Evaluation Of Android Applications Bug Finding Tools

A thesis submitted

in Partial Fulfillment of the Requirements

for the Degree of

Master of Technology

by

Saksham Jain

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

November, 2022

CERTIFICATE

It is certified that the work contained in the thesis titled **Evaluation Of Android Applications Bug Finding Tools**, by **Saksham Jain**, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Subhajit Roy

Prof. Subhajit Roy

Department of Computer Science & Engineering

IIT Kanpur

November, 2022

ABSTRACT

Name of student: **Saksham Jain** Roll no: **20111053**

Degree for which submitted: **Master of Technology**

Department: **Computer Science & Engineering**

Thesis title: **Evaluation Of Android Applications Bug Finding Tools**

Name of Thesis Supervisor: **Prof. Subhajit Roy**

Month and year of thesis submission: **November, 2022**

Nowadays, smart mobile phones have become very important part of everyone's life. Everyone stores their memories and private data in it. People with wrong intention can exploit the users data by hacking into their phone. The most easiest way to attack is through applications installed on the mobile phone. With over 70% market share of Android worldwide, it has become the most popular smart mobile phone OS globally. So, Android is the best option for hackers to target maximum people worldwide. Android doesn't provide its own security framework. It just have a set of permissions that can prevent outside attack but those are not enough. There are mainly two types of analysis done on Android APK :- static analysis and dynamic analysis. Static analysis mainly analyse the application code without running application on Android device and dynamic analysis analyse the application while running on a device.

In this work, we have explored 22 open-source Android vulnerability analysis tools. We have performed static and dynamic analysis on the Droidbench benchmark. This benchmark contains thirteen categories of test cases that checks different aspect of android security. Out of these 22 analysis tools, 16 are static analysis tools and 6 are dynamic analysis tools. Then, this work also discusses best performing tools in each category of Droidbench benchmark.

To my family

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis supervisor Prof. Subhajit Roy. This work is a result of his advice and support. I would like to thank him for giving me the opportunity to work on this topic. His guidance and resources allowed me to make progress of thesis.
I would also like to thank department staff for their support and enabling availability of technical resources that helped me to experiment with my work and understand the work deeply. I want to thank the entire department for the facilities that helped me to carry out the thesis work and experience the flavour of studying in such premier institute.
I would also like to thank God for helping me in going through all the difficulties.
Last but not least, I would like to thank my family and friends for their constant support.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

World has witnessed the fastest technological development in the last decade. With this development, people are able to do time consuming tasks from the comfort of their homes in just few seconds. Smartphones have become a necessary part for everyone's life these days. Android has emerged as a most used mobile operating system in last ten years. The Android phone architecture was developed by Android Inc. but now owned by Google and launched it as Android Open Source Project (AOSP) in 2007. Latest version of Android are developed keeping the security in mind but users can also install third party application which may be vulnerable to attack or those application can leak private user data by gaining permissions on Android device. Applications installed on smartphones are the easiest way to attack for a cyber-criminals.

Many Android developers lack security implementations expertise. One of the most overlooked parts of Android development is security. Application design and development take up a lot of developers' time, but security is rarely given any attention. Users are tricked by attackers into downloading software from dubious sources. After installing an APK, these files may include malicious software that grants an attacker remote access to the device. Other carriers of attacks are - SMS - that claims to reward money if you click on the given link, Phising Email - SPAM emails may steal information from device, Spyware - some applications may be used to spy on the mobile users, Rooting - of an Android phone that also makes device vulnerable.There is a big list of attacks that occur everyday and gone unseen by many users. These attacks result in loss of big amount of money and users private data worldwide. These reasons makes security of Android applications important. There is a wide variety of open source tools that are available online to test security of Android APKs. There is two type of analysis that can be performed on Android APKs -

- **Static Analysis** - Static analysis of an Android APK is done using automated tool that does not require the application to run on the Android device/emulator. It studies the program code to search for harmful lines of code that may provide back door access to unauthenticated user to the user's private data. It also scans the whole code for application coding vulnerabilities and other security aspects. This type of analysis gives an insight of the structure of code and can help guarantee that it complies with best coding practices.
- **Dynamic Analysis** - Dynamic analysis is performed when application is running on mobile device/emulator. This benefits the developer or security professional because they can see the actual behaviour of code and analyse the root cause at run-time.

Android APK contains all required files that are required for its execution. Following is the list of such important files and folders -

- **lib** - This contains native libraries that are required by particular device architecture for execution.
- **res** - This directory contains images, Xml layout files etc.
- **META-INF** - This contains manifest file, signature information, resources list in the archive etc.
- **AndroidManifest.xml** - This file contains information of permissions required by app, name, version of app.

- **classes.dex** - The Java classes that have been compiled for execution on devices.
- **resources.arsc** - This contains all compiled resources like strings etc.

This report contains some of the open-source static and dynamic analysis tools and their performance evaluation on DroidBench open source test suite that comprises of 119 test cases at present.

Chapter 2

Literature Survey

Android security has attracted many researchers and security professionals towards itself due to high demand of Android based smartphones. There has been plethora of work done on Android security. [Haris et al.] covers evolution of Android over the years that make Android OS being used worldwide. [Meshram and Thool] discusses several vulnerabilities of Android OS and Android devices security. [Singh and Sharma] lists challenges in smartphone security while transferring the data and suggests solution for these challenges.

Several reverse engineering tools exists in the market. Apktool [Winsniewski] is widely used tool for compilation and decompilation of Android APKs. It uses Smali [Gruver] to perform this task. Additionally, it gives project-like file structure after decompilation and automatically performs some repetitive operations, like creating Apks, sign the Apks etc. Androguard [Androguard] is another such tool that is used by many static and dynamic analyzers in the intermediate step of their analysis when an Android APK is given to them. It also supports Frida [Nowsecure] for dynamic analysis. Dex2jar [Bob] is a tool that is used to convert .dex files to .class files that are zipped as jar. Enjarify [Google] is developed by Google that is used to convert Dalvik bytecode to Java bytecode. JADX [Skylot] is also used to convert Dex code to Java code. It also comes bundled with Kali distribution as a GUI tool. These tools can analyze AndroidManifest.xml file to find out the permissions that are used by the application. [Albakri et al.] talks about other reverse engineering tool and their evaluation.

Soot [Vallée-Rai et al.] is a tool that optimizes java bytecode into three different intermediate representations: Baf, Jimple and Grimp. Each representations comes with its own motivation and usage. There are unlimited numbers of tools that are used to analyze Android application security. The scale at which Android app development is increasing, is very high when it is compared to the awareness of security measures to be followed during development [Green and Smith].

There are unlimited number of tools to detect vulnerabilities in Android apps. So, it is necessary to make some effort and assess the capabilities of tools and techniques. Various efforts have been made in this direction [Reaves et al.][Sadeghi et al.][Pauck et al.]. [Ranganath and Mitra] assess the quality of tools based on Ghera repository [Mitra and Ranganath]. Several other works have been done in order to raise awareness about security of applications in the developer community. These works performs a thorough literature review on security testing of Android apps [Kong et al.][Scandariato and Walden]. [Tam et al.] represents review on the best methods for analysing and detecting malware.

Some of the papers list various types of tools available for analysis like - Static analyzers, Dynamic analyzers, analyzers that uses machine learning approach, tools to test GUI [Kulkarni and Javaid]. [Tiwari and Velayutham] presents a static analysis tool for Android vulnerabilities called Android vulnerability Scanner. [Wei and Lie] develops a tool that performs memory-efficient vulnerability analysis in managed runtime. Now, lets see what type of testcases Droidbench contains.

Chapter 3

Bug Repository

In this work, we have used Droidbench benchmark to assess the performance of analysis tools. It has 13 testcases category containing 119 testcases. Although it offers testcases for static analysis issues, it can be used for both static and dynamic analysis. In this chapter, we will talk about source and sink of each testcase and its dataflow. The dataflow is mentioned in code files. Lets walk through these testcases -

- **ALIASING**

  Merge1 - Tainted data is shuffled between multiple heap objects. There is no connection between source and sink. Challenge for analysis tool is to compute aliases precisely otherwise false positive will be found.

- **ARRAYS AND LISTS**

  ArrayAccess1 - An array is created which contains untainted and tainted data. Here, untainted data is retrieved and leaked through SMS. The analysis tool must be able to distiguish between different array indices to find out whether tainted or untainted data is leaked.

  ArrayAccess2 - This test case also creates an array that contains tainted and untainted data. CalculateIndex() is used to retrieve the index that contains leaked data. But tool must be able to recognize that index returned by function contains untainted data.

ArrayCopy1 - This test case contains tainted data in array then this is to copied to new string array and then leaked to logs. The analysis tool must have a model for System.arraycopy().

ArrayToString1 - This test case has tainted data in array of string. Afterwards, it is converted to string using Arrays.toString() and then leaked through logs. The analysis tool must have the ability to model that Array.toString() invokes toString() for each object of array.

HashMapAccess1 - In this test case, hashmap has both tainted and untainted data. Then, untainted data is leaked via SMS. The analysis tool must be able to distinguish between different hash map entries to find that tainted data does not get leaked.

ListAccess1 - In this test case, list contains both tainted and untainted data and only untainted data is leaked through SMS. The analysis tool must be able to recognize differentiate between different list positions to recognize that untainted data is sent through SMS.

MultidimensionalArray1 - This test case has tainted data in multi-dimensional array and then it is leaked through logs. The analysis tool must be able to track data in multi-dimensional array.

■ **CALLBACKS**

AnonymousClass1 - This test case has a callback handler for location updates. This location data is stored in static fields that are then leaked to log. The analysis tool must have a capability to handle callbacks, anonymous inner class and static fields.

Button1 - In this test case, when user clicks on button then sink in triggered and leak data though SMS. The analysis tool must have capability to analyze the xml file layout and take lifecycle into account.

Button2 - This test case leaks data if button3 is clicked first and then button1/button2 is clicked. The analysis tool must be able to analyze listeners and to consider all permutations of button clicks.

Button3 - In this testcase, another callback handler's registers a new callback. Handler of Button2 callback leaks the data through SMS obtained by the first handler. The analysis tool must have provision to handle callback that are registered in other callback handlers.

Button4 - In this test case, the data leakage happens when user taps the button. The button handler is defined by XML. To detect this leak, the analysis must have a capability to analyze the layout XML code.

Button5 - This test case tests the correctness of modelling of button object maintained by the run-time and delivered to onClick events handler is defined in XML. The analysis tool must correctly model that a button is represented by a single object in the runtime and that object is delivered to multiple calls of onClick.

LocationLeak1 - This test case consists of location data leakage in the onResume() callback method. The source is placed into the onLocationChanged() callback method. The analysis tool must have capability to analyze the android activity lifecycle correctly and integrate the callback method onLocationChanged and detect the callback method as source.

LocationLeak2 - Difference between LocationLeak1 and this test case is just that the activity class directly implements the callback interface. Challenge here is same as LocationLeak1.

LocationLeak3 - This is same as LocationLeak1 but the handler of callback is in a specific class that is interface-decoupled from the activity.

MethodOverride1 - This test case overrides an internal Android method to leak the data. The analysis tool must have capability to detect that internal android method is overwritten.

MultiHandlers1 - This test case contains the 2 activities and also 2 handlers that do not leak data. So there is no connection between source and sink. The analysis tool must have capability to properly match callback handlers with the respective activities.

Ordering1 - This test case leaks the variable even before the tainted value is assigned to it. So no leakage in this test case. But the analysis tool must have capability to take care of order of callback registration and sink call into consideration.

RegisterGlobal1 - In this test case, the global lifecycle handler includes a source and sink. The analysis tool must have capability to handle globally-registered callback handlers.

RegisterGlobal2 - In this test case, the global lifecycle handler includes a source and sink. The analysis tool must have capability to handle globally-registered callback handlers.

Unregister1 - This test case first registers and then instantly unregisters a callback before it can even be invoked. The analysis tool must have capability to consider that callbacks can also be unregistered before invocation.

■ FIELD AND OBJECT SENSITIVITY

FieldSensitivity1 - This test case is designed to determine whether the analysis tool can distinguish between various object fields. Here, an object has to fields, out of these two one is private value and other is public value. Here public one is leaked which no data leakage occurs in actual.

FieldSensitivity2 - This test case is same as previous one but easier to detect because everything is implemented inside onCreate() function only. Here also, no data leakage occurs.

FieldSensitivity3 - In this test case, the object field that is private for a user is leaked. The analysis tool must have a capability to find that here object field that has private value is leaked.

FieldSensitivity4 - In this case, object has only one field which contains untainted data initially. First untainted data is leaked and then same object field is overwritten with tainted data. So here no data leakage occurs. The analysis tool must have capability to consider the order of execution of statements.

InheritedObjects1 - In this test case, a variable is initialized based on the if-else condition. Condition of "if" is such that only "if" block will always be executed that leaks some private data. So, analysis tool must have capability to see the control-flow in such cases.

ObjectSensitivity1 - This test case has two list of same type but only one has private data. The list which has public data is leaked, so no data leakage. The analysis tool must have a capability to differentiate between two objects of same type that are initialized by same constructor.

ObjectSensitivty2 - In this test case, one local and an object is field is initialized with private data initially. Then, both of them are overwritten by same untainted value and passed onto sink. This means there is no leakage in this case. So, analysis tool must be able to detect that overwritten values are public and hence no leakage.

■ **INTER-APP COMMUNICATION**

Echoer - This test case is based on inter-app communication. So, this case receives data from sender and echos it again to the sender. This case leaks private data through logs.

SendSMS - This test case leaks private data through SMS. In this, source reads the private data, send it to echoer first and then it is leaked via test message.

StartActivityForResult1 - This test case checks if the tool has capability to detect if leak occurs through file system and then passed on over to other activity using startActivityResult that adds private data to the file. Here, private data leakage occurs.

■ INTER-COMPONENT COMMUNICATION

ActivityCommunication1 - This test case leaks the user's private data with use of two activities. Here, one activity contains source and another contains sink. Source Activity initializes the static field of sink activity and then sink activity leaks information. So, tool must be able to consider lifecycle of activities and also their execution order in any sequence.

ActivityCommunication2 - This test case also uses two activities to leak the data. In this, one activity shares the data with another activity using intent object via putExtra() and then other activity leaks data through logs. The tool must be able to track the transfer of private data between activities.

ActivityCommunication3 - This test case also uses componentName with class constant object along with intent object to share data between two activities. The analysis tool must be able to detect the component and track data transfer between activities.

ActivityCommunication4 - This case uses concatenation of strings while creating an object for intent. This also leaks through log by sharing data between activities. The analysis tool must be able to detect the the concatenation operation of two strings and then find intent related to resulted string and then track the data flow.

ActivityCommunication5 - This test case is much like ActivityCommunication3 but here class name is given as string. So in order to find the leakage, tool must be able to to resolve intent's component from component name and also track the data flow to other activity.

ActivityCommunication6 - In this case, an intent object is first added to a linked list and then linked list operations are used to start new activity. It also leaks data through logs. Tool must have capability to detect that an intent is started using linked list operations.

ActivityCommunication7 - Here, object of class of the activity to be started using intent is created first and then that is used to get the class name while creating intent object. In order to detect data leak in this case, a tool must have capability to solve an intent of a variable activity class.

ActivityCommunication8 - This case contains intent object that has to resolve the string that has been passed through a linked list. If tool has this capability, then it would be able to detect the leakage.

BroadcastTaintAndLeak1 - This case leaks the private data through BroadcastReceiver. An intent object contains private data using intent's putExtra() and then it is broadcasted using sendBroadcast() which is received by BroadcastReceiver object. The analysis should be able to detect broadcast receiver.

ComponentNotInManifest1 - This test case contains sink in the activity which is not present in android manifest file. So, since activities which are not present in manifest file can no be started, so no data leakage. The analysis tool must detect that activity is not present inside manifest file and hence it cannot be started.

EventOrdering1 - This case makes use of SharedPreferences to leak private data to logs. This case leaks the data when app is opened for second time. In first time, an activity saves the data in sharedpreferences and when it is opened for second time, it leaks data to log. So tool must be able to take use of SharedPreferences into account and also execution of application.

IntentSink1 - This test case leaks the as a result of an activity using setResult(). This function take result code and an intent object as a argument. So, this case makes use of putExtra() to leak data through intent. The analysis tool must be able to track that intent object that is being sent to setResult contains private data.

IntentSink2 - This case leaks data when user interacts with layout of app. When user clicks the button, it triggers the function that leaks the data through intent object based on user input. The analysis tool should be able to parse xml file and follow the flow of private data to find sink.

IntentSource1 - This test case leaks data in two ways. First one is by starting a new activity and leak the data to other app using intent. Second way is making use of onActivityResult() to leak data through log. The tool must be able to detect both ways of leaking the data.

ServiceCommunication1 - This test case first send the private data to a service and then this service handler leaks the data to log. The tool must have capability to analyze inter-component communication message passing.

SharedPreference1 - This test case tests the ability of tool to correctly model the flow of data using SharedPreferences. Here, one activity saves the data in sharedpreferences and another reads it and leaks to log.

Singletons1 - This test case shares a single object between two activities and one of them leaks the data only if a particular ordering of activities takes place.

UnresolvableIntent1 - This test case starts an activity with an intent that can not be statically resolved. Any one activity can be started based on result of if-else condition and both activity leaks the data to log.

■ **LIFECYCLE**

ActivityLifecycle1 - This test case leaks private data through url. Here, source method concats the private data to a url and then send that url to HttpURLConnection object which leaks it. The tool must be able track flow of activity lifecycle correctly and also handle try-catch blocks.

ActivityLifecycle2 - This test case checks if tool has the capability to track activity lifecycle and also to check the callback method that is inherited from super class. In this test case, source method stores the private data in static variable that is declared in base class and then base class method leaks it which is called in activity lifecycle.

ActivityLifecycle3 - This test case checks if tool have capability to handle leaks via instance-state callback methods. In this test case, one callback method assigns the private data to static variable and another callback method leaks it via SMS.

ActivityLifecycle4 - This case checks if tool has capability to handle loops in activity lifeycle by making the onResume() method as source and onPause() method as sink. In android activity lifecycle, onPause() is called is called before onResume().

ActivitySavedState1 - This test case also checks tool capability to track activity lifecycle and also if it can check activity saved state. This case saves private data during onSaveInstanceState() and leaks it during next onCreate().

ApplicationLifecycle1 - This test case retrieves the private data when application is launched and stores it in a static variable and then leaks it when main activity is launched again and it hit onResume() method. This checks the ability of tool to handle application lifecycle.

ApplicationLifecycle2 - In this test case, private data is obtained in onCreate() method when the application is launched and it is stored in a static variable. Then, it is leaked in onLowMemmory() callback. The tool should be able to handle callbacks in Application object.

ApplicationLifecycle3 - This test case obtains private data when a content provider object is launched and leaks it in further via message. Here, point to note is that Application.onCreate() is called after ContentProvider.onCreate() method and this is the reason this test case leaks the data.

AsynchronousEventOrdering1 - In this test case, analysis tool must check all possible orderings of activity lifecyle to detect the leak. This test case obtains the value in onResume() method and leaks it in onStop() method but if onLowMemory() is called before onStop(), then there will be no data leak.

BroadcastReceiverLifecycle1 - This test case obtains the private data in onReceive() method of BroadcastReceiver lifecycle and leaks it after statisfying a if condition. The tool must be able to track BroadcastReceiver lifecyle and also check the conditional branch.

BroadcastReceiverLifecycle2 - This test case uses nested classes to dynamically register the broadcast receiver, so tool should be able to handle broadcast receivers that are dynamically registered. This case obtains private data in onCreate() method of Activity lifecycle and leaks it in onReceive() method of BroadcastReceiver lifecyle.

EventOrdering1 - This test case also checks if tool is checking all possible order of an activity lifecycle. This test case on leaks data if onLowMemory() method is called two times without calling onContentChanged() in between the two calls.

FragmentLifecycle1 - The private data is stored in static variable in onCreate() method of MainActivity. Then, this static variable is accessed by a fragment after attaching to this MainActivity and then leaks it. The tool should have knowledge about lifecycle of fragments and also this fragment is not defined in xml but used in java code.

FragmentLifecycle2 - This test case tests if tool has capability to handle fragments and activities that are attached to these fragments.

ServiceLifecycle1 - This test case obtains private data to leak in onStartCommand() method of service lifecycle and then leaks it in onLowMemory() method. Analysis tool should be able to track the service lifecyle correctly.

ServiceLifecycle2 - This test case leaks the data when onStartCommand() method of MyService class is called twice. onStartCommand() method is acting as both source and sink. Analysis tool must be able to keep track of service lifecycle.

SharedPreferenceChanged1 - This test case checks if tool has capability to handle the listeners call. This case obtains private data in onCreate() method and then edit the SharedPreferences. On changing this, onSharedPreferenceChanged() method should be called which leaks the data.

■ GENERAL JAVA

Clone1 - Private user information is kept in a linked list in this test case, which is subsequently copied to create a new linked list. Then, private data is leaked using new linked list. The tool should have capability to model the clone of list.

Exceptions1 - This test case leaks the private data in try-catch block. So analysis tool should be able to handle exceptions while analysing the source code.

Exceptions2 - In this test case as well, private data is leaked through try-catch block but here exception is invoked implicitly. Analysis tool should be able to handle implicitly invoked exceptions.

Exceptions3 - This case also uses try-catch block but no data leak occurs, since the catch block that leaks data is never invoked. The analysis tool should have knowledge about which exceptions can occur and which can't occur.

Exceptions4 - This case pass the private data to exception handler as message of exception and leaks it in catch block. The analysis should be able to track flow of private data and also handle exception data.

FactoryMethods1 - This test case obtains users private data using factory functions that are present in android OS and leaks it to the log. The analysis tool should know about factory methods that gives private information of user.

Loop1 - This test case obtains private data in form of string, then convert it to character array and obfuscate it using underscore in between every character and then leaks the obfuscated string through SMS. The analysis should be able to handle obfuscations done on data.

Loop2 - Here, again private data is first obfuscated and then leaked through SMS. But here data is obfuscated in nested loop, so tool should also be able to handle such type of obfuscations.

Serialization1 - This test case obtains private data and stored it in a string initially. Then, it is passed through a series of ByteArrayOutputStream and ObjectOutputStream and then finally leaked via logs. Challenge for analysis tool is that it must be able to model serialization of data.

SourceCodeSpecific1 - This test case obtains private data in a conditional branch. Then, it leaks the data through message. Analysis tool must be able to handle standard java constructs.

StartProcessWithSecret1 - The private data is obtained initially and then leaked through a command using ProcessBuilder. To detect this leak, tool should be able to keep track of flow of private data to sink.

StaticInitialization1 - This test case uses concept of nested class. Here, one class is used to obtain the data and store it in static variable and another class leaks it using static block. The analysis tool must be able to handle static block.

StaticInitialization2 - This test case is reverse of previous one. Here, static block of a class acts as source of private information and then it is leaked through non-static code block. Challenge for analysis tool is also same as previous case.

StaticInitialization3 - Java language doesn't define the order of execution of static initializers. This case leaks data depending on runtime order of execution of code. So analysis tool should check all possible order of execution of static initializers in order to detect leak.

StringFormatter1 - This test case tries to transform the private data using StringBuffer and Formatter object and then leaks it to log. Analysis tool should be able to monitor use of StringBuffer and Formatter.

StringPatternMatching1 - This test case checks usage of pattern and matcher in java. The private data is matched against a regular expression and matched group is leaked via log.

StringToCharArray1 - The private data is initially stored in string object and then converted to character array and then again converted back to string object. Then, it is leaked to logs. So, analysis tool should be aware of the conversions done private data.

StringToOutputStream1 - The private data is written to output stream and then read it back into a string object and then leaked to logs. The analysis tool must be aware of flow of private data through stream or memory operations.

UnreachableCode - In this test case, source function obtains the private data but sink function is never called anywhere, so this case doesn't leak any data. Tool must be know that sink function is never called.

VirtualDispatch1 - This case obtains the private data in onCreate() method and stores it into static variable of class. Then, when button is clicked, its callback method clickButton() is called which leaks the data using another class. So, tool must be able to handle callbacks and invoke-virtual statements.

VirtualDispatch2 - This test uses concept of inheritance between multiple classes and try to manipulate the variable that stores the private data. In order to detect leak, analysis tool must have knowledge of inheritance concepts and overriding functions.

VirtualDispatch3 - In this test case, concept of interface is checked. Here, two classes implements an interface but one of them return the private data and other returns constant data. Only constant data is leaked.

VirtualDispatch4 - This case is same as previous one but with some more complications. The challenge for analysis tool is to deal with factory methods and should have knowledge of interface.

■ <u>**MISCELLANEOUS ANDROID-SPECIFIC**</u>

ApplicationModelling1 - This test case stores the private data in MyApplication class static data member and later leaks it in AnotherActivity via logs. To detect leaks, tool should know how to resolve explicit intent that carry private data.

DirectLeak1 - In this test case, source and sinks are placed in the single statement of onCreate() method. The tool should also able to handle such cases.

InactiveActivity - The activity that leaks the private data in this case is set to inactive in the manifest file of this test case. So no data leakage occurs. Hence tool should also use manifest file information during the analysis.

Library2 - This test case reads the private in a custom made library and leaks it through the application via message. The analysis tool should know how to handle custom libraries.

LogNoLeak - In this test case, no private data is leaked but a constant data is leaked to check if analysis tool has capability to check whether or not any private data can reach to sink or not.

Obfuscation1 - This test case doesn't guarantee to work on emulator as expected but on real device it will work as expected. This test case has its own implementation of a inbuilt system class that helps in obtaining private data of user. But on real device this custom implementation will be overridden by pre-loaded OS implementation. So this case leaks data in actual. The analysis tool must not be fooled by fake implementation of system class present in the source code.

Parcel1 - This test is created to check whether analysis tool has ability to properly model the parcel marshalling and unmarshalling.

PrivateDataLeak1 - This test case leaks username and password entered in the text field of this application after obfuscating the password and concatenate it with username in a single string. The password and username travels through a class to reach the sink. Analysis tool should be able to track the data flow of these fields and handle callbacks as well.

PrivateDataLeak2 - This test case is straight forward that leaks password entered in application to logs. The analysis should treat password field as source.

PrivateDataLeak3 - This test case checks the capability of tool to handle data flow across file system. Here private data is written into a file, then reads it back and leaks it via message.

PublicAPIField1 - This test case checks the ability of tool to correctly model the Application Programming Interface(API) classes that expose fields. Here private data flows through a API field setter method and a direct field access.

PublicAPIField2 - This test case checks tool's ability to link getter and setter methods of intent fields. Tools must be able to model implementation of getter and setter of intent fields.

■ <u>IMPLICIT FLOWS</u>

ImplicitFlow1 - Here, private data is obfuscated in two different ways and then pass onto to sink function that leaks it via logs. The tool should be able to track data flows and implicit flows.

ImplicitFlow2 - This test case leaks data based on the input entered by user. The tool should know about implicit flows and detect callbacks from xml layout file.

ImplicitFlow3 - This test case is a complicated version of previous one. Here, data travels through function and multiple leak occurs. Challenge is also same as previous test case.

ImplicitFlow4 - This test case is also like ImplicitFlow2 with use of exception handling in between data flow. Challenge is also same as previous one.

■ <u>REFLECTION</u>

Reflection1 - This test case checks if tool is able to handle reflective class instantiations or not. Using reflection, an object of BaseClass in created and then a private value is obtained and leaked using this object.

Reflection2 - This test case is little bit trickier than previous. Here, again an object is created using reflection. But this it is leaked using an abstract method declared in that class and defined in a derived class.

Reflection3 - This test case uses getter and setter methods along-with the object that is instantiated using reflection. Challenge is same as previous two.

Reflection4 - In this test case, both source and sink method are declared as abstract in the BaseClass. This case again instantiated object of this class using reflection and check tools ability to detect use of reflection.

- **THREADING**

AsyncTask1 - This test case obtains private data in onCreate() method and then use android's AsyncTask mechanism to send it to a dedicated thread. The tool must be able to handle Android's asynctask mechanism.

Executor1 - This test case also uses dedicated thread to leak the data but by using Java's Executor mechanism. The tool should be able to handle this mechanism.

JavaThread1 - This test case uses normal thread mechanism to leak the data through dedicated thread. The tool should be able to handle java threads.

JavaThread2 - This test case uses dedicated thread created by Runnable mechanism to leak the data. The tool should know how to handle Runnable mechanism.

Looper1 - This test case uses custom made thread that hosts android looper and whose handler leaks the data. The tool should handle android's looper correctly.

- **EMULATOR DETECTION**

ContentProvider1 - This test case only leaks the data when it is run on real device. The analysis should be able to avoid being detected and find another way around.

IMEI1 - This test case also tries to detect the emulator device by cutting the secret data that is exposed at a place computed by IMEI of device (real or emulator) on which app is running.

PlayStore1 - This test case differentiates between emulator and real device by checking whether playstore is installed on the device or not. It also leaks the IMEI only if it is running on real device.

Chapter 4

Static Analysis Tool

In this chapter, we discuss the methodology, features and installation steps of the tools for static analysis that are part of this work. To setup these tools, we have used steps given in github repository of tools or other internet resources.

4.1 Joint Advanced Application Defect Assessment (JAADAs)

JAADAs [Flankerhqd and Shrivastava] is a tool designed for static analysis of Android application. It can detect following types of vulnerabilities -

- API misuse
- Dataflow
- Compatibility leak

4.1.1 Methodology

This tool uses potential of Soot [Vallée-Rai et al.]. Soot performs static analysis for Android application of type intra-procedure and inter-procedure. JAADAs is written in Java and Scala. To perform analysis on multidex, it may integrate them into one and then analyze them simultaneously. JAADAs analysis potential depends on Soot's source-sink file. You can also add your own configuration in groovy config file. It has two modes -

- **Fast Analysis** - It is designed for batch analysis. This mode reduces time consumption by disabling the inter-procedure analysis. This mode is enough to perform a basic security for an Android app.

  ```
  $ java −jar <jar built during setup>
     vulnanalysis\ −f path−to−app.apk −p <./
     Android/Sdk/platforms> −c <config file
     generated during setup> − −fastanalysis
  ```

- **Full Analysis** - This mode is default in JAADAs. This runs a full analysis of the Android application including a full data flow and procedural analysis. This mode consumes more amount of resources during its execution.

  ```
  $ java −jar <jar built during setup> −f
     vulnanalysis path−to−app.apk −p <./Android/
     Sdk/platforms> −c <config file generated
     during setup>
  ```

Output after successful analysis is stored as JSON structure. It conatins a list of vulnerabilities with the description of source. Now, we will look at steps to setup and use the tool.

4.1.2 Setup

This tool mainly requires JDK >= 1.8, Scala >= 2.11 and Android platforms tools to be pre-installed in your system. The platform tools version should be equal to the API version defined in the Android application which is being analysed. If the required API version is not present in the directory, the analysis will fail.
To build the JAADAs, simply clone the Github repository [Flankerhqd and Shrivastava] and run following command inside the cloned folder -

```
$ gradle fatJar
```

This will generate a single-bundled jar and place it in ./jade/build directory. Now using this jar, you can run the tool in any of the two modes given above. We have tested JAADAs on Droidbench benchmark (Refer chap-6 for results).

4.2 Flowdroid

Flowdroid [Arzt et al.] is used to find Android applications data-flows statically. It is used to determine flaws that occured due to human error or malevolent intent.

4.2.1 Methodology

Comprehensive modelling of Android's lifecycle is made. As a result, the analysis can effectively manage callbacks triggered by the Android approach. Reduced false alarm rates are made possible by the analysis's use of the context-flow field and object sensitivity. A comprehensive lifecycle model contains many entry points, components and callbacks that execute asynchronously, and high object sensitivity.

- **Handling more than one points of entry in Android application -** Android applications have multiple entry points such as methods which are implicitly called by Android framework since it does not have main method. Due to this reason, construction of call graph can not be started by simply inspecting a predefined main method. As a solution to this problem, a pseudo main method simulating the lifecycle is introduced by Flowdroid. Flow of an Android app from one component to another may depend on user input. So, Flowdroid assumes that all components inside an app can run in any sequence. Now, if we consider all possible paths for any app, then its analysis will be very expensive. For this purpose, Flowdroid relies on IFDS problems(Interprocedural, Finite, Distibutive, Subset problems)[Reps et al.] that defines an analysis framework analyse merge point of control-flow graph for results and this is not a path sensitive framework. This enables Flowdroid to generate and analyze a pseudo main method without having to take every potential path, allowing for the possibility of any order of component lifecycles and callbacks.

- **Handling Callbacks -** Next thing to take care of is ”Callbacks”. Hackers can register their own callbacks by overriding Android infrastructure methods. Native code can also call these methods. Flowdroid detects such overridden methods and treats them like standard callback handlers. Callbacks can also be invoked in any order, that’s why Flowdroid makes the assumption that any order of callbacks can be used. Callbacks, however, can only be made when the parent component is active. Thus, components are linked to the callbacks in Flowdroid. Starting with lifecycle methods, Flowdroid computes a call-graph for each component in order to find registered callbacks. The generated call-graph is then examined to identify calls to Android system methods that accept callback interfaces as a formal parameter type.

- **Handling High Object-Sensitivity -** Implementing high object sensitivity to adequately address aliasing is another major difficulty. An on-demand backward-alias and a forward-taint-analysis are used by Flowdroid to determine the taint at sink. In taint analysis, forward analysis and backward analysis both traverse access paths. An access path is made by combining local variable and fields. Everytime any tainted value is allocated to memory location, Flowdroid goes bakcwards to look for aliases of the target variables in order to taint them too.

Now, lets see the steps to setup the Flowdroid and use it.

4.2.2 Setup

In order to use FlowDroid, you can either build using Maven or you can use already built Jar files present in FlowDroid repository[1]. To use Flowdroid using JAR files,

[1]https://github.com/secure-software-engineering/FlowDroid

use following command -

```
$ java -jar [path_to_flowdroid_dir]/soot-infoflow-cmd/
    target/soot-infoflow-cmd-jar-with-dependencies.jar -a
    path/to/app.apk -p Android_JAR_file -s SourceSinkfile
```

Here, I have used a full list of sources and sinks[2] for testing. This prints the results on stdout (See section-6 for results).

4.3 AmanDroid

Amandroid [Wei et al.] is another static analysis tool to test security of Android apps. For each input APK, it runs dataflow and data-dependency analysis. It can also integrate the information of component level into information of app level so that it can also perform inter-app analysis. It is inspired from FlowDroid. AmanDroid determines points-to data for all objects and their fields at each program point and calling context.
Using this point-to information, it constructs a inter-procedural control flow graph(ICFG) for each component for input APK with a high precision [Nielson et al.]. This ICFG is both flow and context sensitive. Now for each component, using ICFG of that component it also constructs Data Flow Graph(DFG). After this using Data Dependence Graph(DDG) of each component, AmanDroid constructs DDG for each component. It also makes Summary Tables(ST) that contains list of communication between various components over various channels like Intent, RPC, static fields etc.

4.3.1 Methodology

- We feed an Android APK as input to AmanDroid. Then, it decompiles the input APK and and recover a dex file from it. Then, it converts dex file into some intermediate representation(IR) that can be used by AmanDroid for further analysis.

[2]https://github.com/secure-software-engineering/SuSi/tree/develop/SourceSinkLists/Android%204.2/SourcesSinks

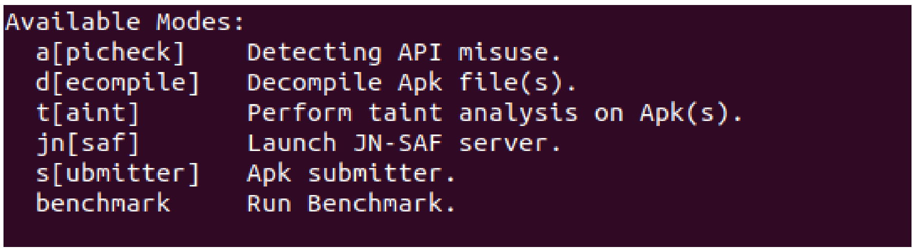
- Now, using this IR and other metadata like manifest and resource file, it builds an environment imitates the interaction of Android OS with the application. Using this, AmanDroid builds DFG. This DFG combinedly represents control flow graph (CFG) and points-to information of objects.
- Then, it builds DDG and ST for each component. Afterwards, a DDG for the app-level is made by integrating DDG of all components.
- Using this DFGs and DDGs, AmanDroid can find possible vulnerabilities of following type - Data Leak Detection, Data Injection Detection, API Misuse Detection etc.

4.3.2 Setup

Download the JAR file given at this link. Then, use following command to check what all analysis can be performed with Amandroid.

```
$ java -jar argus-saf-3.2.1-SNAPSHOT-assembly.jar
```

Its output will be -



```
Available Modes:
  a[picheck]    Detecting API misuse.
  d[ecompile]   Decompile Apk file(s).
  t[aint]       Perform taint analysis on Apk(s).
  jn[saf]       Launch JN-SAF server.
  s[ubmitter]   Apk submitter.
  benchmark     Run Benchmark.
```

Now to check options available for taint analysis you need to run following command -

```
$ java -jar argus-saf-3.2.1-SNAPSHOT-assembly.jar t
```

To use Amandroid to perform staic analysis on Android app, use following command-

```
$ java -jar argus-saf-3.2.1-SNAPSHOT-assembly.jar t -o
    output/path path/to/droidbench
```

This will generate the report and store the output in output-path given. To see performance of this tool on Droidbench benchmark see chap-6.

4.4 IccTA

IccTA [Li et al.] is a static analyzer to detect privacy leaks between across various component of Android application. This tool is mainly designed for Inter-Component Communication(ICC) in Android applications. It fixes the ICC issue by resolving the code discontinuities in Android applications. There are mainly four types of components in Android appps : Activities presents the user interface, Broadcast Receivers listens to incoming calls and messages, Content Providers manages the data storage layer in an application and services performs the task in background that aren't visible to user. Since services runs in background, so it opens up ways to perform malicious actions. So, IccTA targets all communication happened among these components. Android gives in-built methods (called ICC methods) for doing inter-component communication.

4.4.1 Methodology

Now, lets take a look at IccTA methodology -

- First of all, IccTA converts the input into Dalvik bytecode and then using Dexpler, Dalvik bytecode is converted to Jimple representation(Soot's IR). Soot is populary used to analyze Java Based Apps.

- Then, IccTA retrieves the ICC links and stores links and other data (like ICC call parameters etc) into the DB. Now, using the Inter-Component Communication links, it modifies the Jimple representation generated earlier to perform dataflow analysis between various Android components.

- Now using Flowdroid, [Arzt et al.] IccTA constructs a full control flow graph(CFG) of the complete Android app. This facilitates the movement of intent values between various Android components and also provides great accuracy in data-flow analysis.

- Then, it stores the tainted path into DB. Storing the results into the database provides reusability of the results in future. This gives the high precision on Droidbench and ICC-Bench.

Now, lets see the steps to install and use this tool. Later we will test the tool performance on Droidbench benchmark (see chap-6).

4.4.2 Setup

First of all, you need working MySQL server and the client. It should also have one user who is able to create a database and importing table schemas in database. Now, configuring IccTA invloves multiple steps. We will look at them one by one :

- **Database Creation -** Run following commands :

  ```
  $ git clone https://github.com/lilicoding/soot-
      infoflow-android-iccta.git

  $ cd soot-infoflow-android-iccta
  ```

  Now, We will create database and import schema in it -

  ```
  $ mysql -u user -p -e "create␣database␣cc"

  $ mysql -u user -p cc < res/schema
  ```

  For above command, you might get some error. To resolve it, find below lines code block in res/schema :

  ```
  CREATE TABLE `IFMimeTypes` (
      `id` int NOT NULL AUTO_INCREMENT,
      `filter_id` int NOT NULL,
      `type` varchar(512) NOT NULL,
      `subtype` varchar(512) NOT NULL,
      PRIMARY KEY (`id`),
  ```

```
        FOREIGN KEY (`filter_id`) REFERENCES
            IntentFilters(`id`) ON DELETE CASCADE,
        INDEX `type_idx` (`type`, `subtype`)
    );
```

And replace this block by following block :

```
    CREATE TABLE `IFMimeTypes` (
        `id` int NOT NULL AUTO_INCREMENT,
        `filter_id` int NOT NULL,
        `type` varchar(512) NOT NULL,
        `subtype` varchar(512) NOT NULL,
        PRIMARY KEY (`id`),
        FOREIGN KEY (`filter_id`) REFERENCES
            IntentFilters(`id`) ON DELETE CASCADE,
        INDEX `type_idx` (`type`, `subtype`)
    ) ENGINE=InnoDB  CHARACTER SET utf8;
```

Now, try agin the last command and it will work fine.

- **Configuration Files -** Open the res/iccta.properties file and make sure that it contains iccProvider=ic3. Now, modify the res/jdbc.xml file with values that you are using -

  <name>name of the DB (in our case 'cc') </name>

  <username> DB user </username>

  <password> DB password </password>

  Similarly, change these values in release/res/jdbc.xml. Now, change following value of release/res/iccta.properties according to your system -

  android_jars=path/to_your/android/platforms

- **Retarget APK -** This includes conversion of .dex files into .class files. For this, IccTA uses dare tool which can be setup as follows:

```
$ wget https://github.com/JordanSamhi/Tools/raw/
    master/dare.zip

$ unzip dare.zip

$ cd dare

$ ./dare -d path/to/dare/results path/to/app.apk

$ cd ..
```

- **ICC model extraction -** First, make cc.properties and write following values in it -

  ```
  user=test
  password=password
  characterEncoding=ISO-8859-1
  useUnicode=true
  ```

  Now, to get latest IC3 and setup, follow below commands -

  ```
  $ wget https://github.com/JordanSamhi/Tools/raw/
      master/ic3.jar

  $ java -jar ic3.jar -a path/to/app.apk -cp path/
      to/android/platforms -db cc.properties
  ```

  The results will be saved in cc DB and those will be used by IccTA.

- **Execution of IccTA -** Follow below commands to run IccTA:

  ```
  $ wget https://github.com/JordanSamhi/Tools/raw/
      master/iccta.jar

  $ java -jar iccta.jar path/to/app.apk path/to/
      android.jar
  ```

Results will be displayed on stdout. To check performance of this tool on Droidbench benchmark see chap-6.

4.5 Quick Android Review Kit (QARK)

QARK [Linkedin] is another open-source tool to improve Android application security. Core task of QARK is to perform static code analysis and find threats to security for Java based Android apps. But it also has few features that help with dynamic analysis too. It prepares a report about threats to Android app security that contains issues and its description so as to help developers and security professionals. It can also create a dynamically built testing application (i.e ready to use Android APK) that demonstrates the potential issues it discovers.
QARK has mainly following features (but not limited to) -

- It is very simple to setup(see 4.5.2 for setup) and easy to use command line interface.
- It just need an APK as input and provides security issues in the app.
- It provides robust report after analysing the Android app consisting of potential risks and link to detailed description of those issues.
- It can also be easily integrated into organisation's Software Development Life-Cycle.
- It can also parse AndroidManifest.xml file to locate issues with permission.
- Source to sink mapping : following potentially tainted flows through the Java source code.

4.5.1 Methodology

QARK is a completely automated tool that starts with the retrieval of APK, then decompiles it and extract the human readable manifest file. Now, sometimes there may be an issue of not able to generate the actual source code while decompiling. To overcome this problem, QARK uses multiple reverse engineering tool and merges the result of all of them to create the code that is closest to actual source code. Developers of QARK are working actively to improve accuracy (minimize false positives and negatives) and the testing APK it creates. QARK can find following type of vulnerabilities -

- Accidently exported components and exported components that are not properly safeguarded
- Detects use of sticky intents and intents that are susceptible to intruding or intercepting.
- Detects applications that support older API version that have known vulnerabilities, permits backup and debuggable.
- It also finds pending intents that are produces insecurely and data leakage activity.
- Detects if app creates files that can be read or write by anyone
- Use of encryption technique that is either insufficient or flawed.

Now, lets look at its installation and usage steps. Then, we will assess the tool performance on Droidbench benchmark.

4.5.2 Setup

There are two ways to install QARK [Linkedin] -

1. **Using pip :** Run below command. Here, - -user is required only if not using virtualenv ->

```
$ pip install −−user qark

$ qark −−help
```

2. Using **Github repository :** Run below command as is sequentially ->

```
$ git clone https://github.com/linkedin/qark

$ cd qark

$ pip install -r requirements.txt

$ pip install . --user

$ qark --help
```

Now you are ready to test your Android APK with qark. Use below command -

```
$ qark --apk path/to/app.apk
```

You can also explore other options using - -help option. We have tested QARK on Droidbench benchmark (See section-6 for results).

4.6 AndroBugs

Androbugs [Lin] is another open-source tool written in Python that performs static analysis after taking Android APK as input and finding valid security vulnerabilities in an Android App. It can also be used for massive analysis and is easily extendable to add new specs and vulnerability vectors. It can also be used to check if the code is missing best practices. It also checks for dangerous shell commands (e.g "su"), and for app repackaging hacking. AndroBugs framework uses AndroGuard decompiler but Androguard doesn't have a security vulnerability checking feature.

4.6.1 Methodology

There are three important components/engines introduced in AndroBugs framework -

1. **Static DVM Engine** - It is the core of AndroBugs Framework. Just like DVM or ART in Android operating system, the Static DVM Engine runs the

Bytecode statically and partially. This means it doesn't need to run app on an Android phone and doesn't even need to run the code completely. Like DVM or ART, the static DVM Engine maintains a simple register table and also removes useless instructions (opcode).

2. **Efficient String Search Engine** - String search in code using "regex matching" is quite slow and inefficient. Moreover, it is needed to find path sources where a string is showing up rather than just finding whether the string exists or not. So, the author designed a new Efficient String Search Engine that makes string search faster and also gives its location. In Android, strings are referenced by their index position (offset). The engine then compares the string index in the code.

3. **Filtering Engine** - It is used to filter some libraries/packages (like com.parse, com.facebook, com.tapjoy). This is needed because some Android libraries may never fix their vulnerabilites and they are used by many applications. Moreover, some libraries are not vulnerable or their impact is very limited.

It uses MongoDB(NoSQL DB) for massive analysis. The analysis result will be stored by vectorID and PackageName for future usage. In massive analysis, one can keep all Android apps to be tested in a folder and then provide that folder path as input. To check performance of this on tool Droicbench see chap-6.

4.6.2 Setup

There are some prerequisites before going into the setup details of this tool -

- Python3
- MongoDB installation

MongoDB installation is required only when you want to use massive analysis feature of Androbugs. Now, after satisfying above prerequisites, clone the Androbug Github repository ->

```
$ git clone https://github.com/androbugs2/androbugs2.git

$ cd androbugs2

$ python3 -m venv venv

$ source venv/bin/activate

$ pip -r requirements.txt
```

Now, you are all set to test your application with Androbugs using below command -

```
$ python androbugs.py -f path/to/app.apk
```

The results will be stored in Reports folder of Androbugs repository.
To perform massive analysis, first of all make sure that MongoDB server is up and running using below command -

```
$ ps -eaf | grep mongo
```

Now, run below command to perform massive analysis -

```
$ python AndroBugs_MassiveAnalysis.py -b [Analysis Date]
    -t [Analysis Tag] -d [APKs input directory] -o [Report
      output directory]
```

For example -

```
$ python AndroBugs_MassiveAnalysis.py -b 20062020 -t
    BlackHat -d path/to/APKDataset/ -o path/to/output_dir
```

If no output directory is mentioned then, results will be stored in Massive_analysis_report folder of Androbugs repository. We have tested this tool with Droidbench dataset (Refer section-6 for results).

4.7 Androwarn

Androwarn [Debize] is another open-source static analysis tool for malicious Android application that takes Android apks as input file and performs reverse engineering on it and analyse it statically to find out any malicious activity that application can perform when run on mobile device. The Smali code (Android application Dalvik's byte code) along-with Androguard library is used to do static analysis and detect vulnerabilities in application. This tool identifies following information and generate a report for each application fed to it -

- Information transfer of mobile device identifiers - IMEI, IMSI, MCC, name of network operator etc.
- Information transfer of mobile settings - Application versions, Application settings, how frequent apps are used etc.
- User's location information leaks - GPS/Wifi location.
- Information leakage of network information - MAC address of bluetooth, Wifi password etc.
- Misuse of services - sending private data via sms.
- Leak of audio/video data - recording calls, mobile screen capture.
- PIM data leakage - messages, contact details, calendar informaation etc.
- Accessing File system - Some application may access filesystem without users permission.
- Modification of PIM data - delete random messages, calendar events etc.
- DoS atack - Event like sudden reboot/shutdown, blocking of notification of some particular app, deleting the file etc.

Androwarn performs search of all these kinds on Android applications. Then, it generates a report of its findings and developer can use the report to make his app more secured. Now, lets see its installation and usage instructions.

4.7.1 Setup

The setup of Androwarn is very easy. It works with python3 or python2.7. It also requires installation of Androguard, Argparse, Jinja2 and Play_scraper. But you don't need to install them one by one. Lets walk through the installation steps of Androwarn -
Clone the Androwarn repository from Github using below command -

```
$ git clone https://github.com/maaaaz/androwarn.git
```

Now, its time to install dependencies which are accumulated in requirements.txt file.

```
$ cd androwarn
$ pip install -r requirements.txt
```

If installation goes without any error. Then, you are good to go. You can check a variety of options available in Androwarn using below command -

```
$ python androwarn.py -h
```

Then, choose those options according to your requirement to test the Android application you want. A simple example is given below -

```
$ python androwarn.py -i <target apk file> -r <desired
    report format> -v <desired report level>
```

Here, -i will take the application to be tested as input, -r takes report format i.e HTML or JSON report format, -v takes 1(beginner level), 2(advanced level) or 3(expert level) as input to generate report. Example -

```
$ python androwarn.py -i test.apk -r html -v 1
```

This command will generate the report in html format in beginner level format. This tool has also been tested on DroidBench test suite (Refer to 6 for results).

4.8 Mariana-Trench

Mariana-Trench(MT) [Facebook] is another open-source tool that analyzes Android and Java app security in depth. This was built when security engineers and software engineers working at Facebook came together to collaborate. They train mariana-trench to analyze the program and track data flows. Data-flow analysis is a powerful method in security and privacy analysis because it can be find if data is flowing into the places where it shouldn't. It is used to scan large mobile codebases and find vulnerabilities before releasing the app into the production environment.

4.8.1 Methodology

Lets take a look at its methodology -

- High level style of working of MT is closely similar to Zoncolan and Pysa. Zoncolan is used for hacking and Pysa is used to find security issues in Python applications.

- As MT mainly focuses on data flow analysis, so in MT a data flow can be described as -
  Example - Private data should not be logged by an application. So, data flow is -
  Source - This is also called point of origin where application assigns a private value to some string or other datatype.
  Sink - This is the destination from where data is leaked. This can be ”Log.w” for above example.
  Various types of sources and sinks can be found in big codebases. MT is customizable. So in order to customize MT according to your requirements, you can add rules rules to tell MT to show us some particular data flows. Example - If a developer wants to check his application about redirections of

intent that allows hackers to manipulate sensitive data, then we can add a rule to show all data flows from ”user-controlled” sources to an ”intent-redirection” sink.

It is possible to tweak models and rules according to your requirement in MT. To learn more on this click here. Now, we will see how to install MT on your system and how to use it.

4.8.2 Setup

Mariana can be used on MacOS or Debian flavoured Linux. We will mainly talk about setup on Debian system. Before start installing MT, check if your system has recent Python installed or not. If not, you can install it using apt-get. In addition to above requirement, setup steps assumes that there is Android SDK installed and $Android_SDK is referring to its location.
For further installation steps, we will work inside Python virtual environment. This can be done using following steps -

```
$ python3 -m venv ~/.venvs/mariana-trench
$ source ~/.venvs/mariana-trench
```

The name of virtual environment (i.e mariana-trench, in our case) will be visible before your shell prompt that signifies that you have entered your virtual environment.
Now, we will start with main installation steps of MT which is very simple. You just need to run below command in order to install MT.

```
$ pip install mariana-trench
```

If the above command runs successfully. Then you can move on to the next step, otherwise, find the error reason and install the required dependency to run the above command successfully.

```
$ git clone https://github.com/facebook/mariana-trench
$ cd mariana-trench
```

Now, you are ready to use MT. Below command shows an example usage of MT. But you can always refer to help section to explore others options.

```
$ mariana-trench --system-jar-configuration-path <path
    to android.jar file> --model-generator-configuration-
    paths ./configuration/default\_generator\_config.json
   --lifecycles-paths ./configuration/lifecycles.json -
   -rules-paths ./configuration/rules.json --apk-path <
    path to your apk> --source-root-directory <path to
    src code of apk> --model-generator-search-paths ./
    configuartion/model-generators/
```

This will output the set of specifications of each method of the application. You can explore other options in help section. To check performance of this tool on DroidBench test suite refer to chap - 6.

4.9 APKLeaks

APKLeaks [Dwi] is an open-source static analysis tool developed by Indonesian based security engineer Dwi siswanto. It is mainly used to scan Android APK files and detect URIs, endpoints and hidden secrets in the provided APK file. This is also used by FirmwareDroid which is a backend solution for Android firmware analysis. It is fairly simple to install and use. We will now see how to install APKLeaks -

4.9.1 Setup

APKLeaks uses Jadx [Skylot] to decompile the provided APK. There are multiple ways to install APKLeaks on your system. Lets see them one by one -

- From PyPi -

  Just run below command and you are ready to use APKLeaks after successful execution of below command -

```
$ pip3 install apkleaks
```

- From Source -

  This method includes three easy steps as follows -

  ```
  $ git clone https://github.com/dwisiswanto0/
      apkleaks
  $ cd apkleaks/
  $ pip3 install -r requirements.txt
  ```

  After successful completion of above commands, you can use APKLeaks.

- From Docker -

  Just run below command simply -

  ```
  $ docker pull dwisiswant0/apkleaks:latest
  ```

Now, you can verify if its successfully installed or not using below command -

```
$ apkleaks --help
```

Above command should output help section of APKLeaks tool. Now, to use apkleaks use below command based on the method you used to install APKLeaks -
#if installed using pip

```
$ apkleaks -f <path to apk file>
```

#if installed using source

```
$ python3 apkleaks.py -f <path to apk>
```

#if installed using docker

```
$ docker run -it --rm -v /tmp:/tmp dwisiswant0/apkleaks:
    latest -f /tmp/file.apk
```

This will generate the output file in Json format. To check performance of APKLeaks on DroidBench test suite refer to chap - 6.

4.10 DIDFAIL

DIDFAIL [Dwivedi et al.] stands for Droid Intent Data Flow Analysis for Information Leakage. It is also an static analysis tool that finds any information leaks in the application APK provided to it. Learning about secured coding practices and helping developers to develop secured application is possible due to DidFail's methodology.

4.10.1 Methodology

- DidFail uses two phase technique to perform the analysis. DidFail takes APK file as input and APK Transformer transforms the original .apk file to a newly modified version of .apk file that is done using soot. The matching intents of outputs of Flowdroid and Epicc are also enabled using APK Transformer.

- Then, this modified code code APK file is sent to FlowDroid, Epicc and Dare which are intermediate tools used in DidFail.

- The Dare finds the Java class files using modified APK code.

- Then, modified APK code and output of Dare is fed into Epicc that outputs the unique intent IDs.

- DidFail modifies source code of Flowdroid to meet its requirements. In Phase-1, Didfail takes the 'sources' as the point where information flows-in and 'sinks' as the point where information flows-out.

- As a result, they have added onActivityResult() and setResult() method as source and sink respectively. Didfail also takes care of putExtra() method to insert unique intent ID.

- Phase-1 generates 3 output file(one for each tool Flowdroid, Epicc, Dare) for each provided APK files.

- Phase-2 takes these 3 files as input and it provides the information of connection between source and sink. This information also includes intents if they are present in in data flow.

Lets see the steps to install and use DidFail.

4.10.2 Setup

These setup steps were performed on Ubuntu12.04 machine. This tool requires Java-JDK7 and Flowdroid should be already built in order to move ahead. To build Flowdroid click here. After building Flowdroid, follow further steps carefully -

```
$ sudo apt-get install gcc-4.6-multilib lib32stdc++6
  zlib1g-dev:i386
$ export didfail=~/did-fail
$ mkdir $did-fail
$ cd $did-fail
```

Now, download dare, epicc, Android platform JARs and toy apps that were used to test Didfail:

```
$ wget -nc http://www.cs.cmu.edu/~wklieber/didfail/epicc
  -0.1.tgz
$ wget -nc https://github.com/dare-android/
  platform_dalvik/releases/download/dare-1.1.0/dare
  -1.1.0-linux.tgz
$ wget -nc http://www.cs.cmu.edu/~wklieber/didfail/
  platform-16.zip
$ wget -nc http://www.cs.cmu.edu/~wklieber/didfail/
  platform-19.zip
$ wget -nc http://www.cs.cmu.edu/~wklieber/didfail/
  toyapps-2014-04-28.zip
```

Now, unpack all above downloaded files -

```
$ mkdir epicc
$ tar xzf epicc-0.1.tgz -C epicc
$ tar xzf dare-1.1.0-linux.tgz
$ unzip platform-16.zip
$ unzip platform-19.zip
$ unzip -q toyapps-2014-04-28.zip
```

Now, clone the Didfail repository -

```
git clone https://bitbucket.org/wklieber/didfail.git cert
$ cd cert
```

#if you want the latest version of this branch

```
$ git checkout service-additions
```

#if you want release snapshot

```
$ git checkout 2781780c9971f0ec5c998956b14af81196a9879f
$ cd ..
$ cp cert/paths.distrib.sh cert/paths.local.sh
```

Now, based on your paths of didfail and workspace directory modify the paths.local.sh accordingly. You may need to modify following lines -

- ”export didfail=” where you have performed above steps
- ”export wkspc=” to the point where soot, jasmin, heros and soot-infoflow* directories are present.

Now, compile the APK Transformer -

```
$ cd $didfail/cert
$ source paths.local.sh
$ cd transformApk
$ make
$ cd $didfail
```

Then, run analyzer on the toy apps -

```
$ $didfail/cert/run-didfail.sh $didfail/toyapps/*.apk
```

This will perform both phases of DidFail and generate the report. To check the performance of this tool on DroidBench test suite refer to chap-6.

4.11 SUPER Android Analyzer

- SUPER [SUPERAndroidAnalyzer] can be used with Windows, Linux and MacOS and it is a command line tool.
- SUPER analyzes .apk files to detect vulnerabilities. It first decompiles the given APK and detect leaks/malware by using series of rules on the decompiled code.
- It is also extensible and put those functionalities first that can be used by businesses working in Android Analysis.
- To make it extensible, rules.json file contains all rules that are used in analysis. This enables a tester to make his own rules to test the application.
- Almost all the tools are developed in Java or Python, but SUPER uses Rust as programming language. Rust gives many utilities to work with regex, files etc and developed by Mozilla Foundation. Use of Rust programming enables to create the tool that does not need JIT or JVM compilers for execution. Use of Rust has other advantages also like segmentation fault, stack overflows etc are directly not possible. Rust also enables developer to automate the analysis in high volume and to perform optimised analysis.
- After performing analysis, it saves report in Json or Html format.

Now, lets see the installation steps of this tool and then test in on DroidBench test suite.

4.11.1 Setup

This tool has been tested on Ubuntu16.04 machine. There is nothing much to do in order to install SUPER. click here to download .deb file of SUPER.Use the downloaded file to install SUPER. Run below command to install -

```
$ dpkg −i <path to super.deb file>
```

You can verify the installation by running below command -

```
$ super−analyzer −h
```

An example usage -

```
$ super−analyzer <path to .apk file> − −results <path to
    store the result> − −html
```

You can also mention the to be Json using ”- -json” flag in place of ”- -html”. To see performance of this tool on DroidBench suite refer to chap-6.

4.12 MobileAudit

- MobileAudit [Mpast] is web application based tool developed on Django framework.
- It is a static analysi tool that helps in finding vulnerabilities/malware in Android APKs. Use of Django framework in developing provides advantages like rapid development, scalability, security etc.
- This tool is made of three components mainly - Frontend, Backend and External. Each of these components contains a number of subcomponents.
- Frontend component is as usual UI of web application dashboard.
- Backend contains - RabbitMQ, PostgreSQL, nginx and worker.

- External contains - ”DefectDojo” that enables to upload the findings to defect manager, ”VirusTotal” scan the APK and extracts all its information, ”malware database and maltrail” checks in the database if there are URLs in the APK that are related with malware.

- For each application scan, it provides following information - application info, security info, Static Application Security Testing(SAST) findings, strings, databases, files, virustotal info, certificate info, components.

- Other main features of report generated by this tool is - you can edit false positives from generated report, findings are categorized that follows CWE standards and also includes mobile top 10 risks. Results can also be exported to pdf.

Now, lets see how to do setup for mobileaudit and use it.

4.12.1 Setup

This tool can be easily installed using docker image. Just follow below steps -

```
$ git clone https://github.com/mpast/mobileAudit.git
$ cd mobileAudit
$ docker-compose build
```

After successful completion of above commands. Now, to use this tool, run below command and navigate to : `http://localhost:8888/` to access the tool dashboard and scan your application -

```
$ docker-compose up
```

Now, just upload your application on the dashboard and check the findings of this tool. To check performance of this tool on Droidbench test suite refer to chap-6.

4.13 ReverseAPK

ReverseAPK [1N3] is used to perform static analysis on Android application APK file. Given APK file, it analyzes by performing out reverse engineering on Android APK. This tool displays every file that has been extracted for quick referencing. Like all other analysis tools, it performs decompilation of APK files to Java and Smali formats. Its features also include examining the AndroidManifest file to find the permission required and other flaws. After examining the AndroidManifest file, it analyzes the decompiled code statically for common vulnerabilities and malware such as -

- If application extracts device information like IMEI, IMSI etc.
- If application uses Intents to perform some task.
- Execution of some commands.
- Checks if application logs the data.
- Analyze the content providers information.
- Checks if application uses secrets that are hardcoded, fixed URLs, insecure cryptography.
- Performs the analysis to check references to services, to any file or SSL references.

Now, lets see its installation and usage.

4.13.1 Setup

Its installation is very easy. Just follow below sequence of command -

```
$ git clone https://github.com/1N3/ReverseAPK.git
$ cd ReverseAPK
```

Now, to install ReverseAPK, just run below and it is ready to use -

```
$ ./install
```

After successful completion of above commands. Use below command to use it and it will generate the report for input application analysis -

```
$ reverse-apk <path to apk>
```

This will generate a .txt report. To see performance of this tool on Droidbench see chap-6.

4.14 RiskInDroid

RiskInDroid [Merlo and Georgiu] means Risk Index for Android. This tool's main motive is to do quantitative risk analysis on Android mobile application. It is written in Java that is used to find what permission an application requires and Python that is used to evaluate the risks percentage based on permissions of app. To evaluate the risk percentage, it uses Scikit-learn Python library. RiskInDroid uses following Scikit-learn models -

- Support Vector Machines (SVM)
- Multinomial Naive Bayes (MNB)
- Logistic Regression (LR)
- Gradient Boosting (GB)

4.14.1 Methodology

A thorough analysis is done on over 112K application samples and 6K malware samples. This analysis shows that RiskInDroid works much better in terms of precision and reliability than probabilistic techniques. This tool finds four types of Android Permissions(APs) in each app 'A' -

- Declared Android Permission (DAP_A) : Permissions that are declared in Manifest file.
- Exploited Android Permission (EAP_A) : Permissions that the application code really makes use of.

- Ghost Android Permission (GAP_A) : Permissions that are not declared in Manifest file but application code tries to take their advantage.
- Useless Android Permission (UAP_A) : Permission that are declared in manifest file but they are not used in application code.

The author defines a notion of Dynamic impacts that enables the consideration of a statistically important dataset's properties while calculating a probabilistic risk index value. But before it start computing dynamic impacts, it needs to do statistical analysis in advance. To do statistical analysis on Android Permissions, author built a Permission Checker(PC) tool. This PC tool takes app A as input and gives statistics of each type of permission as output. DAP_A can be directly found out from AndroidManifest file. Then, GAP_A, EAP_A and UAP_A are deduced through static analysis of the app. The accuracy of present probabilistic risk indexes, according to the author, might be improved by risk index value computation based on the distribution of Android Permissions on malware and programmes in the dataset. To compute dynamic impacts, following expression can be used -

$$I(p_i|S) = \frac{P(p_i|M,S)}{P(p_i|A,S)}$$

Here, $p(p_i|M,S)$ shows the likelihood that malware requires p_i in the set S. $P(p_i|A,S)$ shows the likelihood that an application requires p_i in the same set S. This method results in increase value of impact as malware required more Android Permissions than apps and vice-versa. This tool uses concept of DroidRisk [Wang et al.] and since DroidRisk considers only DAP, so dynamic impacts is also computed only for DAP.This is the high level methodology of this tool. Lets see how to use this tool.

4.14.2 Setup

We built this tool on a Linux machine using docker. It can also be built from the source but here we will see the docker. Follow below steps carefully -

```
#To download the docker image
  $ docker pull claudiugeorgiu/riskindroid
```

#cmd to give above tool a shorter name

```
$ docker tag claudiugerorgiu/riskindroid riskindroid
```

Now, to install the tool, run below command - #Remember to run this cmd in Riskindroid directory

```
$ docker build –t riskindroid
```

After successful completion of above command, your tool is successfully installed and ready to use using below command -

```
$ docker run – –rm –p 8080:80 riskindroid
```

Now, navigate to `http://localhost:8080/` to use RiskInDroid. To check performance of this tool on DroidBench test suite refer to chap-6.

4.15 AppSweep

AppSweep [Guardsquare] is a mobile-application security testing application that is developed by GuardSquare using open-source ProGuard technology. With the help of this free tool, developers can prioritize mobile security when creating new apps. Making use of Appsweep enables you to find and address security flaws in your programming and dependencies and provides you with actionable insights and recommendations that will aid in the development of more secure mobile apps.
Appsweep utilizes the same principles as your IDE, explore the application package and class hierarchy to learn more about the scan findings. Your code's dependencies are recognised by AppSweep. It uses the mapping files you've provided to deobfuscate names, display package hierarchies and more. Apply Appsweep to you pipelines and advance your understanding of mobile security and gain practical experience. The first version of Appsweep was developed in aug, 2021. After that its latest version was launched in sept, 2022. In latest release they have added manu new features -

- Detects user input flowing into class loaders or interpreters.

- Detects sensitive user data being written to disk.
- Detects sensitive user data being written to IPC mechanism.
- Detects potential misconfiguration of AES encryption modes.
- Detects insecure paddings for RSA ciphers.
- Detects insecure passwords for the certain keystores.
- Detects missing tapjacking protection in .aar files.

There are many more features to be added to this list but these are some key highlights of this tool. Lets see how to use this tool.

4.15.1 Setup

There is nothing to do in setting up the tool. You can use this tool as web application by just clicking here. Then, you can just drag and drop your application to be tested on the dashboard and it will analyze and generate report of all the findings in very structured manner. To check performance of this tool on DroidBench, see chap-6.

4.16 Pithus-Bazaar

Pithus-Bazaar [Pithus] is a free and open-source mobile threat analysis platform for developer, researcher or security professionals. This tool performs only static analysis. The solution to the explosive rise of mobile dangers is Pithus. When it comes to mobile security, malicious apps, phoney apps and data laundering are the biggest dangers. Their identification and analysis ought to be public knowledge not the exclusive domain of a commercial enterprise.

Contrary to certain expensive commercial tools, pithus is a totally open platform that is supported and updated by the community. Threats like ongoing tracking and data smuggling are made feasible by the complete lack of transparency and ignorance surrounding, the types of data that are collected and how they are used. Pithus promotes transparency with well-organized concise reports. These reports are simple to produce and can be used by developers, researchers or any other technical group to better comprehend the threat landscape.This tool collectively uses a number of well-known tools to perform analysis on the Android application - APKiD, ssdeep, Dexofuzzy, Quark-engine, Androguard, MobSF, Exodus-core. General features of this tool are -

- It performs analysis of permissions that application requires, identifies the application's different entry points.
- Retrieves information on signing certificates. This tool provides threat intelligence using Lucene query language.
- It also retrieves fingerprints information via searching by SHA-x, ssdeep and dexofuzzy hashes.
- It uses VirusTotal and malwarebazaar to get information of the threat intel providers.
- You can also customize your own rules to detect malware. It also detects third parties that collect data such as advertisement, analytics and more.
- Analyze source code to find vulnerabilities in the application. It detects common vulnerabilities like network traffic and potential vulnerabilities like debug flag, tap-jacking etc.

Now, let see how to setup this tool.

4.16.1 Setup

There are two ways to use this tool - first, directly click here to access web version of this tool. Second, you can build from source and use it on your Linux local machine. To build environment, follow below steps carefully -

```
$ git clone git@github.com:Pithus/bazaar.git
$ cd bazaar
```

Now, using docker-compose, you can build the project -

```
$ docker-compose -f local.yml build
```

To run the project, execute below command -

```
$ docker-compose -f local.yml up
```

Now, navigate to `http://localhost:8001` and use the tool. To check performance of this tool on DroidBench test suite see chap-6.

Chapter 5

Dynamic Analysis Tool

In this chapter, we discuss the methodology, features and installation and usage steps of dynamic analysis tool that are part of this work. To setup these tools, we have used steps given in github repository of tools or other internet resources.

5.1 Mobile Security Framework (MobSF) Static Analysis

An powerful open source pen-testing framework called MobSF [MobSF] is proficient of performing static and dynamic analysis on all types of mobile applications. Here, all type of mobile applications means whether the application is based on Android or iOS or Windows. But for dynamic analysis, it only works with Android APK.

Because of this tool's strength, the Mobile Security Testing Guide of the Open Web Application Security Project suggests using it for static analysis of mobile applications. It supports binaries and compressed source code analysis for Android, iOS and Windows mobile applications with ease and speed. It is a very easy-to-use tool. You just need to open this tool using the IP address and port number (e.g. 127.0.0.1:8000) after doing the required installation or you can also use mobsf.live. Then, just drag and drop any application you want to test. After uploading the application, the tool will first perform static analysis on the application and generate its report.

After performing static analysis, a report will be generated in which there will be an option to perform dynamic analysis on that app. MobSF developers have kept dynamic analysis in semi-automatic mode. The reason to do so is that different apps have different requirements like some require a username, password to login etc.

It also features CapFuzz-powered Web API fuzzing capabilities. MobSF is developed in such a way that it can make Continuous Integration/Continuous Deployment or DevSecOps pipeline integration easier to work with.

To perform Dynamic analysis, you should have an Android device emulator already installed and online. Dynamic analysis gives you many options to perform automated dynamic analysis like exported activity test, setting HTTP proxy, getting dependency, etc.

It also provides you with some basic frida scripts which you can also change according to need or you can also hook your own frida script. You can perform both static and dynamic analysis and save the result in pdf format if needed. Now, lets see its installation steps and usage.

5.1.1 Setup

Following are the steps to setup MobSF in Linux/Mac based system. Run below commands :

```
$ git clone https://github.com/MobSF/Mobile-Security-
    Framework-MobSF.git

$ cd Mobile-Security-Framework-MobSF
$ ./setup.sh
```

If everything goes alright. Then, you are ready to run MobSF -

```
$ ./run.sh 127.0.0.1:8000
```

Now, open your web browser and open following link - https://localhost:8000/ to access MobSF web interface. Here, you can drag and drop your Android APK to perform static analysis. You can also perform mass static analysis. You will need a REST API KEY for that which is available in documentation option on the webpage opened using the above URL. Then, you can run following command for mass static analysis -

```
$ pip install requests
```

```
$ python mass_static_analysis.py —d path/to/apk/dir —s <
    IP address and Port number of running MobSF server
    (127.0.0.1:8000)> —k <API KEY>
```

This command will save static analysis of all apps present in given directory in the result section of the webpage.
After performing static analysis, you can also perform dynamic analysis using the option given in the report of static analysis. But before performing dynamic analysis, you need to have genymotion emulator device installed in your system(For results on Droidbench test suite see chap-6).

5.2 DroidBox

DroidBox [Lantz et al.] is a tool developed to perform only dynamic analysis of the Android application. DroidBox is an extended version of TaintDroid.
The host and target, which make up DroidBox are its two component pieces. The target component that was launched on the Android emulator is based on Android 4.1.2 and includes a number of pacthes, the majority of which were taken directly from TaintDroid.

Some functions for monitoring data are also added by patches. The host component is a collection of pythonscripts that makes connection with Android emulator, then connect to target component and obtain all required data about the Android application being examined. Then, it displays the gathered data in form of text or graphic. The following information is represented in text format in the generated results -

- Hashings for the examined package
- Network data coming into or leaving
- Permissions to read and write file
- Loaded classes and started services through DexClass Loader
- Leaks of information via network, file and SMS.
- Circumvented permissions
- Cryptographic operations performed using Android API
- Makes list of broadcast receivers
- Sent SMS and phone calls

Now, lets see through how to setup DroidBox on Linux machine to use it.

5.2.1 Setup

This tool can be used on Linux and Mac OS. Here we will see how to use it on Linux specifically. First install some packages before proceeding to the installation -

```
$ sudo apt-get install python-numpy python-scipy python-
    matplotlib subversion
```

Now, check if $PATH contains path to your sdk-tools and sdk platform-tools or not. If not then use below command to add them temporarily (for permanently, update $PATH in .bashrc file) -

```
$ export PATH=$PATH:/path/to/android-sdk/tools/
$ export PATH=$PATH:/path/to/android-sdk/platform-tools/
```

Now, download the ready-made binary files of DroidBox and extract them anywhere-

```
$ wget https://github.com/pjlantz/droidbox/releases/
    download/v4.1.1/DroidBox411RC.tar.gz
```

Now, setup an Android emulator device (i.e. AVD) that targets Android 4.1.2 and choose Nexus4 as emulator device and ARM as CPU type using sdkmanager. After setting up an Android emulator device, run below command inside the extracted folder (for me it was DroidBox_4.1.1) and wait for your device to boot-up -

```
$ ./startemu.sh <AVD name>
```

When emulator has booted-up, run below command to start examining the applications -

```
$ ./droidbox.sh <file.apk> <(optional) duration in
    seconds>
```

This will install the application in AVD and load the main activity. Then, you can start using your app and droidbox will collect the logs. To end the analysis, just press ctrl+c. To check performance of this tool on Droidbench test suite refer to chap-6.

5.3 Quark-Engine

Quark-Engine [quark engine] is an Android application analysis tool that perform both static and dynamic analysis on the given application. It neglects obfuscation and score Android malware, so also called Obfuscation-Neglect Android Malware Scoring System. Obfuscation is a big problem for reverse engineering to detect malware.
Two functionalities that are present in Dalvik Bytecode Loader and helps in neglecting obfuscations - 1. Finding Native API calling sequence and cross reference. 2. Bytecode register tracking. These two functionalities also helps in perfect matching of Android malware scoring system. The tool also assign weights and thresholds at each stage of processing to compute the scores of malware. This generates the

well documented report after performing analysis on given application. Lets see its installation and usage.

5.3.1 Methodology

Quark-Engine goes through six stages to perform static and dynamic analysis on given Android mobile application. Lets see each stage -

- **Command Line -** This is the entry point, quark.cli that initialises the RuleObject and XRule object in accordance with the provided APK file and Json rules. Then, quark.utils.weight is created to calculate the weighted score to present it in the report.

- **APK Information Extract -** After performing reverse engineering on given APK, it extacts the permission request list in AndroidManifest.xml file, what native APIs are called. Then, the cross-reference method from supplied function name and the Dalvik bytecode instruction both can be obtained using Androguard.

- **Load Json Rule -** In this step, each Json file containing rules will be examined as five-steps rules of malware detection. Customised rule files can also be provided using -r option in command-line interface.

- **Level 1-5 Check -** These are the five steps check that each Json file goes through-

  - Requested permissions in the app
  - Native APIs call
  - A specific combination of native APIs calls
  - Native API calling sequence
  - APIs handling same registers

- **Weighted Score Calculation -** Each of the above stage gives their individual score. Using these scores, this tool calculates weighted-score by summing up each score based on their weights. Then, it will set the risk range such as low risk, medium risk or high risk using some set of formulaes.

- **Report Generation -** This is the final step of this tool in which it can generate two types of report based on user input - summarised or detailed. You can also filter the result based on confidence score of rules to examine the report easily.

Now, lets see the installation and usage steps of this tool.

5.3.2 Setup

This tool is also bundled with Kali Linux but we have tested this on Ubuntu20.04. You can easily install it using pip. But your system should already have Python3.8+, git, graphviz, click>=8.0.1 (for CLI supports). If all these are installed, then you can proceed further -

```
$ pip3 install -U quark-engine
```

To get the latest rules of Quark-engine to your home directory -

```
$ freshquark
```

There are lot of options in Quark-Engine. You can check them by running -

```
$ quark --help
```

An example of its using is given below -

```
$ quark -a <path to apk> -s
```

’-s’ represents that analysis is done with rules of quark-engine. To check performance of this tool on DroidBench test suite refer to chap-6.

5.4 AndroPyTool

AndroPyTool [Martín et al.] is a tool that performs both kinds of analysis on an application i.e. static and dynami analysis. It collectively uses exisiting malware analysis tools such as AndroGuard, Flowdroid, DroidBox and Virustotal analysis tool. AndroPyTool makes use of all these analysis tools to perform pre static, static and dynamic analysis. After completion of analysis, it generates report in JSON and CSV formats.

- **Pre-static phase** - In pre-static phase, it uses VirusTotal to extract information of application like filename, MD5, SHA256, AVClass and VirusTotal report.

- **Static analysis** - In this phase, Andropytool uses Androguard and Flowdroid to perform static analysis. Static analysis phase retrieves permissions of app, package name, services, receivers, API calls, system commands with help of Androguard and flowdroid retrieves all sources and sinks in the application.

- **Dynamic Analysis** - Now, next phase is Dynamic analysis that uses DroidBox integrated with Strace. Every system call made from the zygote process while the program is active is tracked by Strace. DroidBox analyze the data sent over network, file read and write operations, circumvented permissions, sent SMS and phone calls etc.

Now lets see the way of working of this tool.

5.4.1 Methodology

AndroPyTool performs seven steps pipeline to generate the analysis report -

- **APK filtering** - This steps uses Androguard tool to find whethern Android application is valid or not.

- **VirusTotal analysis** - This tool is used to report the results scanned by virustotal. Sixty different tools are also used to scan the given APK.
- **Dataset partitioning** - This step tags the APK to malware APK or benignware APK based on a threshold value ϵ. If the number of antivirus of VirusTotal tags this APK as malware is greater than ϵ, then it is tagged as malware otherwise benignware. This threshold value ϵ can be changed.
- **FlowDroid execution** - This tool is run against every sample of APK.
- **FlowDroid result processing** - This step retrieves the links between sources and sinks reported by previous step. To allow for independent adjustment of the result processing phase, the execution and result processing processes are segregated.
- **DroidBox execution** - This is modified version of DroidBox. This performs dynamic analysis of application and also includes Strace tool.
- **Feature extraction** - This step gathers the result of all previous steps and reorganise them in a single JSON and CSV files.

Now, lets see how to install and use this tool.

5.4.2 Setup

There are two ways to install - one using docker and another using source code. However, using docker is recommended and easy. So, lets see the steps to install it using docker. This can only be installed on Ubuntu machine.
From docker hub, pull the container -

```
$ docker pull alexmyg/andropytool
```

After successful completion of above command, you are ready to use AndroPyTool using below command -

```
$ docker run —volume=<path to apk  folder >{:/apks/}
   alexmyg/andropytool {−s /apks/} <arguments>
```

Keep text inside curly braces '{}' as it is for simplicity since, AndroPyTool moves your application files to different location to generate a structured working directory. Explore more options using '-h' option. Example -

```
$ docker run --volume=<path to your apk folder>:/apks/
    alexmyg/andropytool -s /apks/ -all -vt <VirusTotal API
      key> -drt 200
```

You can find use of these options in help section. You can get VirusTotal public API key by simplpy register here. To see performance of this tool on DroidBench test suite refer to chap-6.

5.5 MARA Framework

MARA [Christian] is another tool that performs both static and dynamic analysis on mobile application.This tool also makes use of a variety of other analysis tools to get its results and aid in the testing of applications for OWASP mobile security vulnerabilities. This is easy to use tool for developers and security professionals. Various features that are supported by this tool are - Domain analysis, APK deobfuscation, APK analysis, APK reverse engineering, APK manifest analysis, Security analysis.

5.5.1 Methodology

- For APK reverse enginering, it uses several different tools - baksmali and apktool to disassemble the dalvik bytecode to smali bytecode.

- Then, Enjarify is used to disassemble the dalvik bytecode to Java bytecode and jadx to decompile APK.

- Then, APK deobfuscation is carried out using Apk-deguard.com.

- APK analysis is also done using series of tools - Smalisca is used for parsing Smali files, Openssl is used to extract certificate data, aapt is used to extract string and app permissions, Classyshark is used to identify methods and classes, Androbugs is used to scan APK for vulnerabilities, Andowarn is used to scan APK for any kind of malicious behaviour, regex is used to find execution path, URL, URI, emails etc.

- APKiD is used to identify compilers, packers and obfuscators. There are various things that are done during APK manifest analysis - extract intents, services, receivers, exported activities and exported receivers etc.

- It also checks if app is debuggable, allow backups, allow to receive binary SMS, allow sending of secret codes.

- In Domain analysis, Pyssltest and Testssl is used for Domain SSL scan and Whatweb is used for website fingerprinting.

This are the main features of MARA Framework and tools used to perform the task. This tool also claims to generate Smali control-flow graph but in final output directory of MARA, graphs are nowhere to see(this bug is also mentioned in paper-[Joseph et al.]). Now, lets see how to use MARA Framework.

5.5.2 Setup

To download the source code, run below command -

```
$ git clone --recursive https://github.com/xtiankisutsa/
  MARA_Framework
```

Now, go inside this directory and run setup.sh with sudo privileges to install it.

```
$ cd MARA_Framework
#if running on Ubuntu machine
$ sudo ./setup.sh
```

```
#if running on mac
$ sudo ./setup_mac.sh
```

To update MARA installation -

```
$ sudo ./update.sh
```

To verify that instllation goes correctly, below command should output help section of MARA Framework.

```
$ ./mara.sh --help
```

To analyze .apk file -

```
$ ./mara.sh -s <path to apk>
```

This will store the generated output in 'data' sub-directory of MARA_Framework directory. To see performance of this tool on DroidBench test suite refer to chap-6.

5.6 Drozer

Drozer [Xiaopeng et al.] is a safety detection tool for Android applications. It is developed by MWR labs. It is an interactive Android security testing framework. It is used to carry out dynamic analysis on an application using a server machine and an agent need to be installed on Android device/emulator. The command from server machine are sent to drozer-agent installed in Android device/emulator and then the agent performs the intended action on application package mentioned in user input. It detects following types of vulnerabilities in Android - SQL injection vulnerability, rejection vulnerability, data backup vulnerability.

5.6.1 Methodology

Now, lets see how drozer tries to detect each vulnerability -

- **SQL injection vulnerability detection** - The Android component called Content Provider could be the reason of possible SQL injecton attack. This component is shared with apps that passes down and override the methods used to read and write data. Using the command app.package.attacksurface, it can be tracked if there is any exported content provider in the application. Then, you can find out about query-able providers using scanner.provider. command and exploit the accessible content URIs.

- **Denial of Service vulnerability detection** - Using intents, one can communicate within or outside the application. These is done using description with some additional information. Now, the corresponding component is found out using the given description and then passed onto the called component to complete the intended action. If developer doesn't use exception handling while this handling of components is done, the application may crash which means there exists a vulnerabilty.

- **Data backup vulnerability detection** - To find out about this vulnerability, drozer only need AndroidManifest.xml file. This file has allow-Backup attribute and this vulnerability depends on value of this attribute. This value is TRUE by default. This means use can backup and restore the app data using adb backup and adb reset and this is vulnerable. If value of allow-Backup is FALSE, then this vulnerability is not there in the application.

These are the vulnerabilities that drozer can find in the application that is being analyzed. Now, lets see how to setup and use this tool.

5.6.2 Setup

Now, first make sure you are using Python2.7 and pip for Python2.7. Then, follow below commands -

```
$ mkdir drozer
$ cd drozer
$ wget https://github.com/mwrlabs/drozer/releases/
  download/2.3.4/drozer-agent-2.3.4.apk
```

```
$ wget https://github.com/FSecureLABS/drozer/releases/
    download/2.4.4/drozer-2.4.4-py2-none-any.whl
$ pip install twisted
```

Now, we will build the wheel for drozer -

```
$ pip install drozer-2.4.4-py2-none-any.whl
```

After successful completion of above command, you are ready to use drozer. If there are some error, you an also install drozer using docker. See below command -

```
$ docker run -it fsecurelabs/drozer
```

Now, to use drozer, first start the Android emulator and install dozer-agent-2.3.4.apk using adb -

```
$ adb install drozer-agent-2.3.4.apk
```

Now, open the app and start the drozer agent. This will start drozer agent at 31415 by default. Now, run below command to forward the deafult port 31415 on the Android emulator to local port 31415.

```
$ adb forward tcp:31415 tcp:31415
```

Now, start drozer server on your machine -

```
$ drozer console connect
```

Now, you are ready to use drozer. For a brief tutorial about drozer, click here. To check performance of this tool on DroidBench test suite see chap-6.

Chapter 6

Experiments

In this chapter, we will see the performance of each tool that we saw in chapter-4 & 5 on Droidbench benchmark. This benchmark consists of 13 category containing 119 testcases. Each of these category tests a different aspect of Android security. Since, testcases name are very big, so we have used short-hand notation. Refer to below table-6.1 to see the abbreviations that we used for each testcase.

Table 6.1: Abbreviations used for Testcase Name

Testcase Name	**Abbrev-iation**	**Testcase Name**	**Abbrev-iation|**
Merge1	Mrg1	ArrayAccess1	ArAcc1
ArrayAccess2	ArAcc2	ArrayCopy1	ArCop1
ArrayToString1	ArTStr1	HashMapAccess1	HMAcc1
ListAccess1	Lacc1	MultidimensionalArray1	MulAr1
AnonymousClass1	AnCl1	Button1	Btn1
Button2	Btn2	Button3	Btn3
Button4	Btn4	Button5	Btn5
LocationLeak1	LocLk1	LocationLeak2	LocLk2
LocationLeak3	LocLk3	MethodOverride1	MtdOvr1

Continued on next page

Table 6.1: Abbreviations used for Testcase Name (Continued)

MultiHandlers1	MulHn1	Ordering1	Ord1
RegisterGlobal1	RegG1	RegisterGlobal2	RegG2
Unregister1	Unreg1	FieldSensitivity1	FldSen1
FieldSensitivity2	FldSen2	FieldSensitivity3	FldSen3
FieldSensitivity4	FldSen4	InheritedObjects1	InhObj1
ObjectSensitivity1	ObjSen1	ObjectSensitivity2	ObjSen2
Echoer	Ecr	SendSMS	SMS
StartActivityForResult1	SAR1	ActivityCommunication1	Acom1
ActivityCommunication2	Acom2	ActivityCommunication3	Acom3
ActivityCommunication4	Acom4	ActivityCommunication5	Acom5
ActivityCommunication6	Acom6	ActivityCommunication7	Acom7
ActivityCommunication8	Acom8	BroadcastTaintAndLeak1	BcTLk1
ComponentNotInManifest1	CNM1	EventOrdering1	EvtOrd1
IntentSink1	IntSnk1	IntentSink2	IntSnk2
IntentSource1	IntSrc1	ServiceCommunication1	Srvcom1
SharedPreferences1	ShdPrf1	Singletons1	Sngtn
UnresolvableIntent1	UnInt1	ActivityLifecycle1	ActLcl1
ActivityLifecycle2	ActLcl2	ActivityLifecycle3	ActLcl3
ActivityLifecycle4	ActLcl4	ActivitySavedState1	ActSSt1
ApplicationLifecycle1	AppLcl1	ApplicationLifecycle2	AppLcl2
ApplicationLifecycle3	AppLcl3	AsynchronousEventOrdering1	AEO1
BroadcastReceiverLifecycle1	BRLcl1	BroadcastReceiverLifecycle2	BRLcl2
EventOrdering1	EvtOrd1	FragmentLifecycle1	FrgLcl1

Continued on next page

Table 6.1: Abbreviations used for Testcase Name (Continued)

FragmentLifecycle2	FrgLcl2	ServiceLifecycle1	SrvLcl1
ServiceLifecycle2	SrvLcl2	SharedPreferenceChanged1	ShdPC1
Clone1	Clone1	Exceptions1	Excp1
Exceptions2	Excp2	Exceptions3	Excp3
Exceptions4	Excp4	FactoryMethods1	FacM1
Loop1	Loop1	Loop2	Loop2
Serialization1	Srln1	SourceCodeSpecific1	SCS1
StartProcessWithSecret1	SPWS1	StaticInitialization1	StInit1
StaticInitialization2	StInit2	StaticInitialization3	StInit3
StringFormatter1	StrFmt1	StringPatternMatching1	SPM1
StringToCharArray1	SchAr1	StringToOutputStream1	STOS1
UnreachableCode	UnCode	VirtualDispatch1	VirDis1
VirtualDispatch2	VirDis2	VirtualDispatch3	VirDis3
VirtualDispatch4	VirDis4	ApplicationModeling1	AppM1
DirectLeak1	DirLk1	InactiveActivity	InAc
Library2	Lib2	LogNoLeak	LnoL
Obfuscation1	Obfus1	Parcel1	Parcel1
PrivateDataLeak1	PDLk1	PrivateDataLeak2	PDLk2
PrivateDataLeak3	PDLk3	PublicAPIField1	PapiF1
PublicAPIField2	PapiF2	ImplicitFlow1	InfFlw1
ImplicitFlow2	InfFlw2	ImplicitFlow3	InfFlw3
ImplicitFlow4	InfFlw4	Reflection1	Ref1
Reflection2	Ref2	Reflection3	Ref3

Continued on next page

Table 6.1: Abbreviations used for Testcase Name (Continued)

Reflection4	Ref4	AsyncTask1	AsyTsk
Executor1	Exec1	JavaThread1	Jthrd1
JavaThread2	Jthrd2	Looper1	Looper1
ContentProvider1	ConPrv	IMEI1	IMEI1
PlayStore1	PlyStr1		

Following list gives category name in which every test case belongs -

- **Aliasing -**
  - Mrg1
- **Arrays and Lists -**
  - Aracc1
  - ArAcc2
  - ArCop1
  - ArTStr1
  - HMAcc1
  - Lacc1
  - MulAr1
- **Callbacks -**
  - Ancl1
  - Btn1
  - Btn2
  - Btn3
  - Btn4
  - Btn5
  - LocLk1
  - LocLk2
  - LocLk3
  - MtdOvr1
  - MulHn1
  - Ord1
  - RegG2
  - Unreg1
- **Field and Object Sensitivity -**
  - FldSen1
  - FldSen2
  - FldSen3
  - FldSen4
  - InhObj1
  - ObjSen1
  - ObjSen2
- **Inter-App Communication -**

- Ecr
- SMS
- SAR1

- **Inter-Component Communication -**
  - Acom1
  - Acom2
  - Acom3
  - Acom4
  - Acom5
  - Acom6
  - Acom7
  - Acom8
  - BcTLk1
  - CNM1
  - EvtOrd1
  - IntSnk1
  - IntSnk2
  - IntSrc1
  - Srvcom1
  - ShdPrf1
  - Sngtn
  - UnInt1
- **Lifecycle -**
  - ActLcl1
  - ActLcl2
  - ActLcl3
  - ActLcl4
  - ActSSt1
  - AppLcl1
  - AppLcl2
  - AppLcl3
  - AEO1
  - BRLcl1
  - BRLcl2
  - EvtOrd11
  - FrgLcl1
  - FrgLcl2
  - SrvLcl1
  - SrvLcl2
  - ShdPC1
- **General Java -**
  - Clone1
  - Excp1
  - Excp2
  - Excp3
  - Excp4
  - FacM1
  - Loop1
  - Loop2
  - Srln1
  - SCS1
  - SPWS1
  - StInit1
  - StInit2
  - StInit3
  - StrFmt1
  - SPM1
  - SchAr1
  - STOS1
  - UnCode
  - VirDis1
  - VirDis2
  - VirDis3
  - VirDis4
- **Android-Specific -**
  - AppM1
  - DirLk1
  - InAc
  - Lib2
  - LnoL
  - Obfus1
  - Parcel1
  - PDLk1
  - PDLk2
  - PDLk3
  - PapiF1
  - PapiF2
- **Implicit Flows -**

  - InfFlw1
  - InfFlw2
  - InfFlw3
  - InfFlw4

- **Reflection -**
  - Ref1
  - Ref2
  - Ref3
  - Ref4

- **Threading -**
  - AsyTsk
  - Exec1
  - Jthrd1
  - Jthrd2
  - Looper1

- **Emulator-Detection -**
  - ConPrv
  - IMEI1
  - PlyStr1

Now, we are ready to see the performance of each tool on this benchmark. Following are some notation that we use in the results -

- **True Positive (TP) -** This means that testcase has a leak and the tool also detects it. This is represented using '✓' in the result table.
- **True Negative (TN) -** This means that testcase doesn't have a leak and the tool also doesn't detect it. This is represented using '□' in the result table.
- **False Positive (FP) -** This means that testcase doesn't have a leak but the tool detects a leak. This is represented using '◯' in the result table.
- **False Negative (FN) -** This means that testcase has a leak but the tool doesn't detect it. This is represented using 'X' in the result table.

We have considered a testcase to be in true positive category only if the tool is able to detect source and sink in the analysis. If not then, it is considered to be false positive or false negative based on if testcase contains leak or not. Result table mentions which tool is static analyzer as 'S' and which is dynamic analyzer as 'D' in the column name 'Type'. Refer to the below table-6.2 for results on performance of each tool on Droidbench benchmark.

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X]

Type	Tools	Mrg1	ArAcc1	ArAcc2	ArCop1	ArTStr1	HMAcc1	Lacc1	MulAr1	AnCl1	Btn1
S	Amandroid	◯	◯	◯	X	X	◯	◯	X	X	X
S	Androbug	◯	◯	◯	X	X	◯	◯	X	X	✓
S	Androwarn	◯	◯	◯	✓	✓	◯	◯	X	X	X
S	APKleaks	□	□	□	X	X	□	□	X	X	X
S	Appsweep	□	□	□	✓	✓	□	□	✓	✓	X
S	DIDFAIL	◯	◯	◯	✓	✓	◯	◯	✓	✓	X
S	Flowdroid	◯	◯	◯	✓	✓	◯	◯	✓	✓	✓
S	IccTA	◯	◯	◯	✓	X	◯	◯	✓	✓	✓
S	JAADAS	□	◯	◯	X	X	◯	◯	X	X	X
S	Mariana	□	□	□	X	X	□	□	X	X	X
S	Mobileaudit	◯	◯	◯	✓	✓	◯	◯	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	Btn2	Btn3	Btn4	Btn5	LocLk1	LocLk2	LocLk3	MtdOvr1	MulHn1	Ord1
S	Amandroid	✓	X	X	X	X	X	X	✓	□	□
S	Androbug	X	✓	✓	X	X	X	X	X	□	□
S	Androwarn	✓	X	X	X	✓	✓	✓	✓	○	○
S	APKleaks	X	X	X	X	X	X	X	X	□	□
S	Appsweep	X	X	X	✓	✓	✓	✓	✓	○	○
S	DIDFAIL	X	X	X	X	✓	✓	✓	✓	○	○
S	Flowdroid	✓	✓	✓	✓	✓	✓	✓	✓	□	□
S	IccTA	✓	✓	✓	✓	✓	✓	✓	✓	□	□
S	JAADAS	X	X	X	X	X	X	X	X	□	□
S	Mariana	X	X	X	X	X	X	X	X	□	□
S	Mobileaudit	✓	✓	✓	✓	✓	✓	✓	✓	○	○

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	RegG1	RegG2	Unreg1	FldSen1	FldSen2	FldSen3	FldSen4	InhObj1	ObjSen1	ObjSen2
S	Amandroid	X	X	○	□	□	✓	□	✓	□	□
S	Androbug	✓	✓	○	○	○	✓	○	✓	○	○
S	Androwarn	X	X	□	○	○	✓	□	✓	○	○
S	APKleaks	X	X	□	□	□	X	□	X	□	□
S	Appsweep	X	X	□	□	□	X	□	X	□	□
S	DIDFAIL	✓	✓	○	○	○	✓	○	✓	○	○
S	Flowdroid	✓	✓	○	○	○	✓	○	✓	○	○
S	IccTA	✓	✓	○	□	□	✓	□	✓	□	□
S	JAADAS	X	X	□	○	○	✓	○	X	○	○
S	Mariana	X	X	□	□	□	X	□	X	□	□
S	Mobileaudit	✓	✓	□	○	○	✓	○	✓	○	○

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Ecr	SMS	SAR1	Acom1	Acom2	Acom3	Acom4	Acom5	Acom6	Acom7
S	Amandroid	X	✓	X	X	✓	✓	✓	✓	✓	✓
S	Androbug	X	✓	X	✓	X	X	X	X	X	X
S	Androwarn	✓	✓	✓	X	✓	✓	✓	✓	✓	✓
S	APKleaks	X	X	X	X	X	X	X	X	X	X
S	Appsweep	X	X	X	X	✓	✓	✓	✓	✓	✓
S	DIDFAIL	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	Flowdroid	X	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	IccTA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	JAADAS	✓	✓	✓	✓	✓	✓	✓	X	X	X
S	Mariana	X	X	X	X	X	X	X	X	X	X
S	Mobileaudit	X	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	Acom8	BcTLk1	CNM1	EvtOrd1	IntSnk1	IntSnk2	IntSrc1	Srvcom1	ShdPrf1	Sngtn
S	Amandroid	✓	✓	○	✓	✓	✓	X	X	✓	X
S	Androbug	X	X	○	X	X	X	X	X	X	X
S	Androwarn	✓	✓	○	✓	X	X	✓	✓	X	✓
S	APKleaks	X	X	□	X	X	X	X	X	X	X
S	Appsweep	✓	✓	○	✓	X	X	✓	✓	✓	✓
S	DIDFAIL	✓	✓	○	✓	✓	✓	✓	✓	X	✓
S	Flowdroid	✓	✓	○	✓	✓	✓	✓	✓	X	✓
S	IccTA	✓	✓	○	✓	X	✓	✓	X	✓	X
S	JAADAS	✓	✓	□	✓	X	X	✓	X	X	X
S	Mariana	X	X	□	X	X	X	X	X	X	X
S	Mobileaudit	✓	✓	○	✓	X	X	X	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	UnInt1	ActLcl1	ActLcl2	ActLcl3	ActLcl4	ActSSt1	AppLcl1	AppLcl2	AppLcl3	AEO1
S	Amandroid	✓	X	X	X	X	X	X	X	X	✓
S	Androbug	X	✓	✓	✓	✓	X	✓	✓	✓	X
S	Androwarn	✓	X	X	X	X	✓	X	X	X	X
S	APKleaks	X	X	X	X	X	X	X	X	X	X
S	Appsweep	✓	X	X	X	X	✓	X	X	X	✓
S	DIDFAIL	✓	X	X	X	X	X	✓	✓	✓	✓
S	Flowdroid	✓	✓	✓	✓	✓	X	✓	✓	✓	✓
S	IccTA	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	JAADAS	✓	X	✓	✓	✓	X	✓	X	✓	X
S	Mariana	X	X	X	X	X	X	X	X	X	X
S	Mobileaudit	✓	✓	✓	X	✓	✓	✓	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	BRLcl1	BRLcl2	EvtOrd1	FrgLcl1	FrgLcl2	SrvLcl1	SrvLcl2	ShdPC1	Clone1	Excp1
S	Amandroid	✓	X	X	X	X	✓	X	✓	X	✓
S	Androbug	✓	X	X	✓	X	✓	X	X	X	✓
S	Androwarn	✓	X	X	X	X	X	X	X	✓	✓
S	APKleaks	X	X	X	X	X	X	X	X	X	X
S	Appsweep	X	✓	✓	X	✓	X	✓	✓	✓	X
S	DIDFAIL	X	X	✓	✓	X	✓	✓	X	✓	X
S	Flowdroid	✓	X	✓	✓	X	✓	✓	X	✓	✓
S	IccTA	✓	X	✓	X	X	✓	✓	✓	✓	✓
S	JAADAS	✓	✓	X	X	X	X	X	X	X	✓
S	Mariana	X	X	X	X	X	X	X	X	X	X
S	Mobileaudit	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	Excp2	Excp3	Excp4	FacM1	Loop1	Loop2	Srln1	SCS1	SPWS1	StInit1
S	Amandroid	✓	○	X	X	X	X	X	✓	X	X
S	Androbug	✓	○	✓	X	✓	✓	X	✓	X	✓
S	Androwarn	✓	○	X	X	✓	✓	✓	✓	X	X
S	APKleaks	X	□	X	X	X	X	X	X	X	X
S	Appsweep	X	□	X	✓	X	X	✓	X	X	X
S	DIDFAIL	X	□	X	X	✓	✓	✓	✓	X	✓
S	Flowdroid	✓	○	✓	✓	✓	✓	✓	✓	X	✓
S	IccTA	✓	○	✓	✓	✓	✓	X	✓	X	X
S	JAADAS	✓	○	✓	X	✓	✓	X	✓	X	✓
S	Mariana	X	□	X	X	X	X	X	X	X	X
S	Mobileaudit	✓	○	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - ☐, FP - ◯, FN - X] (Continued)

Type	Tools	StInit2	StInit3	StrFmt1	SPM1	SchAr1	STOS1	UnCode	VirDis1	VirDis2	VirDis3
S	Amandroid	X	X	X	X	X	X	☐	✓	X	☐
S	Androbug	✓	X	X	X	X	X	☐	X	✓	◯
S	Androwarn	X	X	✓	✓	X	✓	◯	✓	✓	◯
S	APKleaks	X	X	X	X	X	X	☐	X	X	☐
S	Appsweep	X	✓	✓	✓	✓	✓	◯	✓	X	☐
S	DIDFAIL	✓	X	X	✓	✓	✓	☐	✓	✓	◯
S	Flowdroid	✓	X	X	✓	✓	✓	☐	✓	✓	◯
S	IccTA	✓	X	X	✓	✓	✓	☐	✓	✓	◯
S	JAADAS	✓	X	X	X	X	X	☐	X	✓	◯
S	Mariana	X	X	X	X	X	X	☐	X	X	☐
S	Mobileaudit	✓	✓	✓	✓	✓	✓	◯	✓	✓	◯

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	VirDis4	AppM1	DirLk1	InAc	Lib2	LnoL	Obfus1	Parcel1	PDLk1	PDLk2
S	Amandroid	□	X	✓	□	✓	□	✓	X	X	X
S	Androbug	○	X	✓	□	✓	□	✓	✓	✓	X
S	Androwarn	○	✓	✓	○	✓	□	✓	X	✓	✓
S	APKleaks	□	X	X	□	X	□	X	X	X	X
S	Appsweep	□	✓	X	○	X	○	X	X	X	✓
S	DIDFAIL	○	X	✓	○	✓	□	✓	X	✓	✓
S	Flowdroid	○	X	✓	□	✓	□	✓	✓	✓	✓
S	IccTA	□	X	✓	□	✓	□	✓	✓	✓	✓
S	JAADAS	○	X	✓	□	✓	□	✓	✓	X	✓
S	Mariana	□	X	X	□	X	□	X	X	X	X
S	Mobileaudit	○	✓	✓	○	✓	○	✓	✓	X	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	PDLk3	PapiF1	PapiF2	InfFlw1	InfFlw2	InfFlw3	InfFlw4	Ref1	Ref2	Ref3
S	Amandroid	X	X	✓	X	X	X	X	✓	X	X
S	Androbug	✓	X	X	X	X	X	X	✓	✓	✓
S	Androwarn	✓	✓	✓	X	✓	✓	✓	X	✓	✓
S	APKleaks	X	X	X	X	X	X	X	X	X	X
S	Appsweep	X	✓	✓	✓	✓	✓	✓	X	X	X
S	DIDFAIL	✓	✓	✓	X	X	X	X	X	X	X
S	Flowdroid	✓	✓	✓	X	X	X	X	✓	X	X
S	IccTA	✓	✓	X	X	X	X	X	✓	X	X
S	JAADAS	✓	X	X	X	X	X	X	✓	✓	✓
S	Mariana	X	X	X	X	X	X	X	X	X	X
S	Mobileaudit	✓	✓	✓	X	X	X	X	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Ref4	AsyTsk	Exec1	Jthrd1	Jthrd2	Looper1	ConPrv	IMEI1	PlyStr1	
S	Amandroid	X	X	X	✓	X	X	✓	X	✓	
S	Androbug	✓	X	X	X	X	X	X	X	X	
S	Androwarn	✓	✓	X	X	X	X	✓	X	X	
S	APKleaks	X	X	X	X	X	X	X	X	X	
S	Appsweep	X	✓	✓	✓	✓	✓	X	X	X	
S	DIDFAIL	X	X	X	X	X	X	X	X	X	
S	Flowdroid	X	✓	✓	✓	✓	✓	✓	X	✓	
S	IccTA	X	✓	✓	✓	X	X	✓	X	✓	
S	JAADAS	X	X	X	X	X	X	✓	✓	✓	
S	Mariana	X	X	X	X	X	X	X	X	X	
S	Mobileaudit	✓	✓	✓	✓	✓	✓	✓	✓	✓	

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Mrg1	ArAcc1	ArAcc2	ArCop1	ArTStr1	HMAcc1	Lacc1	MulAr1	AnCl1	Btn1
S	Pithus	◯	◯	◯	✓	✓	◯	◯	✓	✓	✓
S	QARK	□	□	□	✓	✓	□	□	✓	X	X
S	Reverse-APK	□	□	□	✓	✓	□	□	✓	✓	X
S	Riskindroid	◯	◯	◯	X	X	◯	◯	X	X	✓
S	Super-analyzer	◯	◯	◯	✓	X	◯	◯	✓	✓	✓
D	Andropytool	◯	◯	◯	X	X	◯	◯	X	✓	✓
D	Droidbox	□	□	□	X	X	□	□	X	X	✓
D	Drozer	□	□	□	X	X	□	□	X	X	X
D	MARA	◯	◯	◯	✓	✓	◯	◯	✓	✓	X
D	MobSF	◯	◯	◯	✓	✓	◯	◯	✓	✓	✓
D	Quark-engine	◯	◯	◯	X	X	◯	◯	X	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	Btn2	Btn3	Btn4	Btn5	LocLk1	LocLk2	LocLk3	MtdOvr1	MulHn1	Ord1
S	Pithus	✓	✓	✓	✓	✓	✓	✓	✓	○	○
S	QARK	X	X	X	✓	X	X	X	X	○	○
S	Reverse-APK	X	X	X	✓	✓	✓	✓	✓	○	○
S	Riskindroid	✓	✓	✓	✓	X	X	X	X	□	□
S	Super-analyzer	✓	✓	✓	✓	✓	✓	✓	X	○	○
D	Andropytool	✓	✓	✓	✓	✓	✓	✓	✓	□	□
D	Droidbox	X	✓	✓	X	X	X	X	X	□	□
D	Drozer	X	X	X	X	X	X	X	X	□	□
D	MARA	✓	X	X	✓	✓	✓	✓	✓	○	○
D	MobSF	✓	✓	✓	✓	✓	✓	✓	✓	○	○
D	Quark-engine	✓	✓	✓	✓	X	X	✓	✓	○	□

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	RegG1	RegG2	Unreg1	FldSen1	FldSen2	FldSen3	FldSen4	InhObj1	ObjSen1	ObjSen2
S	Pithus	✓	✓	□	○	○	✓	○	✓	○	○
S	QARK	X	X	□	□	□	X	□	X	□	□
S	Reverse-APK	X	X	□	□	□	X	□	X	□	□
S	Riskindroid	✓	✓	○	○	○	✓	○	✓	○	○
S	Super-analyzer	✓	✓	○	○	○	✓	○	✓	○	○
D	Andropytool	✓	✓	○	□	□	✓	□	✓	□	□
D	Droidbox	✓	X	□	□	□	✓	□	✓	□	□
D	Drozer	X	X	□	□	□	X	□	X	□	□
D	MARA	X	X	□	○	○	✓	□	✓	○	○
D	MobSF	✓	✓	○	○	○	✓	○	✓	○	○
D	Quark-engine	✓	✓	□	○	○	✓	○	✓	○	○

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Ecr	SMS	SAR1	Acom1	Acom2	Acom3	Acom4	Acom5	Acom6	Acom7
S	Pithus	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	QARK	✓	X	X	X	✓	✓	✓	✓	✓	✓
S	Reverse-APK	✓	X	✓	X	✓	✓	✓	✓	✓	✓
S	Riskindroid	X	✓	X	✓	X	X	X	X	X	X
S	Super-analyzer	✓	✓	✓	✓	X	X	X	X	X	X
D	Andropytool	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	Droidbox	X	X	X	✓	X	X	X	X	X	✓
D	Drozer	X	X	X	X	X	X	X	X	X	X
D	MARA	X	X	X	X	✓	✓	✓	✓	✓	✓
D	MobSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	Quark-engine	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	Acom8	BcTLk1	CNM1	EvtOrd1	IntSnk1	IntSnk2	IntSrc1	Srvcom1	ShdPrf1	Sngtn
S	Pithus	✓	✓	□	✓	X	X	✓	✓	✓	✓
S	QARK	✓	✓	○	✓	X	X	X	✓	✓	✓
S	Reverse-APK	✓	✓	○	✓	✓	✓	✓	✓	✓	✓
S	Riskindroid	X	X	□	X	X	X	X	X	X	X
S	Super-analyzer	X	✓	□	X	X	X	✓	✓	X	✓
D	Andropytool	✓	X	○	✓	✓	✓	✓	✓	X	X
D	Droidbox	X	X	□	✓	X	X	X	✓	X	X
D	Drozer	X	X	□	X	X	X	X	X	X	X
D	MARA	✓	✓	○	✓	X	X	✓	✓	✓	✓
D	MobSF	✓	✓	○	✓	X	X	✓	✓	✓	✓
D	Quark-engine	✓	X	○	✓	X	X	X	X	X	X

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	UnInt1	ActLcl1	ActLcl2	ActLcl3	ActLcl4	ActSSt1	AppLcl1	AppLcl2	AppLcl3	AEO1
S	Pithus	✓	X	✓	✓	✓	✓	✓	✓	✓	✓
S	QARK	✓	✓	X	X	X	✓	X	X	X	✓
S	Reverse-APK	X	✓	X	X	X	✓	X	X	X	✓
S	Riskindroid	X	✓	✓	✓	✓	X	✓	✓	✓	X
S	Super-analyzer	X	✓	✓	✓	✓	X	✓	✓	✓	✓
D	Andropytool	✓	✓	✓	✓	✓	X	✓	✓	✓	X
D	Droidbox	X	X	✓	X	✓	X	✓	X	✓	X
D	Drozer	X	X	X	X	X	X	X	X	X	X
D	MARA	✓	✓	X	X	X	✓	X	X	X	✓
D	MobSF	✓	X	✓	✓	✓	✓	✓	✓	✓	✓
D	Quark-engine	✓	✓	✓	✓	✓	X	✓	✓	✓	X

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	BRLcl1	BRLcl2	EvtOrd1	FrgLcl1	FrgLcl2	SrvLcl1	SrvLcl2	ShdPC1	Clone1	Excp1
S	Pithus	✓	✓	X	✓	✓	✓	✓	✓	✓	✓
S	QARK	X	✓	X	X	✓	X	✓	✓	✓	X
S	Reverse-APK	X	✓	X	X	✓	X	✓	✓	✓	X
S	Riskindroid	✓	X	X	✓	X	✓	X	X	X	X
S	Super-analyzer	✓	✓	X	✓	✓	✓	✓	X	✓	✓
D	Andropytool	✓	X	X	✓	X	✓	X	X	X	✓
D	Droidbox	X	X	X	✓	X	X	X	X	X	X
D	Drozer	X	X	X	X	X	X	X	X	X	X
D	MARA	✓	✓	✓	X	✓	X	✓	✓	✓	✓
D	MobSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	Quark-engine	✓	✓	X	✓	X	✓	X	X	X	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Excp2	Excp3	Excp4	FacM1	Loop1	Loop2	Srln1	SCS1	SPWS1	StInit1
S	Pithus	✓	◯	✓	✓	✓	✓	✓	✓	X	✓
S	QARK	X	□	X	✓	X	X	✓	X	X	X
S	Reverse-APK	X	□	X	✓	X	X	✓	X	X	X
S	Riskindroid	X	□	X	X	✓	✓	X	✓	X	✓
S	Super-analyzer	✓	◯	✓	✓	✓	✓	✓	✓	X	✓
D	Andropytool	✓	◯	✓	✓	✓	✓	X	✓	X	✓
D	Droidbox	✓	□	✓	X	✓	✓	X	✓	X	✓
D	Drozer	X	□	X	X	X	X	X	X	X	X
D	MARA	✓	◯	X	✓	✓	✓	✓	✓	X	X
D	MobSF	✓	◯	✓	✓	✓	✓	✓	✓	X	✓
D	Quark-engine	✓	◯	✓	✓	✓	✓	X	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ○, FN - X] (Continued)

Type	Tools	StInit2	StInit3	StrFmt1	SPM1	SchAr1	STOS1	UnCode	VirDis1	VirDis2	VirDis3
S	Pithus	✓	✓	✓	✓	✓	✓	○	✓	✓	○
S	QARK	X	✓	✓	✓	✓	✓	○	✓	X	□
S	Reverse-APK	X	✓	✓	✓	✓	✓	○	✓	X	□
S	Riskindroid	✓	X	X	X	X	X	□	X	X	○
S	Super-analyzer	✓	✓	✓	✓	✓	✓	□	✓	✓	○
D	Andropytool	✓	X	X	X	X	X	□	✓	X	□
D	Droidbox	✓	X	X	X	X	X	□	X	X	□
D	Drozer	X	X	X	X	X	X	□	X	X	□
D	MARA	X	✓	✓	✓	✓	✓	○	✓	✓	○
D	MobSF	✓	✓	✓	✓	✓	✓	○	✓	✓	○
D	Quark-engine	✓	X	X	X	X	X	○	✓	✓	○

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	VirDis4	AppM1	DirLk1	InAc	Lib2	LnoL	Obfus1	Parcel1	PDLk1	PDLk2
S	Pithus	◯	✓	✓	◯	✓	◯	✓	✓	✓	✓
S	QARK	□	✓	X	◯	X	◯	X	X	X	✓
S	Reverse-APK	□	✓	X	◯	X	◯	X	X	X	✓
S	Riskindroid	◯	X	✓	□	✓	□	✓	X	✓	X
S	Super-analyzer	◯	✓	✓	□	✓	◯	✓	✓	✓	✓
D	Andropytool	□	X	✓	□	✓	□	✓	X	✓	✓
D	Droidbox	□	X	✓	□	✓	□	✓	X	X	✓
D	Drozer	□	X	X	□	X	□	X	X	X	X
D	MARA	◯	✓	✓	◯	✓	◯	✓	X	✓	✓
D	MobSF	◯	✓	✓	◯	✓	◯	✓	✓	✓	✓
D	Quark-engine	◯	X	✓	◯	✓	□	✓	✓	✓	X

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	PDLk3	PapiF1	PapiF2	InfFlw1	InfFlw2	InfFlw3	InfFlw4	Ref1	Ref2	Ref3
S	Pithus	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
S	QARK	X	✓	✓	✓	X	X	X	X	X	X
S	Reverse-APK	X	✓	✓	✓	✓	✓	✓	X	X	X
S	Riskindroid	✓	X	X	X	X	X	X	✓	✓	✓
S	Super-analyzer	✓	✓	✓	X	X	X	X	✓	✓	✓
D	Andropytool	✓	X	X	X	X	✓	X	✓	✓	✓
D	Droidbox	✓	X	X	X	X	✓	X	✓	✓	✓
D	Drozer	X	X	X	X	X	X	X	X	X	X
D	MARA	✓	✓	✓	✓	✓	✓	✓	X	✓	✓
D	MobSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
D	Quark-engine	✓	X	X	✓	X	X	X	✓	✓	✓

Table 6.2: Results of all tools on DroidBench Benchmark [TP - ✓, TN - □, FP - ◯, FN - X] (Continued)

Type	Tools	Ref4	AsyTsk	Exec1	Jthrd1	Jthrd2	Looper1	ConPrv	IMEI1	PlyStr1	
S	Pithus	✓	✓	✓	✓	✓	✓	✓	✓	✓	
S	QARK	X	✓	✓	✓	✓	✓	X	X	X	
S	Reverse-APK	X	✓	✓	✓	✓	✓	X	X	X	
S	Riskindroid	✓	X	X	X	X	X	✓	✓	✓	
S	Super-analyzer	✓	✓	✓	✓	X	✓	✓	✓	✓	
D	Andropytool	✓	✓	✓	✓	✓	✓	✓	X	✓	
D	Droidbox	✓	X	✓	X	X	X	✓	X	✓	
D	Drozer	X	X	X	X	X	X	X	X	X	
D	MARA	✓	✓	✓	✓	✓	✓	✓	X	X	
D	MobSF	✓	✓	✓	✓	✓	✓	✓	✓	✓	
D	Quark-engine	✓	✓	✓	✓	✓	✓	✓	✓	✓	

In Droidbench benchmark, out of 119 testcases, 99 testcases leak the data that means they can either fall into true positive or false negative category and remaining 20 testcases don't leak data that means they can either fall into true negative or false positive category.
Now, next task is to evaluate the performance of these tools to find out which tool is best out of the tools that we have seen. For this we consider three parameters - Precision, Recall and F-score. For this we need counts of true positive, false positive and false negative.

- **Precision (P) -** The number of accurate positive labels to the total number of positive labels found by tools is the measure of precision.

$$\text{Precision(P)} = {}^{C_{TP}}/_{C_{TP}+C_{FP}}$$

- **Recall (R) -** The proportion of correctly labeled positive samples to actually positive samples is known as recall.

$$\text{Recall(R)} = {}^{C_{TP}}/_{C_{TP}+C_{FN}}$$

$$\text{Here, } C_{TP} = \text{count of True Positives for a tool}$$

$$C_{FP} = \text{count of False Positives for a tool}$$

$$C_{FN} = \text{count of False Negatives for a tool}$$

- **F-score -** The harmonic mean of Precision and Recall is the F-score.

$$\text{F-score} = {}^{\text{2*P*R}}/_{\text{P+R}}$$

Here, we judge the performance of tool on the basis of F-score. Refer to the table-6.3 for evaluation results. In table, Hyphen(-) denotes that the corresponding value is undefined.

Table 6.3: Evaluation of each tool on Droidbench benchmark

Type	Tools	Precision	Recall	F-score
S	Amandroid	0.81	0.34	0.48
S	Androbug	0.72	0.38	0.5
S	Androwarn	0.77	0.57	0.66
S	APKleaks	-	0	-
S	Appsweep	0.9	0.53	0.67
S	DIDFAIL	0.78	0.6	0.68
S	Flowdroid	0.84	0.82	0.83
S	IccTA	0.89	0.76	0.82
S	JAADAS	0.77	0.41	0.54
S	Mariana-trench	-	0	-
S	Mobileaudit	0.82	0.9	0.86
S	Pithus-bazaar	0.84	0.95	0.89
S	QARK	0.88	0.44	0.59
S	Reverse-APK	0.9	0.56	0.69
S	Riskindroid	0.75	0.38	0.5
S	Super-analyzer	0.82	0.77	0.79
D	Andropytool	0.9	0.7	0.79
D	Droidbox	1	0.35	0.52
D	Drozer	-	0	-
D	MARA	0.8	0.73	0.76
D	MobSF	0.83	0.96	0.89
D	Quark-engine	0.8	0.68	0.74

* Hyphen(-) denotes undefined value

According to evaluation results -

- **Evaluation of Static Analysis tools -**
  - In list of static analysis tool, Pithus-bazaar is the one with the highest F-score of 0.89, followed by Mobileaudit with F-score of 0.86 and then Flowdroid with F-score of 083.
  - Precision of Appsweep is highest with the value of 0.90 among all the toolsand Recall of Pithus-bazaar is highest with the value of 0.95 among all the tools.
  - Precision of Pithus-bazaar and Flowdroid is same with value of 0.84.
  - There are two tools with undefined value of F-score, APKleaks and Mariana-trench.
- **Evaluation of Dynamic Analysis Tool -**
  - In this category of tools, Mobsf leads with F-score value of 0.89.
  - Then, second best tool according to F-score is Andropytool with the value of 0.79 and then it is followed by MARA with F-score value of 0.76.
  - Droidbox has highest Precision value of 1. Mobsf has highest recall value of 0.96.
  - Here, Drozer has undefined value of Precision and F-score.

Now, lets see which tool is performing best for each testcase category and then we will see which tool is performing best for the whole Droidbench benchmark.

Chapter 7

Inferences

In this chapter, we will see the results of Static and Dynamic Analysis tools on each category of Droidbench testcases. There are thirteen testcases category. First, we will see best performing static analysis tool on each category. Refer to table-6.2 for results.

7.1 Static Analysis Tools Evaluation

- **Aliasing -** This category contains only one test case and that too doesn't contain a leak. Jaadas, Qark and Reverseapk correctly work in this category for given testcase.

- **Arrays and Lists -** This category contains seven testcases. Out of these seven testcases, three leaks the data and the others don't leak any data. In this category, QARK and Reverseapk performs best. Both these tools detects all test cases correctly. Flowdroid and Didfail are second best performer that are unable to detect three cases correctly.

- **Callbacks -** This category contains fifteen testcases. Out of these fifteen, twelve testcases leak the data and the remaining three don't. In this category, there are two tools that give good results, IccTA and Flowdroid. Both have same the result on each testcase. Pithus is second best performer that detects two less true positives than Flowdroid.

- **Field and Object Sensitivity -** This category contains seven testcases. Out of these seven, only two leak the data and the others don't leak any data. In this category, Amandroid and IccTA perform best. Androwarn is second best performer in this category that detects two false positives.

- **Inter-App Communication -** This category contains three testcases and all the three leak the data. In this category, there is a big list that performs good - Androwarn, Didfail, IccTA, Jaadas, Pithus, Super. Flowdroid is second best performer that is unable to detect one test case that leaks the data.

- **Inter-Component Communication -** This category contains nineteen testcases. Out of these nineteen, only one doesn't leak data. In this category, Didfail and Flowdroid perform best. Pithus detects one more false negative in comparsion to Flowdroid and Didfail.

- **Lifecycle -** This category contains seventeen testcases and all of them leak the data. In this category, Mobileaudit performs best and Pithus performs second best. Pithus doesn't detect one more testcase leak in comparison to Mobileaudit.

- **General Java -** This category contains twenty-three testcases and only four of them don't leak any data. In this cactegory, Super and Mobileaudit performs best but precision of Super is more and recall of Mobileaudit is more. Pithus detects one more false positive as compared to Super and Mobileaudit.

- **Android Specific -** This category contains twelve testcases and only two of them don't leak the data. In this category, Flowdroid and Super perform very well. Precision of Flowdroid is high but recall of Super is high. Here also, Pithus is second best performer as it detects one more false positive as compared to Super and Flowdroid.

- **Implicit Flows -** This category contains four testcases and all of them leak the data. In this category, Appsweep, Pithus and Reverseapk perform very well. Androwarn performs second best in this category as it detects one more false negative as compared to best performing tools.

- **Reflection -** This category contains four testcases and all of them leak the data. In this category, various tools detect all the leaks - Androbug, Mobileaudit, Pithus, Riskindroid and Super. All of these correctly detect the leak in all testcases. In this category, Androwarn performs second best as it detects one more false negatives as compared to best performing tools.

- **Threading -** This category contains five testcases and all of them leak the data. In this category, various tools perform very well - Flowdroid, Qark, Reverseapk, Appsweep, Pithus. Here, Super detects one more false negatives so it is second best performer.

- **Emulator Detection -** This category contains three testcases and all of them leak the data. In this category, various tools perform very well - Jaadas, Pithus, Super and Mobileaudit. In this, Flowdroid and IccTA are second best performer as they detect one more false negative when compared to best performing tool.

Out of sixteen static analysis tool that we have explored, Pithus has highest F-score, Mobileaudit has second highest F-score and Flowdroid has third highest F-score on the whole Droidbench benchmark. Now, lets see evaluation of dynamic tools.

7.2 Dynamic Analysis Tools Evaluation

- **Aliasing -** For this category, it is hard to decide which tool works best but Droidbox works correctly in this testcase category.

- **Arrays and Lists -** In this category, Mara and Mobsf work good. All other tool do not detect any true positive in this category.

- **Callbacks -** In this category, Andropytool works really well. Mobsf is second best in this category as it detects more false positives.
- **Field and Object Sensitivity -** In this category, Andropytool and Droidbox work best. Both predict all test cases correctly. MARA is second performer in this category as it detects four false positives.
- **Inter-App Communication -** In this category, three tools perform best - Quark-engine, Mobsf and Andropytool. All other tools do not detect any true positive in this category.
- **Inter-Component Communication -** In this category, Mobsf works really well. Mara and Andropytool performs second best in this category as they detect one more false negative as compared to Mobsf.
- **Lifecycle -** In this category, Mobsf works really well. Quark is second best performer in this category as it detects five more false negatives in comparison to Mobsf.
- **General Java -** Mobsf performs very good in this testcase category. Here, Mara performs second best as it detects three more false negatives in comparison to Mobsf.
- **Android Specific -** In this category also Mobsf performs best out of all dynamic analysis tool. Here Mara detects one more false negatives so it is second best performer in this category.
- **Implicit Flows -** Mara and Mobsf are best performing tools in this category. Both detect each testcase correctly. Here, Andropytool, Droidbox and Quark are second best performer.
- **Reflection -** Andropytool, Mobsf, Droidbox and Quark-engine are the tools that perform excellent in this category. Mara is second best performer that detects one more false negative.

- **Threading -** In this category, four tools perform best - Quark-engine, Mobsf, Andropytool and Mara. Second best performer is Droidbox that detects only one true positive.
- **Emulator Detection -** Mobsf and Quark-engine perform excellent in this category. Here, Andropytool and Droidbox are second best performer as they one less true positive when compared to best performing tools.

As you can see, Mobsf is performing best in almost all category except Aliasing, Callbacks and Field and Object Sensitivity category. So, we can say Mobsf is giving the best performance overall on the Droidbench benchmark. F-score(refer to table-6.3) of Mobsf also shows the same result. Andropytool is giving the second best performance on Droidbench.

Chapter 8

Future Work and Conclusion

In this work, we have explored sixteen static analysis tools and six dynamic analysis tools and evaluated their performance on Droidbench test suite. We saw what types of vulnerabilities each testcase contains in Droidbench. We have also discussed features, methodology, installation and usage steps of all these tools. We have divided the results into two categories of static and dynamic. We have also seen best performing tools of both(static and dynamic) types in each category of Droidbench. Out of all these 22 tools, Pithus-bazaar(static tool) and Mobsf(dynamic tool) perform best on the overall Droidbench benchmark.

8.1 Scope for further work

This work is a small effort towards making the Android mobile applications security more stronger. In this, we only focused on open-source static and dynamic analysis tools. In future, this work can be extended to explore more tools that work using machine learning techniques. New static and dynamic analysis tools to evaluate their performance. Same tools can also be evaluated on other benchmarks. We can also explore some paid online analysis tools. In the long term, we hope the community will find this work useful and will contribute to this work.

References

[1] 1N3 (2021). Github - 1n3/ReverseAPK: Quickly analyze and reverse engineer Android packages. https://github.com/1N3/ReverseAPK. [Online; accessed 2022-10-13].

[2] Albakri, A., Fatima, H., Mohammed, M., Ahmed, A., Ali, A., Ali, A., and Elzein, N. M. (2022). Survey on reverse-engineering tools for android mobile devices. *Mathematical Problems in Engineering*, 2022.

[3] Androguard (2022). Github - androguard/androguard: Reverse engineering and pentesting for Android applications. https://github.com/androguard/androguard. [Online; accessed 2022-10-10].

[4] Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Le Traon, Y., Octeau, D., and McDaniel, P. (2014). Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269.

[5] Bob, P. (2021). Github - pxb1988/dex2jar: Tools to work with android .dex and java .class files. https://github.com/pxb1988/dex2jar. [Online; accessed 2022-10-10].

[6] Christian, K. (2019). Github - xtiankisutsa/MARA_Framework: Mara is a Mobile Application Reverse engineering and Analysis Framework. It is a toolkit that puts together commonly used mobile application reverse engineering and analysis tools to assist in testing mobile applications against the OWASP mobile security threats. https://github.com/xtiankisutsa/MARA_Framework. [Online; accessed 2022-10-07].

[7] Debize, T. (2012). maaaaz/androwarn: Yet another static code analyzer for malicious android applications.

[8] Dwi, S. (2022). Github - dwisiswant0/apkleaks: Scanning APK file for URIs, endpoints secrets. https://github.com/dwisiswant0/apkleaks. [Online; accessed 2022-10-04].

[9] Dwivedi, K., Yin, H., Bagree, P., Tang, X., Flynn, L., Klieber, W., and Snavely, W. (2017). Didfail: Coverage and precision enhancement. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA PITTSBURGH United States.

[10] Facebook (2022). Github - facebook/mariana-trench: Our security focused static analysis tool for Android and Java applications. https://github.com/facebook/mariana-trench. [Online; accessed 2022-10-17].

[11] Flankerhqd and Shrivastava, A. (2017). Github - flankerhqd/JAADAS: Joint Advanced Defect assEsment for android applications. [Online; accessed 2022-06-01].

[12] Google (2016). Github - google/enjarify. https://github.com/google/enjarify. [Online; accessed 2022-10-10].

[13] Green, M. and Smith, M. (2016). Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46.

[14] Gruver, B. (2015). Smali/baksmali tool.

[15] Guardsquare (2021). Github - Guardsquare/appsweep-gradle: This Gradle plugin can be used to continuously integrate app scanning using AppSweep into your Android app build process. https://github.com/Guardsquare/appsweep-gradle. [Online; accessed 2022-10-09].

[16] Haris, M., Jadoon, B., Yousaf, M., and Khan, F. H. (2018). Evolution of android operating system: a review. *Asia Pacific Journal of Contemporary Education and Communication Technology*, 4(1):178–188.

[17] Joseph, R. B., Zibran, M. F., and Eishita, F. Z. (2021). Choosing the weapon: A comparative study of security analyzers for android applications. In *2021 IEEE/ACIS 19th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 51–57. IEEE.

[18] Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T. F., and Klein, J. (2018). Automated testing of android apps: A systematic literature review. *IEEE Transactions on Reliability*, 68(1):45–66.

[19] Kulkarni, K. and Javaid, A. Y. (2018). Open source android vulnerability detection tools: a survey. *arXiv preprint arXiv:1807.11840*.

[20] Lantz, P., Desnos, A., and Yang, K. (2011). Droidbox: An android application sandbox for dynamic analysis. *GitHub: San Francisco, CA, USA*.

[21] Li, L., Bartel, A., Bissyandé, T. F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D., and McDaniel, P. (2015). Iccta: Detecting inter-component privacy leaks in android apps. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 280–291. IEEE.

[22] Lin, Y.-C. (2015). Github - AndroBugs/AndroBugs_Framework: Androbugs Framework is an efficient Android vulnerability scanner that helps developers or hackers find potential security vulnerabilities in Android applications. No need to install on Windows. [Online; accessed 2022-06-05].

[23] Linkedin (2019). Github - linkedin/qark: Tool to look for several security related Android application vulnerabilities. [Online; accessed 2022-06-07].

[24] Martín, A., Lara-Cabrera, R., and Camacho, D. (2018). A new tool for static and dynamic android malware analysis. In *Data Science and Knowledge Engineering for Sensing Decision Support: Proceedings of the 13th International FLINS Conference (FLINS 2018)*, pages 509–516. World Scientific.

[25] Merlo, A. and Georgiu, G. C. (2017). Riskindroid: Machine learning-based risk analysis on android. In *Ifip international conference on ict systems security and privacy protection*, pages 538–552. Springer.

[26] Meshram, P. and Thool, R. (2014). A survey paper on vulnerabilities in android os and security of android devices. In *2014 IEEE Global Conference on Wireless Computing & Networking (GCWCN)*, pages 174–178. IEEE.

[27] Mitra, J. and Ranganath, V.-P. (2017). Ghera: A repository of android app vulnerability benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52.

[28] MobSF (2022). Github - MobSF/Mobile-Security-Framework-MobSF: Mobile Security Framework (MobSF) is an automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, malware analysis and security assessment framework capable of performing static and dynamic analysis. https://github.com/MobSF/Mobile-Security-Framework-MobSF. [Online; accessed 2022-10-17].

[29] Mpast (2022). Github - mpast/mobileAudit: Django application that performs SAST and Malware Analysis for Android APKs. https://github.com/mpast/mobileAudit. [Online; accessed 2022-10-06].

[30] Nielson, F., Nielson, H. R., and Hankin, C. (2004). *Principles of program analysis*. Springer Science & Business Media.

[Nowsecure] Nowsecure. Frida • A world-class dynamic instrumentation framework. https://frida.re/. [Online; accessed 2022-11-10].

[32] Pauck, F., Bodden, E., and Wehrheim, H. (2018). Do android taint analysis tools keep their promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 331–341.

[33] Pithus (2022). Github - Pithus/bazaar: Android security privacy analysis for the masses. https://github.com/Pithus/bazaar. [Online; accessed 2022-10-07].

[34] quark engine (2022). Github - quark-engine/quark-engine: Android Malware (Analysis | Scoring) System. https://github.com/quark-engine/quark-engine. [Online; accessed 2022-10-07].

[35] Ranganath, V.-P. and Mitra, J. (2020). Are free android app security analysis tools effective in detecting known vulnerabilities? *Empirical Software Engineering*, 25(1):178–219.

[36] Reaves, B., Bowers, J., Gorski III, S. A., Anise, O., Bobhate, R., Cho, R., Das, H., Hussain, S., Karachiwala, H., Scaife, N., et al. (2016). * droid: Assessment and evaluation of android application analysis tools. *ACM Computing Surveys (CSUR)*, 49(3):1–30.

[37] Reps, T., Horwitz, S., and Sagiv, M. (1995). Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA. Association for Computing Machinery.

[38] Sadeghi, A., Bagheri, H., Garcia, J., and Malek, S. (2016). A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software. *IEEE Transactions on Software Engineering*, 43(6):492–530.

[39] Scandariato, R. and Walden, J. (2012). Predicting vulnerable classes in an android application. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 11–16.

[40] Singh, V. and Sharma, K. (2016). Smartphone security: Review of challenges and solution. In *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, pages 1–3.

[41] Skylot (2022). Github - skylot/jadx: Dex to Java decompiler. https://github.com/skylot/jadx. [Online; accessed 2022-10-07].

[42] SUPERAndroidAnalyzer (2018). Github - SUPERAndroidAnalyzer/super: Secure, Unified, Powerful and Extensible Rust Android Analyzer. https://github.com/SUPERAndroidAnalyzer/super. [Online; accessed 2022-10-04].

[43] Tam, K., Feizollah, A., Anuar, N. B., Salleh, R., and Cavallaro, L. (2017). The evolution of android malware and android analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):1–41.

[44] Tiwari, P. and Velayutham, T. (2019). Avs(android vulnerability scanner): Static analysis for scanning of android vulnerabilities.

[45] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. (2010). Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224.

[46] Wang, Y., Zheng, J., Sun, C., and Mukkamala, S. (2013). Quantitative security risk assessment of android permissions and applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 226–241. Springer.

[47] Wei, F., Roy, S., Ou, X., and Robby (2018). Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. *ACM Trans. Priv. Secur.*, 21(3).

[48] Wei, Z. and Lie, D. (2014). Lazytainter: Memory-efficient taint tracking in managed runtimes. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices*, pages 27–38.

[49] Winsniewski, R. (2012). Android–apktool: A tool for reverse engineering android apk files. *Retrieved February*, 10:2020.

[50] Xiaopeng, L., Ning, W., Fei, X., Fengchen, Q., and Simin, M. (2018). Safety detection method of android app based on drozer. In *2018 International Conference on Smart Grid and Electrical Automation (ICSGEA)*, pages 170–172. IEEE Computer Society.