B-droid: A Static Taint Analysis Framework for Android Applications

Rehab ALmotairy[1], Yassine Daadaa[2]
College of Computer and Information Sciences
Al Imam Mohammad Ibn Saud Islamic University (IMSIU)
Riyadh, Saudi Arabia

***Abstract*—Android is currently the most popular smartphone operating system in use, with its success attributed to the large number of applications available from the Google Play Store. However, these contain issues relating to the storage of the user's sensitive data, including contacts, location, and the phone's unique identifier (IMEI). Use of these applications therefore risks exfiltration of this data, including unauthorized tracking of users' behavior and violation of their privacy. Sensitive data leaks are currently detected with taint analysis approaches. This paper addresses these issues by proposing a new static taint analysis framework specifically for Android platforms, termed "B-Droid". B-Droid is based on static taint analysis using a large set of sources and sinks techniques, side by side with the fuzz testing concept, in order to detect privacy leaks, whether malicious or unintentional by analyses the behavior of Applications Under Test (AUTs). This has the potential to offer improved precision in comparison to earlier approaches. To ensure the quality of our analysis, we undertook an evaluation testing a variety of Android applications installed on a mobile after filtering according to the relevant permissions. We found that B-Droid efficiently detected five of the most prevalent commercial spyware applications on the market, as well as issuing an immediate warning to the user, so that they can decide not to continue with the AUTs. This paper provides a detailed analysis of this method, along with its implementation and results.**

Keywords*—Static analysis; taint analysis; fuzz testing; android applications; mobile malwares; data flow analysis

I. Introduction

Android's market share of mobile phones grew to 74.6% in 2020 [1]. However, there remains considerable concern that popular Android apps tend to leak sensitive information about the user, i.e. phone number, the ID of the mobile device, location, and details of the Subscriber Identity Module (SIM) card. In addition to violating the privacy policies of the user, this can potentially lead to the user's behavior being unknowingly tracked. Even precisely programmed apps may suffer from these leaks. A major contributor to user data leaks is advertisement libraries, included by some applications to earn money, often enabling the applications to be free to use [2]. These libraries permit advertisements to target a user's private information and identify him or her through unique identifiers (e.g. the MAC-address and IMEI) as well as location or country [3]. However, many app developers are unaware of the scale to which user's privacy can be compromised and the large amounts of data potentially held by these ad libraries.

A number of steps have been previously undertaken to ensure the security of apps available to users, but this procedure is complex. For example, developers can upload a new app to the Google Play Store, where it is checked by the Google Bouncer [4]. This analyses the security of the app by conducting a time-limited dynamic analysis, with the aim of identifying any malicious content or behavior. This represents welcome progress, but has, offered only limited success. The majority of mobile platforms, including Android, limit the privileges of apps with a permission model, one that should prevent such apps from gaining access to sensitive data. However, this does not always prove effective, resulting in the release of sensitive data [2]. Such data leaks may not only occur in response to malicious apps, but also be due to an app's failure to adhere to secure coding practices [4]. It is therefore vital to analyze the data flow within the device, ensuring firstly, sensitive data does not cross established boundaries and secondly, that any untrusted aspect is unable to enter trusted data repositories. This type of analysis is known as taint analysis, i.e. any untrustworthy data source is considered to be tainted. Taint analysis has been conducted on Android systems by several researchers.

This current paper focuses on establishing whether data can flow from a sensitive data source to an undesired data sink. For example, a smartphone holds data that should be private to the user, i.e. personal messages, unique identifier and banking details. An undesired sink can be set up to divert the network Application Programming Interface (API) or applications that cannot be trusted. This study's definition of a data source is one that is external to the app, and from which the app reads data (i.e. a device's ID, along with contacts, photographs, and current location). A data sink is a resource external to the app to which the app writes data (i.e. the Internet, outbound text messages and the file system).

II. Background and Related Work

In the following sections, we provide some concepts which are relevant to our research and lists previous and related work in the field of static taint analysis.

A. Background

This section provides the theoretical background to the present study, including an overview of Android systems, taint analysis, mobile malwares, and the fuzz testing concept.

1) Android overview: This study focuses on Android, which is currently the most popular mobile platform globally,

being ranked as the premier operating system for smartphones in 2020 [1]. Android apps generally use four types of components, either on their own or in combination. Fig. 1 (below) illustrates the possible interactions between these components.

- An Activity refers to the parts of the Android app visible to the user, thus forming the user interface.
- A Service performs tasks that are time intensive but invisible to the user, i.e. they run in the background.
- System events and user-specific events are received by the Broadcast Receiver.
- The Content Provider allows additional apps or components to access unstructured data, functioning as a standard interface.

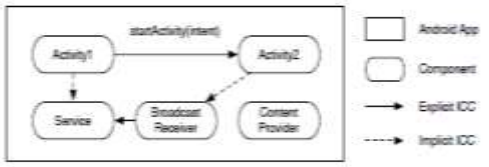

Fig. 1. Overview of Basic Concepts of Android Apps.

2) Taint analysis overview: Taint analysis can be either dynamic or static. Static analysis can be described as a method for program analysis in which the source code (or bytecode) is analyzed but not executed. On the other hand, dynamic analysis involves analyzing a code or application by means of its execution, i.e. analyzing an Android app by running it on an Android device.

Examples of dynamic taint analysis include CopperDroid [5], TaintDroid [3], DroidScope [6] and Aurasium [7]. However, it should be noted that the rate of execution of dynamic analysis is less advanced and deficient in code coverage [8] in comparison to static taint analysis.

Static taint analysis lacks runtime overheads, but can detect privacy leaks without running the application [2]. Static taint analyses do not contain the same problems as dynamic taint analyses, but they have their own issues, i.e. they lack the precision required to approximate runtime objects and abstract from the inputs of a program [9]. Much work has been invested in static taint analysis, with over half focusing on points-to-based methods and data flow ([9], [10], [11], [12], [13]). Little research has previously been conducted into analyzing data flows statically in a system comprised of multiple applications. This is significant, as the data's path to a sink might involve passing through one or more components. The advantage of static analysis is that all of the possible execution paths can be explored, not just those invoked during execution. This is beneficial when conducing security analysis, due to breaches of security frequently occurring through unforeseen methods. However, it may not be easy to predict how a program will behave if it is not actually executed. Furthermore, it has been proven that it is not possible to predict all of the means by which an arbitrary nontrivial program may be executed. Thus, the behavior of programs cannot always be correctly predicted. Nonetheless, static analysis remains valuable, as it provides some approximation of how different parts of a program may behave when executed [14].

3) Mobile malwares: Skycure [15] have reported that as many as one in three mobile devices has a medium to high risk of disclosing its user's data. Furthermore, in comparison with iOS devices, Android devices demonstrate almost twice the level of risk of containing malware. This section discusses the most common and damaging malwares impacting on mobile phones.

a) Trojan: This is defined as software that works in the background, acting maliciously, but regarded by the user as harmless. A Trojan assists hacker by carrying out actions facilitating attacks by weakening the system security. One example is FakeNetflix, which tricks users into believing they are downloading the official version of Netflix, but instead installs a Trojan on the phone, allowing hackers to access an Android user's credentials for their Netflix account.

b) Spyware: This is further common type of malware, consisting of software allowing its author to 'spy' on the user by facilitating unauthorized access to data or collecting information. It runs in the background of the system, where it remains unnoticed by the device user. Two examples of spyware are Nickspy and GPSSpy, both of which run on Android devices. They are able to access the user's confidential information and transmit it to the author of the spyware.

4) Fuzz testing: Fuzz testing or fuzzing [16] is an effective technique for finding security vulnerabilities in software or computer systems. It is a technique traditionally used for testing software and can be fully or semi-automated. It supplies unexpected, invalid, or random data as inputs to applications, which can then be monitored to determine whether it engages in unexpected behavior, crashes, or fails built-in code assertions. Applications under fuzz testing fail when they behave in a manner their developers neither intended nor anticipated. There are four categories of failure modes in traditionally applied fuzzing:

a) Crashes
b) Endless loops
c) Resource leaks or shortages
d) Unexpected behavior

These failure modes vary, based on factors including the type of system or software being tested and the underlying operating system. In this current study, B-Droid was designed and implemented according to the last failure mode category, i.e. unexpected behavior [17]. The consequences depend on the function and purpose of the software, including when and where it is operated. In general, the key to detecting malware is to place it in an ideal environment, followed by providing it with information and monitoring its behavior.

B. Related Work

The literature has suggested that a large body of work is related to static taint analysis approaches, each of which aims to resolve one or more of the many problems facing program analysts when dealing with Android applications. In this section we briefly review some of the existing solutions.

Arzt et al. [9] used the FlowDroid static taint analysis approach to simultaneously evaluate efficiency and precision. This is also the tool we used in our own approach. The results showed that FlowDroid only took a minute to find several security breaches for 500 real world applications, along with 1000 malware samples that were analyzed for 16 seconds per minute.

Bintaint [18] addressed the binary vulnerability mining problem using static taint analysis, so generating the Taint Control Flow Graphs (TCFG). This proposed tool is evaluated employing different compatibility levels of machines using X86 programs for different architecture embedded devices. The outcomes indicated that the proposed Bintaint framework is capable of defeating all the computational overheads of conventional methodologies, as well as addressing all vulnerability issues, without false negativity.

Precise-DF [19] is a novel static analysis method of detecting the taint flow in android apps, through the methodology of reusing the DidFail static analysis tool for fast modular analysis. This approach also uses the Boolean formulas for the DidFail's flow equations, which can help to record the conditions of the flow control for all possible taint flow programs. This approach is novel, as it does not depend on traditional taint flow analysis approaches to evaluate code and using reflection of apps. This method is more involved in providing security to the android apps up to the next level. Taint analysis to GDPR [20] formulizes how taint analysis can be stretched out and enlarged, in order to identify the likely unintended leakage of sensitive data. This study applies the standard static taint analysis methodology to detect any potential data breach, as well as the reconstruction of the flow for the data breach, with the aim of identifying how a breach could take place in relation to the flow. The results demonstrated that flows are not permitted by the privacy policies.

STAR [21] is a prototype designed to address the context-sensitive, flow-sensitive, and multi-source sensitive static taint analysis designed to track the information leaks in android apps. Its novel approach uses two concepts to achieve the performance and scalability of the analysis. The first approach employs the novel summarization technique, beneficial for analyzing the scale for the number of source APIs. The second approach combines techniques in order to establish an efficient propagation of the abstract states, both within and across the method boundaries. FastDroid [22] is a tool proposed as a security measure tool for android apps, being capable of providing efficiency and precision in the detection of sensitive data leaks. Three test suites were used to evaluate the performance of the FastDroid tool, with the results demonstrating high levels of precision and recalls, as well as effective efficiency in the results achieved. FastDroid differs from conventional approaches due to its technique of using propagation of taint values rather than the data flow values employed in traditional approaches. This resulted in improved efficiency.

COVA [23] this study offers a comprehensive level qualitative analysis for the evaluation of increased precision in static taint analysis. The study used the taint flows reported in FlowDroid [9] in 1,022 real world apps for android, with the results showing some key findings relating to conditions under which taint flows occur.

The analysis showed that specific settings (i.e. environmental setting, user interaction and I/O) are taint flows that are also involved in some specific conditions. BackFlow [24] is a context-sensitive taint flow reconstructor tool that builds paths linking sources to sinks. The results revealed that when BackFlow generates a taint graph for an injection warning, there is empirical proof that such an alert is a true alarm**.** DepTaint [25] implements a form of static taint analysis that analyzes the taint variables propagated by implicit flows and explicit flows. DepTaint greatly exceeds the static checker of LLVM in both defining taint variables and achieving more fine-grained pathways of taint propagation. ANTaint [26] improves scalability. An experiment involving 60 cases demonstrated that ANTaint is appropriate for 95% of cases, by extending the call graph and applying taint propagation on demand for libraries.

III. Analysis Method

When designing and implementing a static taint analysis for detecting malware, it is first vital to consider a robust and native Android anti-malware platform capable of running on smartphones rather than on a third-party device i.e. a computer or laptop. This requires a platform that is efficient modular, automated and static. Our platform consists of three main stages (De-Obfuscation – Data Flow Analysis – Fuzz Testing). During the first stage, we de-obfuscate all Android smart-phone-installed applications if their granted permissions touch our privacies, in order to obtain their source code. From this de-obfuscation stage, we go through the static analysis stage using taint analysis, i.e. a special type of data flow analysis. We then integrate the open source FlowDroid tool [9] as a module into our B-Droid platform, followed by double checking the results during the third stage, using a fuzz testing module for the Applications Under Test (AUT). This efficient platform can sandbox any doubtful applications, while instantly testing their leakage by placing them into a real and ideal environment and giving them fake privacy information. The platform then monitors any unacceptable behavior.

The design and implementation of the B-Droid is accomplished in four layers, as shown in Fig. 2.

- Layer 1: Permissions analysis layer:

This is the main module focusing on the following functionalities:

a) Reading all installed apps' AndroidManifest files.

b) Searching for permissions of interest inside AndroidManifest files.

c) Preparing the doubtful apps filter.

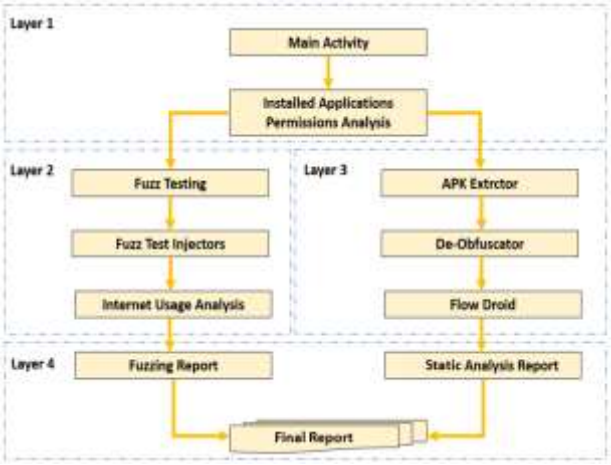

Fig. 2. Proposed Approach.

- Layer 2: Fuzz testing layer:

This layer is comprised of three fuzzing stages:

1) Fuzz testing: This is the base class for the fuzzing process, containing the parent attributes and methods of fuzz testing. The main task of the B-Droid is to detect Internet usage that demonstrates unexpected behavior for doubtful applications authorized to access the following permissions:

a) Receive SMS.

b) Process outgoing calls.

c) Read phone state.

These doubtful applications will certainly be authorized to access INTERNET permissions. The B-Droid test case scenario is organized to enable it to detect the misuse of any of the above granted permissions.

2) Fuzz testing injectors: This is the base class for the following injector classes:

a) Fuzz Received SMS: This class model is responsible for preparing a well-formatted fake SMS.

b) Fuzz Incoming Call: This class model is responsible for triggering a real incoming call.

c) Fuzz Outgoing Call: This class model is responsible for initiating a real outgoing call [17].

3) Internet usage analysis: This is the Internet usage measurement layer, which monitors the sent and received packet changes in a specific period for AUT.

- Layer 3: De-Obfuscator & Static Analysis:

This layer is comprised of three stages:

4) APK Extractor: This is the module that we developed to extract the APK file from our doubtful applications list transferred from layer 1.

5) De-Obfuscator: This is the module responsible for decompressing the APK file and retrieving its source code [27]:

a) Resource files (xml, text, icons).

b) Dex files (JAVA, JAR).

6) Flow droid: This is the malware static analysis module [9] that inspects the AUT code, in order to derive information concerning path behavior.

- Layer 4: Final Report Generator:

This layer is comprised of 3 stages:

7) Fuzzing report: This is the result obtained after fuzz testing successfully finalizes its mission on AUT and reports whether or not the application carries out malicious behavior.

8) Static analysis report: This is the malware analysis decision maker module, which generates a pass/fail report about AUT in response to any identified information leakages or program vulnerabilities.

9) Final report: This compares the two previous reports to conclude our work and clearly classify whether or not AUT is malware.

In general, a misinterpretation of a non-malicious activity as an attack by security system results in a “False Positive” error. These errors are a critical issue for today’s cybersecurity. The design of our anti-malware B-Droid platform (which uses both taint analysis and fuzz testing running separately on an AUT) will, as discussed in Section IV, decrease the false positive rate.

IV. Implementation Details

The following subsections discuss the structure and flow chart related to each layer.

A. Permissions Analysis and Filter Layer

This section illustrates the structure and flow chart of the permissions analysis and filter layer. Prior to an examination of the details of the flow chart, we must first note that we have selected a set of dangerous permissions (RECEIVED_SMS, READ_PHONE_STATE, NEW_OUTGOING_CALL) which can be requested to invade the user's privacy and access private data by any malicious application [28].

As shown in Fig. 3, the functionalities of this layer can be divided into two classes: firstly, main activity and secondly, permissions analysis.

The main activity class is the starting point of the B-Droid lifecycle. The main functions of this class are to characterize the application permissions risk and prepare the signatures of application permissions. These are mined in the AndroidManifest files of all installed applications. We then turn to the permissions analysis class functionalities, which appear in the third process. This process enabled us to sequentially read the AndroidManifest files of all installed applications, followed by searching for one or more permission signatures within our area of interest. The application is added to the doubtful list if the signature is found. This layer mechanism resulted in a list of all the installed doubtful applications that are granted one (or more) permissions of interest, in addition to the INTERNET permission.

Fig. 3. Permissions Analysis and Filter Flow Chart.

B. Fuzzing Injector Layer

This section examines the issue from the opposing perspective, i.e. broadcasting fake intents that perform as if real. Although there are an almost infinite number of possible inputs to any given application, our specific inputs or fuzzing injectors focused solely on calls and SMSs. We therefore prepared a real SMS, along with an incoming call and an outgoing call, which we termed injectors. These were then broadcast into the Android application layer to act as bait. B-Droid is able to identify whether AUT takes any of these baits.

Starting from the end of the previous layer, and after filtering all installed applications, the process commences when the user chooses any of the doubtful applications, i.e. AUT. The first process after selecting AUT is to obtain this application's signature, as outlined above. The signature(s) stimulate the B-Droid to prepare the matched injector(s) for the fuzzing process. The following points outline the structures of the three injectors participating in the fuzzing scenario.

1) SMS Injector:

- The received SMS injector is initiated by the previous layer if the AUT permission signature was RECEIVED _SMS. The first process is to create a PDU-formatted SMS [29] with content and other metadata.
- The second process consists of creating an implicit intent to be loaded with the SMS PDU message as additional data. This intent is simultaneously deployed with "android.provider.Telephony.SMS_RECEIVED". This action is used by most malware applications interested in SMSs and is coded in AndroidManifest. The SMS injector is then ready to simulate a real received SMS content, along with its intent.
- The final process in this injection is to broadcast the fake SMS. This is done by broadcasting the prepared intent to the Android application layer, which then informs the Android OS that a new SMS has been received within the message body.

2) Incoming call injector: This class implements the incoming call injector with the help of the telephone verification service. This service entails returning a call to a customer on the number provided, in order to verify that: a) the individual placing the order is the same as the owner of the phone and b) that the phone is indeed working. We used this service to perform an automated real incoming call injected into AUT by integrating Cognalys [30] Android API with our project. Cognalys provides a telephone verification service through a multi-platform package that application developers are able to use in their applications to check mobile phone numbers.

- The incoming call injector commences as a response to the AUT permission signature Read_Phone_State.
- The first step in the process was to request Cognalys' API service, by registering with the Cognalys server, then downloading and integrating its Android API with our Android project B-Droid. When registering with Cognalys, the user obtains an API key and an Access Token. These two entities are embedded into the verification call request, in addition to the cell phone number receiving the verification call.
- The second process was to process Cognalys' incoming call, i.e. informing the user that we were waiting for an incoming call by forcing B-Droid to run a waiting view until the incoming call was successfully received.
- Once the verification call had successfully taken place, the role of the third process was to read the response of this verification call, which contained a verification code and the result code. In our case, we were not interested in the verification code (i.e. our main task was to simply receive a real incoming call), but we were concerned with the result code, which informed us whether or not the incoming call was successfully transmitted. If the call is not received (which rarely happens), we would need to resend a new Cognalys request until the response result status code was returned successfully.

3) Outgoing call injector: We turn our attention to the final injector, i.e. the outgoing call. The simplest and most efficient means of carrying out a fully automated dynamic real outgoing call is to find a means of forcing the cell phone to call its number. If the user tries to call, then a real outgoing call is made for a period of 4 seconds, which results in the mobile operator giving a response of "busy number" and the call is automatically terminated. This led to our injector simulating a real outgoing call environment for AUT.

- As with previous injectors, the current injector is initiated as a result of the AUT permission signature New_Outgoing_Call.
- The first process is to prepare an implicit intent with the action of calling a phone number and loading it with a bundle of additional data for the dialed mobile phone number.

- The second process is to take the result of the previous process, (i.e. the prepared intent), which then initiates a new process with that intent, i.e. forcing the cell phone to call itself.

C. Internet usage Analysis Layer

This forms the comparator layer for each of the above layers, being considered a monitoring layer in the B-Droid application hierarchy. It is responsible for accounting the transmitted and received internet packets of AUT before, during, and following the injection or fuzz testing lifecycle. The results of this layer directly impact AUT's pass/fail report, i.e. it establishes whether or not this particular AUT is a malware application.

- The starting point is triggered automatically in a synchronous manner when any of the doubtful applications listed is selected by the user, thus entering the fuzzing scenario.
- The first process of this layer is to obtain and store the AUT traffic information (i.e. numbers of transmitted and received packets) prior to fuzzing. This information is comprised of the offset numbers the comparison will use to determine whether, by the end of fuzzing, it increases in number or not.
- During the fuzzing life cycle (approximately 15 seconds for each fuzz test case or injector lifecycle), the second process is run in the background to count any transmitted or received packets related to AUT's internet usage.
- By the end of the fuzz test case(s) life cycle(s), the third process amalgamates the previous process results with the first process offsets, storing them classified by the injectors' internet usage. It will then be fed back into the pass/fail report generator, as discussed in the next section.

D. De-Obfuscator and Static Analysis Layer

The de-obfuscator and static analysis layer inspect the AUT code to derive information about the app's behavior. In general, static analysis can check for programming errors and security flaws. However, our platform uses a taint analysis approach [9], i.e. a special type of data flow analysis. It follows a sensitive "tainted" object from source to sink, tracking the relevant tainted data along the path. Taint analysis can be used to find information leakages and program vulnerabilities, which form the focus of this paper.

1) APK Extractor: The Android Application Package (APK) contains the executable application installed on android phones or tablets. We therefore needed to implement a module on our B-Droid platform capable of extracting APK from the installed application. An APK Extractor module was designed and implemented for this purpose, capable of extracting APKs from the AUT list. The APKs thus obtained were stored in the phone's internal memory, ready for the next static analysis stage.

- Initiation of the application leads to processing of a list of all the applications installed on the device. Our APK extractor module employs the built-in classes PackageManager and ApplicationInfo to identify and retrieve all the APK files of the installed app.
- We accessed the AUT public source directory paths through our implemented APKExractor Class.
- We then converted these paths to an APK File Object, storing them in the phone's internal memory.

2) De-Obfuscator: This forms our module to decompile and extract the source code of an Android application (including XML files and image assets), JAR Packages and dex files, which work natively on our Android device. Generally, any Android application consists of 3 main components:

a) JAVA files: inside which the developer draws his/her picture.

b) JAR files: all external ready-made libraries the developer imports into his/her project to use its built-in classes and functions easily and fairly.

c) Resources files: in this case we have all xml files, layouts, media files, drawables and AndroidManifestFile.

From the above main components, we implemented three main decompiler classes in each one: JAVAExtractionWorker Class, JARExtractionWorker Class, and ResourceExtractionWorker Class [27].

3) FlowDroid: We used FlowDroid [9] as this implements a special technique for data flow analysis, known as taint analysis. Its procedure is to follow data along the programs' path of execution, which can be performed both forwards and backwards. A taint analysis keeps a record of data and its path from preset data sources to preset data sinks. It is designed with the objective of discovering a range of existing connections between provided sources and sinks. It is frequently used for security-relevant tasks. When the analysis is focused on the integrity of the application, untrusted inputs are specified as sources and should not reach sensitive sinks. Fig. 4 shows the different steps necessary for the analysis of our AUTs. Once the De-Obfuscation module is ready, FlowDroid [9] searches for call-back methods and lifecycles, as well as calls to sources and sinks in the application source code.

This is achieved by parsing different Android-specific files, such as the layout XML files, the dex files including the executable code and the manifest file that specifies the services, activities, content providers and broadcast receivers in the application. Furthermore, from the entry point list, FlowDroid [9] produces the main method. This is the primary approach for producing a call graph and an Inter-procedural Control-Flow Graph (ICFG). This detects all sources capable of being accessed from the given entry points. Starting at these sources, the taint analysis tracks taints by traversing the ICFG. This also introduces a function called Taint Wrapping, which can be used to substitute code unavailable for analysis, so as to

optimize performance. Finally, FlowDroid [9] reports all discovered flows from sources to sinks. The detailed information is provided in the final report module.

E. Final Report layer

The final destination in our approach is the report generator layer, in which we conclude our output results, drawn from the fuzzing and taint analysis modules. It is within this layer that we make and present our decision over whether or not the AUT presents malicious behavior.

1) Fuzzing report: The final stage in our fuzzing module is the pass/fail report generator, which is responsible for giving the Android user a clear report concerning AUT. In the case of failure, the report contains the leakage of privacies on which AUT has eavesdropped. In addition, it states whether the AUT passed the test without any leakage.

- The starting point commences automatically following the successful conclusion of the injection life cycle(s) for all injector(s).
- The output of the "Tracking AUT Traffic Information after fuzzing" process is requested from the previous processes to enable a comparison of the traffic information before and after fuzzing, and to calculate these changes for each injector during its lifecycle.
- If the AUT traffic information is found to have increased following the fuzzing lifecycle, it is written in the failure report as spy evidence, tagged by its permission signature. As an example, If the AUT traffic information increased during fuzzing with Received SMS, then it is written in the report as: This App Intercepts Your Received SMS, and the same for the other two properties (Incoming and Outgoing calls). If the traffic information is the same before and after the fuzzing lifecycle, then it will be checked as a clean app.
- The second process stores the results of the pass/fail report, sending them immediately to the user's notification bar.

2) Static analysis report: This analyzes the apps' bytecode and configuration files to find potential privacy leaks, as follows:

- It searches the application for lifecycle and callback methods, as well as calls to sources and sinks.
- It then generates the dummy main method from the list of lifecycle and callback methods. This is then used to generate a call graph and an inter-procedural control-flow graph (ICFG), as shown in Fig. 4.
- Starting at the detected sources, the taint analysis then tracks taints by traversing the ICFG.
- Finally, Taint Analysis reports all discovered flows from sources to sinks, including full path information.

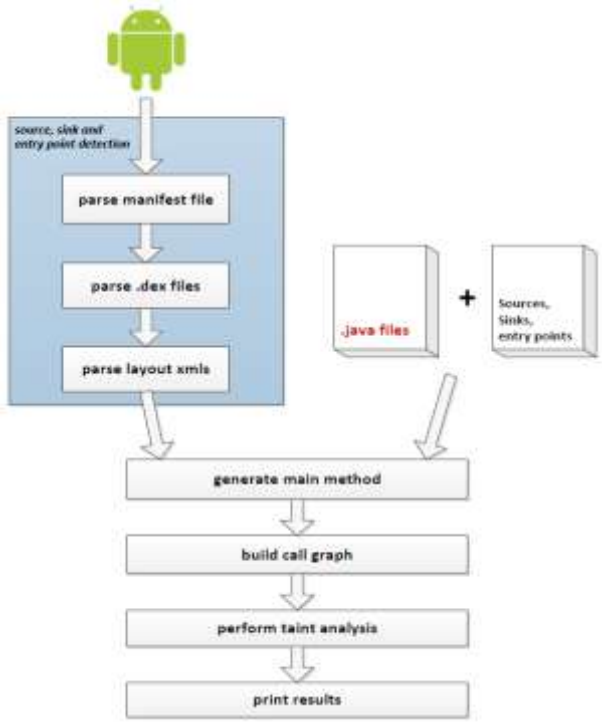

Fig. 4. Overview of Flowdroid.

V. Evaluation and Results

A. Results

We tested B-Droid against a dataset of over 100 Android applications uploaded on Google Play (or other third-party Android stores), as shown in Fig. 5. We selected a variety of application categories (Social Media Apps, Chat Apps, Caller Id Apps, and pure Mobile Remote Access Trojans (MRATs Apps)). As show Fig. 8, the results of B-Droid against Social Media Applications and Chat Applications were negative (pass) as well as for the Flowdroid tool. However, for Caller Id and MRATs Applications, the results were positive (fail) in fuzz testing and a few were positive (fail) in Flowdroid. The sample of Social Media and Chat Applications is summarized in Table I and the sample of MRATs and Caller ID Applications are summarized in Table II. These tables show the results of the comparison between B-droid and the prominent taint analysis tool Flowdroid.

- Evaluation 1:

The Caller ID Applications and MRATs we examined changed their Internet usage behavior during the fuzzing lifecycle, employing the available mobile Internet data, i.e. Mobile Data or Wi-Fi. Fig. 7 demonstrates that the number of bytes transmitted during the fuzzing lifecycle differed in AUTs. B-Droid reported that all these applications spied on information related to outgoing calls, incoming calls and received SMSs (see the sample of these results in Table II). Furthermore, the majority of MRAT vendors allowed potential customers to have a free trial of their spy product for 2–7 days. However, B-Droid detected that, even after the ending of the trial period, these free versions continued to transmit private data from the mobile phone.

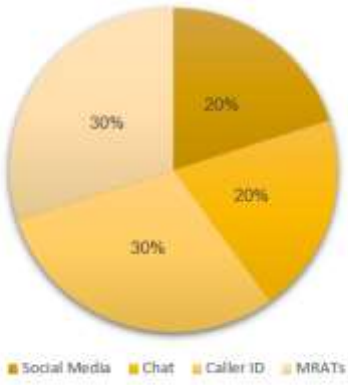

Fig. 5. AUT Flow Chart.

TABLE I. SAMPLE OF TESTED SOCIAL MEDIA AND CHAT APPLICATIONS

App Name	Installed Package	Store/ Provider	B-droid	FlowDroid
Snapchat	com.snapchat.android	Google Play	Pass	0 Leaks
Instagram	com.instagram.android	Google Play	Pass	0 Leaks
Messenger	com.facebook.orca	Google Play	Pass	0 Leaks
Twitter	com.twitter.android	Google Play	Pass	0 Leaks
WhatsApp	com.whatsapp	Google Play	Pass	0 Leaks
Telegram	org.telegram.messenger	Google Play	Pass	0 Leaks

TABLE II. SAMPLE OF TESTED MRATS AND CALLER ID APPLICATIONS

App Name	Installed Package	Store/ Provider	B-droid	FlowDroid
Android Auto	com.system.task	Xnspy.com	Fail	0 Leaks
Sync Manager	com.android.core.mngp	Snoopza.com	Fail	0 Leaks
Sync Service	com.android.core.mntq	Hoverwatch.com	Fail	0 Leaks
Setting	com.sec.android.as	my.a-spy.com	Fail	1 Leaks
Vibo Caller	com.vibolive	Google Play	Fail	0 Leaks
CallApp	com.callapp.contacts	Google Play	Fail	0 Leaks
True Caller	com.truecaller	Google Play	Fail	0 Leaks

[a] NOTE: (Fail= Positive Pass= Negative)

- Evaluation 2:

As shown in Fig. 6, the fuzz testing of the Caller Id Applications gave positive (fail) results for all the permissions of interest to B-Droid (Incoming Call, Outgoing Call, and Received SMS). End users can be sure that the scope of work of this type of application is simply to read incoming and outgoing dialled phone numbers and instantaneously, once it has access to the Internet, it works outside the phone to retrieve the names matched with those numbers as stored in cloud databases.

- Evaluation 3:

Compared to the popular Flowdroid tool, B-droid is able to detect leaks that the Flowdroid misses. As shown in Table II, static taint (Flowdroid) is insufficiently accurate with the clear malicious applications (MRATs) and the results showed that the majority of these malwares had no leakage (0 Leakage). Our contribution here is B-Droid, which can work hand-in-hand with Flowdroid to correct its weaker points, along with the development team of Static taint (Flowdroid) recommended a dynamic analysis technique work with Flowdroid to review its results. So, we see that static analysis and fuzz testing work hand in hand, such that they further strengthen their respective findings.

- Evaluation 4:

We have transferred our B-Droid model into a form useable by smartphone end users, enabling it to achieve the usability concept as shown in Fig. 9.

Fig. 6. Sample of Caller Id Apps Report.

Fig. 7. Sample of MRATs Apps Report.

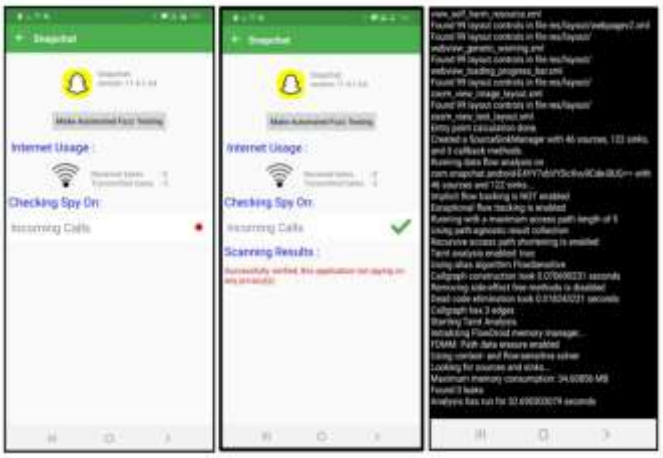

Fig. 8. Sample of Social Media App Report.

Fig. 9. B-droid Application Interface.

B. Limitations

Malware must be already installed on the mobile and running, this exposes our data to dangerous and frequent device failures. We must have a sim card and internet access to produce accurate results. In addition, due to the limitations of mobile memory, some applications are incapable of being decompiled.

VI. Conclusion and Future Work

The future of applications analysis lies in the Incorporation of several techniques all must work in tandem to reduce their respective weaknesses and turn their integration into strength. In our research, this is the approach we took.

This paper proposes an anti-malware platform, which we have called B-Droid. This is based on static taint analysis using source and sink techniques alongside the fuzz testing concept, in order to analyze the behavior of AUTs against internet usage during the fuzzing lifecycle. B-Droid enables us to test malware applications, examine the results and issue an immediate warning to the user, allowing him to decide whether or not to continue with AUT malware. B-Droid can carry out static taint analysis and fuzz testing on all installed applications after filtering, relative to their permissions. We tested our approach on a set of real-world apps randomly selected from the Google Play market (or other third-party Android stores), which resulted in identifying a number of leaks. Our results confirmed that a large percentage of Caller Id applications fail to implement appropriate security safeguards. B-Droid detected that all MRATs applications were spying on information related to outgoing and incoming calls and received SMSs, particularly those that were free. In addition, these free versions continued to transmit private data from the mobile phone following the ending of the trial period. Furthermore, B-Droid efficiently detected five of the top commercial spyware applications sold on the market.

In the future, we will focus on designing a cloud database to be connected to our B-Droid for storing all malware-detected applications and their attached information. This will be a valuable reference source for all researchers in this field. We also aim to work on implementing new injectors for fuzzing more important privacies (i.e. location, camera, call recording) and plan to test a large number of publicly available Android apps. In addition, we will focus on improving efficiency.

Acknowledgment

We would like to thank Dr.Mohammad al-zawawy for his useful comments and suggestions.

References

[1] Statista, “Market share of mobile operating systems in Indonesia from January 2012 to August 2020,” no. July, 2020, [Online]. Available: https://www.statista.com/statistics/262205/market-share-held-by-mobile-operating-systems-in-indonesia/.

[2] W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, “Android taint flow analysis for app sets,” 2014, doi: 10.1145/2614628.2614633.

[3] W. Enck et al., “TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones,” ACM Trans. Comput. Syst., 2014, doi: 10.1145/2619091.

[4] A. S. Bhosale, “Precise Static Analysis of Taint Flow for Android Application Sets,” 2014.

[5] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, “CopperDroid: Automatic Reconstruction of Android Malware Behaviors,” no. January, 2015, doi: 10.14722/ndss.2015.23145.

[6] L. K. Yan and H. Yin, “DroidScope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis,” 2012.

[7] R. Xu, H. Saïdi, and R. Anderson, “Aurasium: Practical policy enforcement for android applications,” 2012.

[8] W. Huang, Y. Dong, A. Milanova, and J. Dolby, “Scalable and precise taint analysis for android,” 2015 Int. Symp. Softw. Test. Anal. ISSTA 2015 - Proc., pp. 106–117, 2015, doi: 10.1145/2771783.2771803.

[9] S. Arzt et al., “FlowDroid : Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps.”

[10] L. Li et al., “IccTA : Detecting Inter-Component Privacy Leaks in Android Apps.”

[11] Z. Yang and M. Yang, “LeakMiner : Detect information leakage on Android with static taint analysis,” pp. 0–3, 2012, doi: 10.1109/WCSE.2012.26.

[12] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, “ScanDal: Static analyzer for detecting privacy leaks in android applications,” MoST, vol. 12, 2012.

[13] C. Gibler, J. Crussell, J. Erickson, and H. Chen, “AndroidLeaks: Automatically detecting potential privacy leaks in Android applications on a large scale,” 2012, doi: 10.1007/978-3-642-30921-2_17.

[14] I. Dillig, T. Dillig, and A. Aiken, “Precise reasoning for programs using containers,” 2010, doi: 10.1145/1926385.1926407.

[15] B. Amro, "Malware Detection Techniques for Mobile Devices," vol. 7, no. 4, pp. 1–10, 2017.

[16] A. Labade and H. Ambulgekar, "Fuzzing for Android Application : Systematic Literature Review," 2020.

[17] M. H. Saad, A. Serageldin, and G. I. Salama, "Android spyware disease and medication," 2015 2nd Int. Conf. Inf. Secur. Cyber Forensics, InfoSec 2015, no. 5, pp. 118–125, 2016, doi: 10.1109/InfoSec.2015.7435516.

[18] Z. Feng, Z. Wang, W. Dong, and R. Chang, "Bintaint: A Static Taint Analysis Method for Binary Vulnerability Mining," Int. Conf. Cloud Comput. Big Data Blockchain, ICCBB 2018, 2018, doi: 10.1109/ICCBB.2018.8756383.

[19] W. Klieber, W. Snavely, L. Flynn, and M. Zheng, "Practical precise taint-flow static analysis for android app sets," ACM Int. Conf. Proceeding Ser., 2018, doi: 10.1145/3230833.3232825.

[20] A. Mitrakas, K. Rannenberg, E. Schweighofer, and N. Tsouroulas, Tailoring Taint Analysis to GDPR. 2018.

[21] W. Choi, J. Kannan, and D. Babic, "A scalable, flow-and-context-sensitive taint analysis of android applications," J. Vis. Lang. Comput., vol. 51, no. October 2018, pp. 1–14, 2019, doi: 10.1016/j.jvlc.2018.10.005.

[22] J. Zhang, C. Tian, and Z. Duan, "FastDroid: Efficient taint analysis for android applications," Proc. - 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. Companion, ICSE-Companion 2019, pp. 236–237, 2019, doi: 10.1109/ICSE-Companion.2019.00092.

[23] L. Luo, E. Bodden, and J. Spath, "A qualitative analysis of android taint-analysis results," Proc. - 2019 34th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2019, pp. 102–114, 2019, doi: 10.1109/ASE.2019.00020.

[24] P. Ferrara, "BackFlow:Backward Context-Sensitive Flow Reconstruction of Taint Analysis Results," 2020.

[25] R. Ma, X. Wang, and X. Wang, "DepTaint : A Static Taint Analysis Method Based on Program Dependence," 2020.

[26] Y. Wu et al., "Scaling static taint analysis to industrial SOA applications : a case study at Alibaba," 2020.

[27] N. Rajendran, "decompiler for android.," pp. 1–9, 2020, [Online]. Available: https://github.com/niranjan94/show-java.

[28] Android Developers, "Manifest.permission," [Online]. Available: https://developer.android.com/reference/android/Manifest.permission.

[29] John Wiley & Sons, "Android Messaging," 2014.

[30] Cognalys Inc, "Cognalys Android Library (CAL)," [Online]. Available: https://cognalys.com/.