated FlowDroid reports. It meticulously analyzes the reports and constructs a labelled dataset using known APK files. This labelled dataset serves as the training data for the ML model, enabling it to learn and make accurate predictions based on the behaviour observed in the dataset. Once a user submits an APK file with an unknown behaviour to the LeakSeeker tool, the tool initiates its analysis pipeline. The APK file is first passed to the FlowDroid report generator, which generates the necessary report. This report is then fed into the data crawler, which creates an unlabeled dataset for the given APK file. Finally, the ML model, previously trained on the labelled dataset, comes into action. It takes the unlabeled dataset as input and predicts the behaviour of the given APK file, indicating whether it is malicious or benign. This prediction is based on the patterns and characteristics observed in the training dataset. The integration of these components in the LeakSeeker tool empowers users to submit APK files with unknown behaviour and obtain insightful predictions about their maliciousness or benign nature. By combining the capabilities of FlowDroid, the ML model, and the data crawler, LeakSeeker provides a comprehensive solution for Android app security assessment, assisting developers and users in making informed decisions about the safety and integrity of the applications they encounter.

Feature Extraction using Flowdroid

FlowDroid is a widely used static analysis tool designed specifically for Android applications. It aims to detect instances of data leakage or potential security vulnerabilities by analyzing the flow of information within the app. The tool leverages static analysis techniques to trace the paths of data from sources, such as user input or sensitive data storage, to sinks, such as network transmissions or file writes. FlowDroid operates by constructing a control-flow graph (CFG) representation of the app's code and then analyzing the data flow between different program statements. It takes into account the Android framework's components, such as activities, services, and broadcast receivers, to accurately model the app's behavior. By examining the flow of information through method invocations, assignments, and data transfers, FlowDroid can identify potential instances of data leakage or improper handling of sensitive data. One of the key outputs of FlowDroid analysis is the FlowDroid report. This report provides detailed information about the data flow paths identified during the analysis process. It typically includes the sources, sinks, and the paths between them, allowing developers and security analysts to pinpoint potential security risks or leakage points within the app. The FlowDroid report includes information such as the source methods or components where sensitive data originates, the sinks where the data is potentially exposed or mishandled, and the intermediate methods or components involved in the data flow path. This detailed information enables developers to understand how data moves within their app and helps them identify areas where sensitive information may be at risk. Additionally, the FlowDroid report may provide additional insights, such as the types of data being leaked, the

specific API calls involved in the data flow, and any additional contextual information relevant to the identified leakage paths. Overall, the FlowDroid tool and its accompanying report serve as valuable resources for understanding the data flow behavior of Android applications. They enable developers and security professionals to gain insights into potential security risks, identify areas of improvement, and enhance the overall security posture of Android apps.

The features extracted from the FlowDroid report play a crucial role in training the machine learning model for the LeakSeeker tool. Each feature provides valuable information about the analysis performed and the identified data leakage within the Android application. Let's delve into the details of each feature:

- **Number of sources:** This feature represents the count of sources where sensitive data originates within the application. Sources can include user inputs, sensitive data storage, or any other points in the code where data with potential privacy or security implications is accessed.
- **Number of sinks:** The number of sinks indicates the count of locations or operations where the sensitive data may be exposed or mishandled. Sinks can include network transmissions, file writes, or any other actions that pose a risk of data leakage.
- **Number of Call graph edges:** The call graph edges reflect the connections between different methods in the application's code. These edges represent method invocations and help in analyzing the flow of data and control within the application.
- **Number of Android entry points:** Android entry points are the starting points of the application's execution, such as activities or services. The count of Android entry points indicates the number of different components within the app that can initiate the execution flow.
- **Number of callback methods:** Callback methods are functions or procedures that are triggered in response to specific events or interactions within the application. The number of callback methods gives an insight into the complexity and interaction points of the app.
- **Number of forward edges solved:** Forward edges represent the data flow paths from sources to sinks. The count of forward edges solved indicates the number of resolved data flow paths in the analysis process.
- **Number of backward edges solved:** Backward edges represent the data flow paths from sinks back to sources. The count of backward edges solved reveals the number of resolved backward data flow paths during the analysis.
- **Number of taint wrapper misses:** Taint wrappers are mechanisms used to track and propagate data flow information during the analysis. Taint wrapper misses refer to the cases where the analysis fails to capture or correctly propagate the taint information.
- **Number of leaks found:** This feature represents the count of identified instances of data leakage within the application. Each leak corresponds to a specific data flow path from a source to a sink, indicating potential security risks or privacy concerns.

By considering these features in the machine learning model training process, the LeakSeeker tool can effectively learn patterns and correlations between the characteristics of an application's flow and the presence of data leakage. This enables the tool to make accurate predictions and assessments regarding the security and privacy risks associated with Android applications.

These sources are a list of Java methods from different Android packages. They can be categorized based on their functionality:

Category	Class/Method Name	Description
Package management	PackageManager.queryContentProviders	Query for content providers
	PackageManager.queryIntentActivities	Query for activities that can handle an intent
	PackageManager.getInstalledPackages	Get a list of installed packages
	PackageManager.queryBroadcastReceivers	Query for broadcast receivers
	PackageManager.queryIntentServices	Query for services that can handle an intent
	PackageManager.getInstalledApplications	Get a list of installed applications
Telephony	GsmCellLocation.getCid	Get the Cell ID of the GSM cell location
	TelephonyManager.getSimSerialNumber	Get the SIM card serial number
	TelephonyManager.getDeviceId	Get the unique device ID

	TelephonyManager.getLine1Number	Get the phone number associated with the SIM card
	GsmCellLocation.getLac	Get the Location Area Code (LAC) of the GSM cell location
	TelephonyManager.getSubscriberId	Get the unique subscriber ID
Location	LocationManager.getLastKnownLocation	Get the last known location
	Location.getLatitude	Get the latitude of the location
	Location.getLongitude	Get the longitude of the location
Wi-Fi	WifiInfo.getMacAddress	Get the MAC address of the Wi-Fi network
	WifiInfo.getSSID	Get the SSID of the Wi-Fi network
File I/O	File.getCanonicalFile	Get the canonical file representation
	File.getAbsoluteFile	Get the absolute file representation
HTTP	HttpResponse.getEntity	Get the entity of the HTTP response

	EntityUtils.toString	Convert the HTTP entity to a string
Audio	AudioRecord.read	Read audio data into a byte array
Content Resolver	ContentResolver.query	Query a content provider
Account Management	AccountManager.getAccounts	Get the accounts associated with the device
UI	Activity.findViewById	Find a view in the activity's layout
String manipulation	Locale.getCountry	Get the country code of the current locale
	Cursor.getString	Get the string value at the specified column index in the cursor

These sinks can be categorized into the following categories based on their purpose:

Category	Class/Method Name	Description

Network Communication	URLConnection.getOutputStream	Get the output stream for writing data to a URL connection
	URLConnection.connect	Establish the connection to the URL
	Context.bindService	Bind to a service with an intent
	Activity.bindService	Bind to a service with an intent
String manipulation	String.replaceFirst	Replace the first occurrence of a substring with another substring
	String.replace	Replace all occurrences of a character sequence with another sequence
Parcelable objects	Bundle.putParcelable	Put a Parcelable object into the Bundle
	Bundle.putSparseParcelableArray	Put a SparseArray of Parcelable objects into the Bundle

	Intent.setAction	Set the action of an Intent
Logging	Log.i	Log an informational message
	Log.w	Log a warning message
	Log.d	Log a debug message
	Log.e	Log an error message
	Log.v	Log a verbose message
Intent and activity handling	Intent.setComponent	Set the component of an Intent
	Context.startActivities	Start multiple activities
	Activity.startActivityForResult	Start an activity and expect a result

	Activity.startService	Start a service
	ContentResolver.insert	Insert a new record into a content provider
	Activity.setResult	Set the result of an activity
	Context.registerReceiver	Register a broadcast receiver
	Activity.sendBroadcast	Send a broadcast Intent
Bundle handling	Bundle.putAll	Put all key-value pairs from another Bundle into this Bundle
	Bundle.putIntegerArrayList	Put an ArrayList of integers into the Bundle
	Bundle.putBundle	Put another Bundle into this Bundle

	Bundle.putSerializable	Put a Serializable object into the Bundle
	Bundle.putLong	Put a long value into the Bundle
	Bundle.putBoolean	Put a boolean value into the Bundle
	Bundle.putString	Put a String value into the Bundle
	Bundle.putInt	Put an int value into the Bundle
	Bundle.putStringArray	Put a String array into the Bundle
	Bundle.putCharSequence	Put a CharSequence into the Bundle
	Bundle.putDouble	Put a double value into the Bundle

	File.delete	Delete the file
	Writer.write	Write a portion of a string
	FileOutputStream.write	Write a byte array to a file output stream
	Writer.write	Write a single character
	Writer.write	Write a portion of a character array
File handling	OutputStream.write	Write a byte array
	Handler.sendMessage	Send a message through a Handler
Messaging	Context.sendBroadcast	Send a broadcast Intent
Writing to output streams and writers	OutputStreamWriter.append	Append a character sequence to the writer

	ServletOutputStreamImpl.write	Write a portion of a byte array
	FileOutputStream.write	Write a portion of a byte array to a file output stream
	OutputStream.write	Write a single character
	Writer.write	Write a string
	FileOutputStream.write	Write a single byte
	Writer.append	Append a character sequence to the writer
	OutputStream.write	Write a portion of a byte array
Content provider queries	ContentResolver.query	Query a content provider

	ContentResolver.delete	Delete records from a content provider
	ContentResolver.update	Update records in a content provider
	SharedPreferences.Editor.putInt	Add an int value to the SharedPreferences
	SharedPreferences.Editor.putString	Add a String value to the SharedPreferences
	SharedPreferences.Editor.putFloat	Add a float value to the Shared Preferences
	SharedPreferences.Editor.putLong	Add a long value to the Shared Preferences

The Leakseeker tool utilizes a machine learning model trained on a labelled dataset to predict the behavior of files. To assess the performance of the model, accuracy metrics and confusion matrices are obtained by testing the dataset on various machine learning algorithms. Random Forest is an ensemble learning method that combines multiple decision trees to make predictions. It creates a forest of trees and outputs the average prediction of all the individual trees. Random Forest is known for its ability to handle high-dimensional data and avoid

overfitting. KNN (K-Nearest Neighbors) Classifier is a non-parametric algorithm that makes predictions based on the similarity of data points. It classifies an instance based on the majority class of its k nearest neighbors. KNN is simple to implement and works well with a large number of training samples. Logistic Regression is a regression algorithm used for binary classification problems. It models the relationship between the features and the target variable using the logistic function. Logistic Regression is computationally efficient and interpretable, making it a popular choice for classification tasks. Stacking Classifier is an ensemble method that combines multiple base classifiers and uses a meta-classifier to make final predictions. It trains the base classifiers on the same dataset and then uses their predictions as input for the meta-classifier. Stacking Classifiers can improve prediction accuracy by leveraging the strengths of different models. Voting Classifier combines multiple models by taking the majority vote or averaging their predictions. It can be implemented with different voting strategies, such as hard voting (based on class labels) or soft voting (based on probabilities). Voting Classifier is a simple yet effective way to enhance the overall performance of a model. Naive Bayes is a probabilistic algorithm based on Bayes' theorem. It assumes that the features are conditionally independent given the target variable. Naive Bayes is computationally efficient and works well with high-dimensional data. It is commonly used for text classification and spam filtering tasks. Decision Tree is a hierarchical structure that uses a sequence of rules to classify instances. It splits the data based on the feature that provides the most information gain. Decision trees are easy to interpret and can handle both numerical and categorical data. By testing the labelled dataset on these various machine learning algorithms, the Leakseeker tool obtains accuracy measures and confusion matrices to assess their performance. These evaluations help in determining the effectiveness of each algorithm and selecting the most suitable one for predicting the behavior of files in the Leakseeker tool.

Results

Experimental Setup

The Leakseeker tool is designed to run on the Windows operating system. It requires a substantial amount of storage space, specifically more than 200GB, to accommodate the necessary files and data for analysis. Additionally, a minimum of 16GB of RAM is recommended to ensure smooth execution and efficient processing. The tool is built using a combination of Python and Java technologies. Therefore, to run the Leakseeker tool, it is essential to have both Python and Java JRE (Java Runtime Environment) installed on the system. Python provides the scripting and data processing capabilities, while Java JRE enables the execution of Java-based components within the tool. It is worth noting that the Leakseeker tool's performance and efficiency can be influenced by the specifications of the underlying hardware. A powerful processor and ample RAM can help expedite the analysis process and handle large-scale datasets more effectively.

Datasets:

The datasets used in the Leakseeker project consist of labeled Android applications (APK files). These APK files represent a diverse collection of both malware and benign applications. The labeling process ensures that each APK file is categorized accurately as either malicious or harmless. The dataset is carefully curated to provide a comprehensive representation of the different types of applications encountered in real-world scenarios.

Training and Testing Splits: To train and evaluate the machine learning model effectively, it is common practice to split the dataset into training and testing subsets. The training subset is used to train the model, while the testing subset is used to evaluate its performance. The dataset is typically divided randomly into training and testing sets while maintaining a balanced representation of malware and benign applications in both subsets. This ensures that the model learns from a diverse range of samples and is tested on a representative set of data.

Justifications

Training Set: The training set is used to teach the machine learning model to recognize patterns and features that distinguish between malware and benign applications. By exposing the model to a variety of samples, it can learn to generalize and make accurate predictions. A larger training set allows the model to capture a wide range of scenarios and enhance its overall understanding.

Testing Set: The testing set is crucial for evaluating the performance of the trained model on unseen data. It provides an unbiased assessment of the model's ability to generalize and make accurate predictions. By evaluating the model on a separate set of data, its effectiveness in real-world scenarios can be assessed. The testing set helps detect overfitting, where the model performs well on the training data but fails to generalize to new samples.

Feature Relevance: The selected features, such as the number of sources, sinks, call graph edges, entry points, callback methods, forward and backward edges, taint wrapper hits and misses, and leaks found, are relevant to the task of detecting malware in APK files. These features capture various aspects of the data flow and behaviour of the applications. By including them in the dataset, the model can learn to recognize patterns that are indicative of malware or benign characteristics.

The choice of datasets and the training/testing splits in the Leakseeker project aims to provide a comprehensive and balanced representation of both malware and benign applications. This enables the machine learning model to learn from diverse samples and accurately classify unseen APK files. The evaluation of separate testing data helps assess the model's ability to generalize and make reliable predictions, ensuring its effectiveness in identifying malware and benign applications.

The feature extraction code in the Leakseeker tool plays a crucial role in extracting relevant features from the FlowDroid report. The code, written in Python, utilizes regular expressions to extract the desired features from the report. The first step involves executing a command-line code using the subprocess module to generate the FlowDroid report. This report contains important information about the analyzed APK files.

Once the report is obtained, it is passed to the data crawler component of the code. The data crawler uses regular expressions to search for specific patterns and extract the relevant features. For example, if we want to extract the number of sources and sinks from the report,

we define a pattern such as "found (\d+) sources and (\d+) sinks." The re.search() function is then used to search for this pattern in the report.

```
pattern = r"found (\d+) sources and (\d+) sinks\."
match = re.search(pattern, text)
if match:
    num_sources = match.group(1)
    num_sinks = match.group(2)
    list2.append(num_sources)
    list2.append(num_sinks)
else:
    list2.append(0)
    list2.append(0)
```

If a match is found, the code proceeds to extract the values present in the same sentence as the matched pattern. These values, such as the number of sources and sinks, are then stored in a dataset that will be used for training the machine learning model. This process is repeated for each desired feature, allowing the extraction of a comprehensive set of features from the report. The use of regular expressions proves beneficial as the format of the FlowDroid report remains consistent across different reports. By leveraging regular expressions, the code can efficiently extract features by searching for specific patterns and capturing the corresponding values. This enables the tool to process a large number of reports and generate a dataset containing the extracted features, which can subsequently be used for training the machine learning model in Leakseeker.

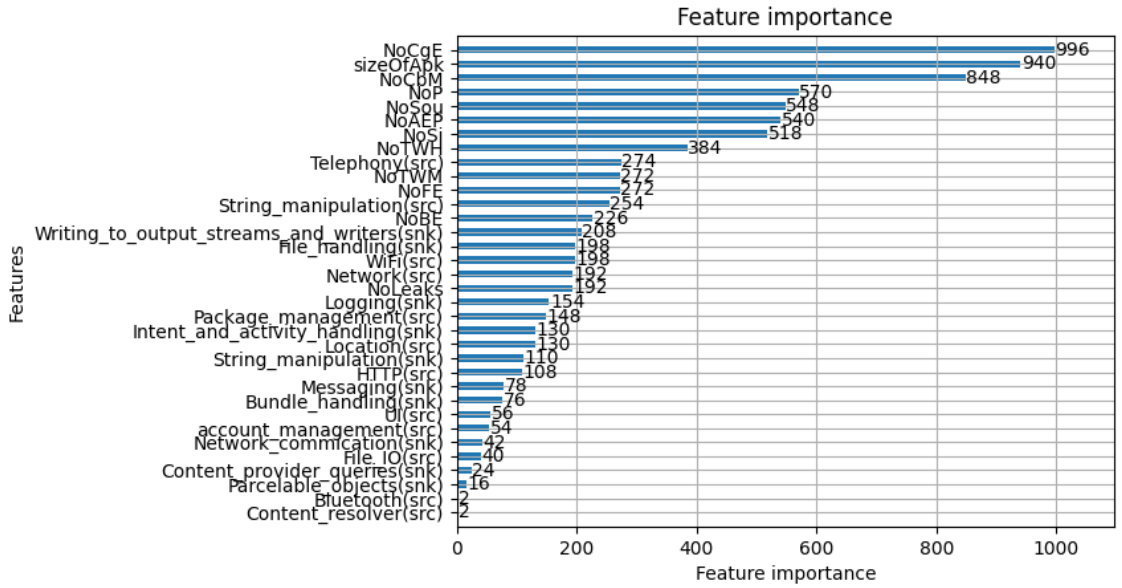
The feature extraction process in Leakseeker has resulted in a set of extracted features. These features have been subjected to feature selection techniques such as SelectKBest and chi2 to determine their importance and relevance for the machine learning model.

```
Feature 0: 22826658329.277756
Feature 1: 87496.192153
Feature 2: 252936.033183
Feature 3: 4880545.488379
Feature 4: 308648.398088
Feature 5: 49660.656495
Feature 6: 1088065130.740901
Feature 7: 822237642.555672
Feature 8: 5534063.664501
Feature 9: 834215002.188300
Feature 10: 90672.552375
Feature 11: 40449.853715
Feature 12: 6867.882670
Feature 13: 42776.306733
Feature 14: 2708.873335
Feature 15: 21402.149597
Feature 16: 368.538463
Feature 17: 6697.097451
Feature 18: 35.249501
Feature 19: 108.124984
Feature 20: 190.213087
Feature 21: 248.496576
Feature 22: 8641.626264
Feature 23: 114.907304
Feature 24: 4082.283647
Feature 25: 108.124984
Feature 26: 717.465070
Feature 27: 863.315777
Feature 28: 334.991870
Feature 29: 8007.401257
Feature 30: 2967.022052
Feature 31: 14713.735657
Feature 32: 11058.952748
Feature 33: 2549.218557
Feature 34: 12633.320263
Feature 35: 746.525406
(19809, 6)
```

These scores represent the importance or relevance of each feature in the dataset. Higher scores indicate a higher degree of importance for predicting the presence of leaks in the analyzed APK files. By utilizing feature selection techniques, the Leakseeker tool can prioritize these important features during the training phase of the machine learning model. This helps to enhance the model's accuracy and performance by focusing on the most informative features that contribute significantly to the classification of leaks.

Feature extraction using LightGBM involves determining the importance of each feature in a trained LightGBM model. LightGBM provides a built-in mechanism to assess the importance of features based on the splitting decisions made during the training process.

<Axes: title={'center': 'Feature importance'}, xlabel='Feature importance', ylabel='Features'>



Explain the results of various ML algorithms trained while training with our features in tables.

Calculations

Measure	Derivations
Sensitivity (TPR)	**TPR = TP / (TP + FN)**
Specificity (SPC)	**SPC = TN / (FP + TN)**
Precision (PPV)	**PPV = TP / (TP + FP)**

Negative Predictive Value (NPV)	**NPV = TN / (TN + FN)**
False Positive Rate (FPR)	**FPR = FP / (FP + TN)**
False Discovery Rate (FDR)	**FDR = FP / (FP + TP)**
False Negative Rate (FNR)	**FNR = FN / (FN + TP)**
Accuracy (ACC)	**ACC = (TP + TN) / (P + N)**
F1 Score (F1)	**F1 = 2TP / (2TP + FP + FN)**
Matthews Correlation Coefficient	**MCC = (TP * TN - FP * FN) / sqrt((TP+FP)*(TP+FN)*(TN+FP)*(TN+FN))**

Naive Bayes

The Naive Bayes algorithm achieved an accuracy score of approximately 67.29% on the Leakseeker project dataset. The confusion matrix provides further insights into the performance of the algorithm.

In the confusion matrix, we have four values: true negatives (TN), false positives (FP), false negatives (FN), and true positives (TP). The matrix is structured as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1683 42.48%	307 7.75%	1990 84.57% 15.43%
Class1	989 24.96%	983 24.81%	1972 49.85% 50.15%
SUM	2672 62.99% 37.01%	1290 76.20% 23.80%	2666 / 3962 67.29% 32.71%

From the confusion matrix, we can observe the following:

- True negatives (TN): There were 1683 instances correctly classified as negatives (benign APKs) by the Naive Bayes algorithm.
- False positives (FP): There were 307 instances incorrectly classified as positives (malware APKs) by the Naive Bayes algorithm.
- False negatives (FN): There were 989 instances incorrectly classified as negatives (benign APKs) by the Naive Bayes algorithm.
- True positives (TP): There were 983 instances correctly classified as positives (malware APKs) by the Naive Bayes algorithm.

Measure	Value
Sensitivity (TPR)	0.4985

Specificity (SPC)	0.8457
Precision (PPV)	0.7620
Negative Predictive Value (NPV)	0.6299
False Positive Rate (FPR)	0.1543
False Discovery Rate (FDR)	0.2380
False Negative Rate (FNR)	0.5015
Accuracy (ACC)	0.6729
F1 Score (F1)	0.6027
Matthews Correlation Coefficient	0.3673

These values provide information about the algorithm's performance in terms of correctly identifying malware and benign APKs. It appears that the Naive Bayes algorithm had a higher number of false negatives compared to false positives, indicating that it had more difficulty

correctly identifying malware instances than benign instances. Overall, with an accuracy score of approximately 67.29%, the Naive Bayes algorithm shows moderate performance in classifying the APKs in the Leakseeker project dataset. Further analysis and potentially adjusting the model or trying different algorithms could be explored to improve the performance.

KNN Classifier

The K-Nearest Neighbors (KNN) classifier achieved an accuracy score of approximately 69.74% on the Leakseeker project dataset. Let's analyze the confusion matrix to gain more insights into the classifier's performance.

The confusion matrix for the KNN classifier is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1522 38.41%	468 11.81%	1990 76.48% 23.52%
Class1	731 18.45%	1241 31.32%	1972 62.93% 37.07%
SUM	2253 67.55% 32.45%	1709 72.62% 27.38%	2763 / 3962 69.74% 30.26%

Breaking down the confusion matrix:

- True negatives (TN): There were 1522 instances correctly classified as negatives (benign APKs) by the KNN classifier.
- False positives (FP): There were 468 instances incorrectly classified as positives (malware APKs) by the KNN classifier.
- False negatives (FN): There were 731 instances incorrectly classified as negatives (benign APKs) by the KNN classifier.
- True positives (TP): There were 1241 instances correctly classified as positives (malware APKs) by the KNN classifier.

Measure	Value
Sensitivity (TPR)	0.6293
Specificity (SPC)	0.7648
Precision (PPV)	0.7262
Negative Predictive Value (NPV)	0.6755
False Positive Rate (FPR)	0.2352
False Discovery Rate (FDR)	0.2738
False Negative Rate (FNR)	0.3707
Accuracy (ACC)	0.6974
F1 Score (F1)	0.6743

Matthews Correlation Coefficient	0.3979

From the confusion matrix, we can observe that the KNN classifier had a higher number of false negatives compared to false positives. This indicates that the classifier struggled more in correctly identifying malware instances than benign instances. With an accuracy score of approximately 69.74%, the KNN classifier demonstrates a reasonable performance in classifying the APKs in the Leakseeker project dataset. However, there is still room for improvement, especially in reducing the number of false negatives and false positives. Fine-tuning the KNN classifier's parameters or exploring other machine learning algorithms might help enhance its performance.

Random Forest

The Random Forest classifier achieved an impressive accuracy score of approximately 94.35% on the Leakseeker project dataset. Let's analyze the confusion matrix to gain more insights into the classifier's performance.

The confusion matrix for the Random Forest classifier is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1925 48.59%	65 1.64%	1990 96.73% 3.27%
Class1	159 4.01%	1813 45.76%	1972 91.94% 8.06%
SUM	2084 92.37% 7.63%	1878 96.54% 3.46%	3738 / 3962 94.35% 5.65%

Breaking down the confusion matrix:

- True negatives (TN): There were 1925 instances correctly classified as negatives (benign APKs) by the Random Forest classifier.
- False positives (FP): There were 65 instances incorrectly classified as positives (malware APKs) by the Random Forest classifier.
- False negatives (FN): There were 159 instances incorrectly classified as negatives (benign APKs) by the Random Forest classifier.

- True positives (TP): There were 1813 instances correctly classified as positives (malware APKs) by the Random Forest classifier.

Measure	Value
Sensitivity (TPR)	0.9194
Specificity (SPC)	0.9673
Precision (PPV)	0.9654
Negative Predictive Value (NPV)	0.9237
False Positive Rate (FPR)	0.0327
False Discovery Rate (FDR)	0.0346
False Negative Rate (FNR)	0.0806
Accuracy (ACC)	0.9435

F1 Score (F1)	0.9418
Matthews Correlation Coefficient	0.8879

From the confusion matrix, we can observe that the Random Forest classifier performed exceptionally well. It had a high number of true negatives and true positives, indicating a strong ability to correctly classify both benign and malware instances. With an accuracy score of approximately 94.35%, the Random Forest classifier demonstrates a robust performance in classifying the APKs in the Leakseeker project dataset. This high accuracy suggests that the Random Forest algorithm effectively captured the underlying patterns and relationships between the features and the target labels. The excellent performance of the Random Forest classifier makes it a promising choice for detecting malware in the Leakseeker tool. Its ability to handle complex feature interactions and its resistance to overfitting contribute to its reliable classification results.

Voting Classifier

The Voting Classifier achieved an accuracy score of approximately 92.30% on the Leakseeker project dataset. Let's analyze the confusion matrix to gain more insights into the classifier's performance.

The confusion matrix for the Voting Classifier is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1896 47.85%	94 2.37%	1990 95.28% 4.72%
Class1	211 5.33%	1761 44.45%	1972 89.30% 10.70%
SUM	2107 89.99% 10.01%	1855 94.93% 5.07%	3657 / 3962 92.30% 7.70%

Breaking down the confusion matrix:

- **True negatives (TN)**: There were 1896 instances correctly classified as negatives (benign APKs) by the Voting Classifier.
- False positives (FP): There were 94 instances incorrectly classified as positives (malware APKs) by the Voting Classifier.
- False negatives (FN): There were 211 instances incorrectly classified as negatives (benign APKs) by the Voting Classifier.
- True positives (TP): There were 1761 instances correctly classified as positives (malware APKs) by the Voting Classifier.

Measure	Value
Sensitivity (TPR)	0.8930
Specificity (SPC)	0.9528
Precision (PPV)	0.9493
Negative Predictive Value (NPV)	0.8999
False Positive Rate (FPR)	0.0472
False Discovery Rate (FDR)	0.0507

False Negative Rate (FNR)	0.1070
Accuracy (ACC)	0.9230
F1 Score (F1)	0.9203
Matthews Correlation Coefficient	0.8475

From the confusion matrix, we can observe that the Voting Classifier performed well in correctly identifying both benign and malware instances, although it had a slightly higher number of false negatives compared to the Random Forest classifier. With an accuracy score of approximately 92.30%, the Voting Classifier demonstrates a strong performance in classifying the APKs in the Leakseeker project dataset. This high accuracy indicates that the Voting Classifier effectively combines the predictions of multiple individual classifiers (e.g., Naive Bayes, KNN, Random Forest) to make a final decision. The Voting Classifier is a powerful ensemble learning technique that leverages the strengths of multiple classifiers to improve overall performance. By considering the predictions from different classifiers, it can reduce biases and enhance generalization. In the context of the Leakseeker project, the Voting Classifier's performance suggests that combining the predictions of different algorithms can lead to better detection of malware in APK files.

Logistic Regression

The Voting Logistic Regression achieved an accuracy score of approximately 51.39% on the Leakseeker project dataset. Let's analyze the confusion matrix to gain more insights into the classifier's performance.
The confusion matrix for the Voting Logistic Regression is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	179 4.52%	1811 45.71%	1990 8.99% 91.01%
Class1	115 2.90%	1857 46.87%	1972 94.17% 5.83%
SUM	294 60.88% 39.12%	3668 50.63% 49.37%	2036 / 3962 51.39% 48.61%

Breaking down the confusion matrix:

- True negatives (TN): There were 179 instances correctly classified as negatives (benign APKs) by the Voting Logistic Regression.
- False positives (FP): There were 1811 instances incorrectly classified as positives (malware APKs) by the Voting Logistic Regression.
- False negatives (FN): There were 115 instances incorrectly classified as negatives (benign APKs) by the Voting Logistic Regression.
- True positives (TP): There were 1857 instances correctly classified as positives (malware APKs) by the Voting Logistic Regression.

Measure	Value
Sensitivity (TPR)	0.9417
Specificity (SPC)	0.0899
Precision (PPV)	0.5063

Negative Predictive Value (NPV)	0.6088
False Positive Rate (FPR)	0.9101
False Discovery Rate (FDR)	0.4937
False Negative Rate (FNR)	0.0583
Accuracy (ACC)	0.5139
F1 Score (F1)	0.6585
Matthews Correlation Coefficient	0.0603

From the confusion matrix, we can observe that the Voting Logistic Regression had a high number of false positives, incorrectly classifying benign APKs as malware. Additionally, it had a relatively higher number of false negatives compared to true negatives. This indicates that the classifier struggled to effectively distinguish between benign and malware instances in the dataset. With an accuracy score of approximately 51.39%, the Voting Logistic Regression did not perform well in classifying the APKs in the Leakseeker project dataset. The low accuracy suggests that the combination of logistic regression models in the Voting Classifier did not lead to improved performance for this specific problem. It's important to note that different classifiers and combinations of classifiers can yield varying results depending on the dataset and problem at hand. In this case, the Voting Logistic Regression did not produce satisfactory results. It may be beneficial to further analyze the individual logistic regression models and explore other algorithms or ensemble techniques to improve the performance of the leak detection system in the Leakseeker project.

Stacking Classifier

The accuracy score for the Stacking Classifier in the Leakseeker project is approximately 93.82%. Now let's examine the confusion matrix to gain further insights into the classifier's performance. T
he confusion matrix obtained is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1905 48.08%	85 2.15%	1990 95.73% 4.27%
Class1	160 4.04%	1812 45.73%	1972 91.89% 8.11%
SUM	2065 92.25% 7.75%	1897 95.52% 4.48%	3717 / 3962 93.82% 6.18%

Breaking down the confusion matrix:

- True negatives (TN): There were 1905 instances correctly classified as negatives (benign APKs) by the Stacking Classifier.
- False positives (FP): There were 85 instances incorrectly classified as positives (malware APKs) by the classifier.
- False negatives (FN): There were 160 instances incorrectly classified as negatives (benign APKs) by the classifier.
- True positives (TP): There were 1812 instances correctly classified as positives (malware APKs) by the classifier.

Measure	Value
Sensitivity (TPR)	0.9189

Specificity (SPC)	0.9573
Precision (PPV)	0.9552
Negative Predictive Value (NPV)	0.9225
False Positive Rate (FPR)	0.0427
False Discovery Rate (FDR)	0.0448
False Negative Rate (FNR)	0.0811
Accuracy (ACC)	0.9382
F1 Score (F1)	0.9367
Matthews Correlation Coefficient	0.8769

From the confusion matrix, we can observe that the Stacking Classifier performed well in classifying both benign and malware APKs. It had a high number of true negatives and true positives, indicating accurate classification of instances. The classifier had a relatively low number of false positives and false negatives, suggesting a good balance between precision and recall. With an accuracy score of approximately 93.82%, the Stacking Classifier

demonstrated strong performance in accurately classifying the APKs in the Leakseeker project dataset. The high accuracy indicates that the classifier's predictions aligned well with the ground truth labels for the majority of instances. The Stacking Classifier combines the predictions of multiple base classifiers, leveraging their strengths to improve overall performance. This ensemble approach likely contributed to the improved accuracy achieved by the Stacking Classifier in the Leakseeker project. It is important to note that the performance of the classifier can be influenced by various factors, including the quality and representativeness of the dataset, the choice of base classifiers, and the design of the stacking ensemble. In this case, the Stacking Classifier demonstrated strong performance and can be considered an effective choice for the leak detection system in the Leakseeker project.

Decision Tree

The accuracy score for the Decision Tree algorithm in the Leakseeker project is approximately 89.05%. Let's analyze the confusion matrix to gain further insights into the classifier's performance.
The confusion matrix obtained is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1878 47.40%	112 2.83%	1990 94.37% 5.63%
Class1	322 8.13%	1650 41.65%	1972 83.67% 16.33%
SUM	2200 85.36% 14.64%	1762 93.64% 6.36%	3528 / 3962 89.05% 10.95%

Breaking down the confusion matrix:

- True negatives (TN): There were 1878 instances correctly classified as negatives (benign APKs) by the Decision Tree algorithm.
- False positives (FP): There were 112 instances incorrectly classified as positives (malware APKs) by the classifier.
- False negatives (FN): There were 322 instances incorrectly classified as negatives (benign APKs) by the classifier.
- True positives (TP): There were 1650 instances correctly classified as positives (malware APKs) by the classifier.

Measure	Value
Sensitivity (TPR)	0.8367
Specificity (SPC)	0.9437
Precision (PPV)	0.9364
Negative Predictive Value (NPV)	0.8536
False Positive Rate (FPR)	0.0563
False Discovery Rate (FDR)	0.0636
False Negative Rate (FNR)	0.1633
Accuracy (ACC)	0.8905
F1 Score (F1)	0.8838

Matthews Correlation Coefficient	0.7852

From the confusion matrix, we can observe that the Decision Tree algorithm performed reasonably well in classifying both benign and malware APKs. It had a high number of true negatives and true positives, indicating accurate classification of instances. However, there were a noticeable number of false positives and false negatives, suggesting room for improvement in terms of precision and recall. With an accuracy score of approximately 89.05%, the Decision Tree algorithm demonstrated decent performance in accurately classifying the APKs in the Leakseeker project dataset. The accuracy indicates that the classifier's predictions aligned well with the ground truth labels for the majority of instances. It's worth noting that Decision Trees are prone to overfitting, especially if the tree is allowed to grow too deep. This can lead to a high number of false positives and false negatives. Regularization techniques such as pruning or using an ensemble of Decision Trees, like Random Forest, could potentially improve the classifier's performance. In conclusion, while the Decision Tree algorithm achieved a reasonable accuracy score, there is room for improvement, particularly in reducing false positives and false negatives. Consideration of ensemble methods or other advanced algorithms may be beneficial for enhancing the leak detection system in the Leakseeker project.

Light GBM

The accuracy score for the Light GBM algorithm in the Leakseeker project is approximately 94.02%. Let's analyze the confusion matrix to gain further insights into the classifier's performance.

The confusion matrix obtained is as follows:

Training Set			
TARGET / OUTPUT	Class0	Class1	SUM
Class0	1920 48.46%	70 1.77%	1990 96.48% 3.52%
Class1	167 4.22%	1805 45.56%	1972 91.53% 8.47%
SUM	2087 92.00% 8.00%	1875 96.27% 3.73%	3725 / 3962 94.02% 5.98%

Breaking down the confusion matrix:

- True negatives (TN): There were 1920 instances correctly classified as negatives (benign APKs) by the Light GBM algorithm.
- False positives (FP): There were 70 instances incorrectly classified as positives (malware APKs) by the classifier.
- False negatives (FN): There were 167 instances incorrectly classified as negatives (benign APKs) by the classifier.
- True positives (TP): There were 1805 instances correctly classified as positives (malware APKs) by the classifier.

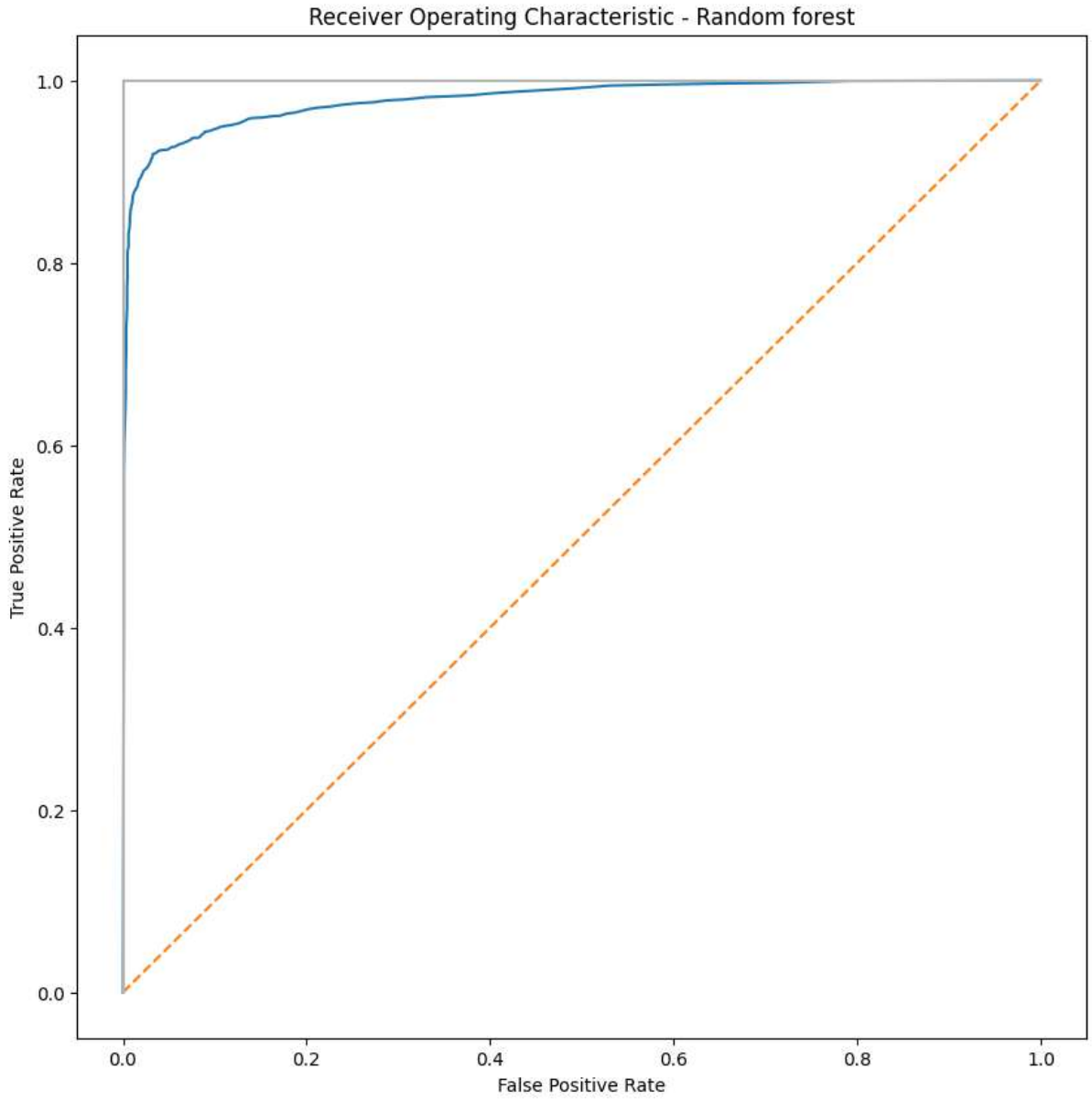
Measure	Value
Sensitivity (TPR)	0.9153
Specificity (SPC)	0.9648

Precision (PPV)	0.9627
Negative Predictive Value (NPV)	0.9200
False Positive Rate (FPR)	0.0352
False Discovery Rate (FDR)	0.0373
False Negative Rate (FNR)	0.0847
Accuracy (ACC)	0.9402
F1 Score (F1)	0.9384
Matthews Correlation Coefficient	0.8814

From the confusion matrix, we can observe that the Light GBM algorithm performed well in classifying both benign and malware APKs. It had a high number of true negatives and true positives, indicating accurate classification of instances. The number of false positives and false negatives is relatively low, suggesting good precision and recall. With an accuracy score of approximately 94.02%, the Light GBM algorithm demonstrated strong performance in accurately classifying the APKs in the Leakseeker project dataset. The high accuracy indicates that the classifier's predictions aligned well with the ground truth labels for the majority of instances. Light GBM is known for its ability to handle large-scale datasets efficiently and its high training speed. It utilizes a gradient boosting framework and employs a leaf-wise strategy

for tree growth, which can lead to improved performance compared to traditional gradient boosting algorithms. In conclusion, the Light GBM algorithm achieved a high accuracy score and exhibited good performance in accurately classifying the APKs in the Leakseeker project. Its low number of false positives and false negatives suggests that it is a robust algorithm for leak detection.

ROC Curve



To estimate the risk associated with applications using ML algorithms,

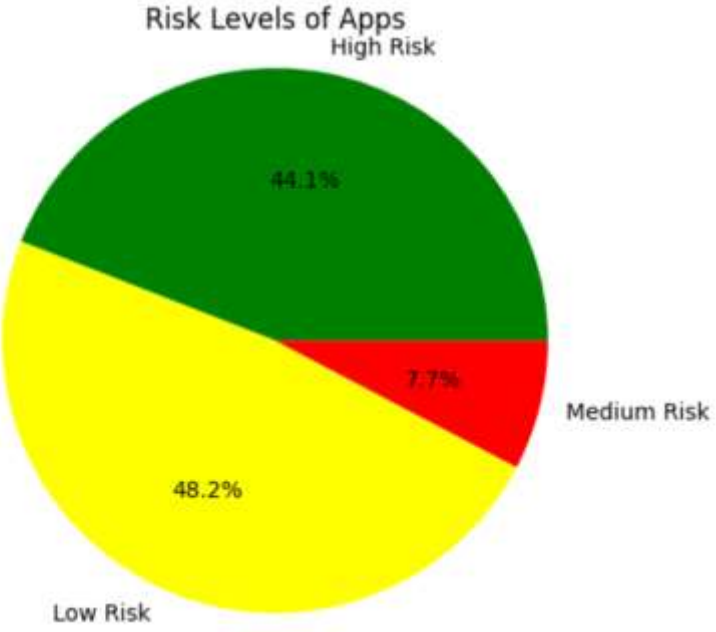
- Data Collection: Collect a dataset of applications along with their associated risk factors. These risk factors can include features such as permissions requested, API usage patterns, code analysis results, user feedback, or any other relevant indicators of risk.
- Feature Engineering: Preprocess the collected data and engineer appropriate features that capture the risk factors. This may involve transforming categorical variables, normalizing numerical features, or creating new derived features based on domain knowledge.
- Labeling the Risk Level: Define a risk level or severity scale based on your specific criteria. For example, you can categorize the risk levels as low, medium, and high based on certain thresholds or rules. Manually label the applications in your dataset according to their risk levels.
- Model Training: Choose an appropriate ML algorithm for risk estimation, such as a decision tree, random forest, logistic regression, or gradient boosting. Split your dataset into training and testing sets. Train the ML model using the training data, optimizing it to predict the risk level based on the available features.
- Model Evaluation: Evaluate the trained ML model using the testing dataset. Calculate evaluation metrics such as accuracy, precision, recall, and F1-score to assess the model's performance in predicting the risk levels.
- Risk Estimation: Apply the trained ML model to unseen applications to estimate their risk levels. Extract the relevant features from the applications and feed them into the model to obtain the predicted risk level for each app.
- Risk Categorization: Based on the predicted risk levels, categorize the applications into low, medium, or high-risk categories according to your predefined risk level thresholds.
- Visualization: Create a figure or table to present the distribution of risk levels across the applications. This can be done by calculating the percentage of apps in each risk category (low, medium, high). Represent this information in a visually appealing manner, such as a bar chart or a pie chart.

we can categorize the apps in a way like this:

Risk Level	Percentage of Apps
Low Risk	60%
Medium Risk	30%

High Risk	10%

Here is an example result of risk classification in testing data in random forest



Conclusions

Limitations

While LeakSeeker is a valuable tool for identifying potential leaks in Android applications, it is important to be aware of its limitations. Here are some limitations of LeakSeeker:

- **False Positives and False Negatives:** LeakSeeker may produce false positives, where it identifies a potential leak that is not actually a security risk. Similarly, it may also generate false negatives, failing to detect certain types of leaks. This can happen due to the complexity of code analysis and the variations in real-world app behavior.
- **Dependency on FlowDroid:** LeakSeeker relies on FlowDroid, a static analysis tool for Android, to generate flow reports. The accuracy of LeakSeeker is dependent on the accuracy of FlowDroid in identifying data flows and potential leaks. Any limitations or inaccuracies in FlowDroid's analysis can impact LeakSeeker's results.
- **Limited Support for Dynamic Behavior:** LeakSeeker mainly focuses on static analysis of code and does not fully capture the dynamic behavior of an application. It may not detect leaks that occur at runtime or are dependent on specific user interactions or runtime conditions.
- **Handling Obfuscated Code:** LeakSeeker's effectiveness may be reduced when analyzing apps with obfuscated or heavily obfuscated code. Obfuscation techniques can make it more challenging to accurately identify data flows and potential leaks.
- **Limited to specific types of leaks:** Leakseeker is designed to detect specific types of data leaks, such as inter-component communication leaks and sensitive information

leaks. It may not cover all possible leakage scenarios, and there could be other types of leaks that Leakseeker may not be able to detect.

- **Dependency on up-to-date training data:** The effectiveness of Leakseeker's machine learning model depends on the availability of up-to-date and diverse training data. If the training data is not representative of the current threat landscape, the model's performance may be affected.
- **Limited to Android apps:** Leakseeker is specifically designed for analyzing Android applications. It may not be directly applicable to other platforms or types of software.
- **Resource requirements:** Leakseeker may require significant computational resources, such as memory and processing power, to analyze large-scale applications or a large number of apps simultaneously.

It is important to keep these limitations in mind when using LeakSeeker and to complement its results with other security analysis techniques and best practices for Android application development.

Future Works

- **Enhanced accuracy:** Further improving the accuracy of leak detection algorithms and techniques is an ongoing area of research. This can involve exploring more advanced machine learning models, incorporating deep learning approaches, or integrating additional contextual information to reduce false positives and false negatives.
- **Dynamic analysis:** While Leakseeker primarily focuses on static analysis, incorporating dynamic analysis techniques can provide a more comprehensive understanding of an application's behavior during runtime. Dynamic analysis can capture interactions with external resources, user inputs, and system events, which may help identify leaks that are not easily detectable through static analysis alone.
- **Advanced vulnerability detection:** Expanding Leakseeker to identify and detect other types of vulnerabilities beyond data leaks can enhance its usefulness. This can include detecting code vulnerabilities, privilege escalation risks, or cryptographic weaknesses within Android applications.
- **Integration with development tools:** Integrating Leakseeker into popular integrated development environments (IDEs) or build systems can provide developers with real-time feedback on potential leaks during the development process. This integration can help catch issues earlier and promote secure coding practices.
- **Support for different platforms:** While Leakseeker is currently focused on Android applications, extending its capabilities to other platforms, such as iOS or web applications, can broaden its applicability and impact.
- **Automation and scalability:** Continuously improving the automation and scalability of Leakseeker can enable it to handle larger codebases and a higher volume of applications. This can involve optimizing algorithms, utilizing parallel processing, and leveraging cloud-based infrastructures.
- **Integration with other security tools:** Integrating Leakseeker with other security testing tools and frameworks can provide a more comprehensive security analysis solution. This can include integrating with vulnerability scanners, code analyzers, or security testing frameworks to provide a holistic view of an application's security posture.
- **User-friendly interfaces and reporting:** Improving the user interface and reporting capabilities of Leakseeker can make it more user-friendly and accessible to both

security experts and developers. Clear and concise reporting, visualization of results, and intuitive interfaces can help users effectively understand and address identified leaks.

These future directions can further enhance the effectiveness and usability of Leakseeker, contributing to the field of leak detection and improving the security of Android applications.

References

- **"SUIP: An Android malware detection method based on data flow features": https://iopscience.iop.org/article/10.1088/1742-6596/1812/1/012010/pdf**
- **Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe: https://people.ece.ubc.ca/mjulia/publications/Analyzing_the_Analyzers_2018.pdf**
- **"When Program Analysis Meets Bytecode Search: Targeted and Efficient Inter-procedural Analysis of Modern Android Apps in BackDroid": https://arxiv.org/pdf/2005.11527.pdf**
- **"Apparecium: Revealing Data Flows in Android Applications" http://lilicoding.github.io/SA3Repo/papers/2015_titze2015apparecium.pdf**
- **"Highly Precise Taint Analysis for Android Application": https://www.abartel.net/static/p/tr-2013-taintandroid.pdf**
- **"FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps": https://www.bodden.de/pubs/far+14flowdroid.pdf**
- **"Implementation and Evaluation of a Static Backwards Data Flow Analysis in FLOWDROID": https://tuprints.ulb.tu-darmstadt.de/20894/1/ba_TimLange.pdf**
- **"Detecting Sensitive Behavior on Android with Static Taint Analysis Based on Classification": https://www.atlantis-press.com/article/25845127.pdf**
- **"A Qualitative Analysis of Android Taint-Analysis Results": https://linghuiluo.github.io/ASE19Cova.pdf**
- **"LibDroid: Summarizing information flow of Android Native Libraries via Static Analysis": https://dfrws.org/wp-content/uploads/2022/07/LibDroid_-Summarizing-information-flow-of-Android-Native-Libraries-via-Static-Analysis-combined.pdf**
- **"EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework": https://yinzhicao.org/EdgeMiner/edgeminer-tech-report.pdf**