2017

Detecting Information Leakage in Android Malware Using Static Taint Analysis

Soham P. Kelkar
Wright State University

DETECTING INFORMATION LEAKAGE IN ANDROID MALWARE USING STATIC TAINT ANALYSIS

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Cyber Security

by

SOHAM P. KELKAR
B.E., Nagpur University, 2015

2017
Wright State University

WRIGHT STATE UNIVERSITY
GRADUATE SCHOOL

November 21, 2017

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Soham P. Kelkar ENTITLED Detecting Information Leakage in Android Malware Using Static Taint Analysis BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF Master of Science in Cyber Security.

_____
Junjie Zhang, Ph.D.
Thesis Director

_____
Mateen Rizki, Ph.D.
Chair
Department of Computer Science and Engineering

Committee on Final Examination

_____
Junjie Zhang, Ph.D.

_____
Adam Bryant, Ph.D.

_____
Yong Pei, Ph.D.

_____
Barry Milligan, Ph.D.
Interim Dean of the Graduate School

ABSTRACT

KELKAR, SOHAM, P. MSCS, Department of Computer Science, Wright State University, 2017. *DETECTING INFORMATION LEAKAGE IN ANDROID MALWARE USING STATIC TAINT ANALYSIS.*

According to Google, Android now runs on 1.4 billion devices. The growing popularity has attracted attackers to use Android as a platform to conduct malicious activities. To achieve these malicious activities some attacker choose to develop malicious Apps to steal information from the Android users. As the modern day smartphones process, a lot of sensitive information, information security, and privacy becoming a potential target for the attacker. The malicious Apps steal information from the infected phone and send this information to the attacker-controlled URLs using various Android sink functions. Therefore, it necessary to protect data as it can prove detrimental if sensitive data of the user gets leaked to the attacker. In this thesis research, we first discuss our static taint analysis framework used to track sensitive information flow from source to sink. We then study the relationship between the leaked data and URLs involved in the information leakage. The framework was tested on more than 2000 malicious samples to determine whether the samples leak information and the external URLs participating in the information leakage. The result shows that 30 percent of malware samples leak 24 unique Android sensitive information to around 330 suspicious URLs. We try to derive relations between the leaked data and the suspicious URLs to gain more intelligence on information security and privacy threat from information leaking malware samples. Finally, we conclude our research by discussing some various information leakage scenarios other than suspicious URLs. Our study raises awareness in both network security and information security domains where programmers fail to follow secure coding practices.

Contents

List of Figures

List of Tables

LIST OF LISTINGS

Abbreviations

SSA — Single Static Assignment

APK — Android Package

GPS — Global Positioning System

API — Application Programming Interface

URL — Uniform Resource Locator

IP — Internet Protocol

DEX — Dalvik Executable

IR — Intermediate Representation

Acknowledgment

I would like to thank my thesis advisor Dr. Zhang for the invaluable experience and guidance he provided me during the thesis research. His weekly supervision through meetings was a great asset as it helped me gaining research knowledge not only in my project but also discussing research methodologies used in projects of other students. I am incredibly grateful to him for the invaluable expertise and mentorship he provided me during the entire research study. His enthusiasm for Cyber Security and supportive nature has been great inspiration and motivation in conducting such high-value research in such hot topic.I would like to thank my committee members Dr. Adam Bryant and Dr. Yong Pei for their precious time and giving me supervision, feedbacks, ideas and sincere inputs in evaluating my thesis.I am grateful to Trustlook Inc for providing a valuable dataset of the malware samples for the research study. The samples gave me a baseline for the research and also helped to deduce essential findings.I would especially like to thank all my family members, friends, and roommates for their love and extreme support by providing positive encouragement and motivation throughout my master's studies.

Introduction

In this thesis, we propose a solution that can analyze the Android source code for information leakage detections. We first discuss static taint analysis framework to detect information leakage and then try to correlate the leaked data with different sinks. We also demonstrate the use of Single Static Assignment and the Environment Structure mechanism in our taint analysis framework and how it makes smooth and efficient tracking information flow in an object-oriented code.

1.1 Motivation

The advancing technology in mobility has attracted millions of smartphone users. Android is a Linux-based operating system developed by Google that nearly runs over 1.4 billion devices [1]. According to Statista's comparison for leading app stores [2], there are around 2.8 Million apps on Google Play store. The number indeed indicates that there is a huge demand for Android applications in the smart-phone market.Apart from Google play store there also third-party app stores and forums which distribute free Android applications. Most of these app stores do not have strict security checking on the uploaded App. Lack of security checks makes it possible for attackers to upload their malicious apps on the market and infect a large number of users. The modern smartphone is as capable as our laptops to perform a task. We can do all types of tasks such as email, listen to music, browse the internet and internet banking that we used to do on our personal computers or laptops. The sensitive data processed by the phone and applications running on the phone

are thus becoming vulnerable to the attacker. An attacker who wants to collect sensitive information from the target can develop malicious apps or modify the existing benign apps to harvest the data. These Apps then communicate with the attacker to send sensitive information from the infected device which poses a significant information security and privacy risk to the device owner. Therefore it is necessary to develop a framework which can analyze the malicious Apps to determine whether App leaks any sensitive information to the attacker. The more we can collect data from the malicious App, the more we can provide security measures to safeguard the sensitive data of the Android user.

1.2 Background

To develop a security framework which can analyze malicious binaries and detect information leakage we need to understand different methodologies. Below are some of the background information on various information security aspects and analysis methodologies

1.2.1 Information Leakage and its impact on the security

We can categorize information into sensitive and non-sensitive depending upon the context of the application. The latest smartphones are capable of carrying sensitive information associated with the smartphone users. Location coordinates obtained from GPS receiver can be used to mark the location of the user on the map. DeviceID can be used to identify a smartphone user uniquely. Network information related to Wireless network can tell wireless security of the access points. Most of the new Android smartphones come with an extensive range of built-in sensors. These sensors carry information which is personal to the user. In addition to data collected from hardware sensors, data stored in phones internal storage or external storage such as contacts, messages, phone logs, sim card information, camera photos can also reveal much information related to the user. If the sensitive infor-

mation of the user gets leaked, then the user is exposed to additional risks associated with the criticality of data, e.g., GPS coordinate can be used by an attacker to track user location. In one of the articles[3] terrorist groups spied on army location based on the information leaked by the malware. Wifi related information can be used by the attacker to determine the wireless security of the Android users operating environment. An attacker can use malware to leak sensitive information of the Android smartphone users to the malicious domain or logs.

1.2.2 Static Analysis

To analyze an Application for finding malicious behavior, we need to understand what are the instructions that are executed by the application. To follow these instructions we can reverse engineer the Application to suitable intermediate representation. The intermediate representation can be assembly level language or source code itself. Now that we have the intermediate representation or the source code of the application we can start analyzing the application by determining malicious instructions or applying machine intelligence to detect malicious intent of the App. As we do not have to run the application, we term this analysis as static analysis. The major advantage of performing the static analysis is that we are directly applying intelligence on the malicious code rather than just identifying malicious behavior. Moreover, many malware detects if the App is running in the virtual environment and changes their malicious activity which makes it difficult to analyze them. In the next section, we discuss Dynamic analysis methodology and its advantages and disadvantages.

1.2.3 Dynamic Analysis

Dynamic methods use the execution of methods to analyze the behavior of the software. The software can be analyzed dynamically by running it on a virtual machine and then observing the data generated using packet capturing tools. The main advantage of

using dynamic analysis over software is tracking runtime behavior of the app which is not possible in static analysis. The figure shows the network traffic captured by Wireshark of one malware sample. The data leaked is in plain text so we can see what type of data is leaked by the sample.

Figure 1.1: Information leakage capture by Wireshark

1.2.4 Why Static over Dynamic for taint tracking?

Capturing encrypted network traffic generated by the samples are hard to analyze for information leakage. Malware can encrypt the data before it leaks information to a malicious domain which makes dynamic analyzers challenging to detect information leakage. However, a static analyzer can use taint tracking to determine information flow from source to sink. In our research, we developed a taint analysis framework which leverages the benefits of static code analysis for detecting information leakage in Android malware samples.

1.2.5 Taint Analysis

Taint analysis is an information flow analysis technique to analyze data carried by different variables and objects in the program. When an instruction executes, the data associated with the variable is modified. By tracking the information flow, taint analysis can be used to detect data flow of sensitive information from source to sink. In our research context, we term sensitive information as a taint. We use code analysis technique to track the propagation of taint to determine the malicious behavior of the software. The following pseudo code 1.1 illustrates how taint tracking works.

Listing 1.1: Simple Pseudocode For Taint Tracking

```
1 var a  = getSensitiveInfoFromPhone();
2 var b =  a ;
3 var z = b +1;
```

Line 1 of the code use getSensitiveInfoFromPhone() to get sensitive information from the phone. We term sensitive information as taint, therefore the var a which stores taint information is now mark as tainted. To keep tracking the propagation, we can maintain table containing variables and their taint status in the memory. Line 2 assigns the value of a to variable b. The operation transfers taint value of a to variable b. The above operations is a simple representation of taint tracking. In our research, we developed taint analysis framework to handle complex data assignment operations. We now proceed to our next chapter which discusses our implementation of taint analysis framework.

1.2.6 Goal

In this thesis, we developed a taint analysis framework which runs analysis on Shimple an SSA representation of the Android source code. The framework should handle as much as possible the single static representation of the code and provide an accurate tracking mechanism for information flow within different variables. As the Android programming

is based on Java, the framework has to handle both APIs from Android and Java to support taint analysis over object-oriented language. There are a lot of input and output vectors for the information flow in object-oriented code. We focus on developing on framework capable of covering as much as information flow vectors by reducing false negatives and false positive results. The system is designed to track data to suspicious URLs. The framework should be capable of reliably detecting suspicious URLs and other sinks in information leakage. We relate the information to the suspicious domains after the taint analysis. We try to cover as much as possible the relation of leaked data to different sinks.

1.2.7 Organization

We have divided the thesis into different chapters. Chapter 1 discusses the background information on security impacts of information leakage, taint analysis and methodology used in the project. We review and discuss the related research studies in Android malware taint analysis by various security researchers in Chapter 2. Chapter 3 presents the design and challenges of our framework. We then discuss the implementation of the framework in Chapter 4. Chapter 5 discusses the results found and provide in-depth details of the correlation of leaked data with different sinks. Finally, we discuss the limitations of our framework, and possible future work needed to solve the shortcomings in Chapter 6.

Related Work

There are numerous studies which concentrate on taint analysis of Android applications. Most of them are based on static and dynamic taint analysis to detect information leakage. Steven [4]created Flowdroid a static taint analysis tool for Android applications. The Flowdroid uses Android lifecycle and context flow analysis to achieve taint analysis. Android applications have multiple entry points. The Android application uses four different types of components, activities for user actions, services for background tasks, content providers for storage and broadcast receivers for global events. These all components are registered in the AndroidManifest.xml file. The Flowdroid uses all these components to construct lifecycle of the application. Using the layout, the Flowdroid generates a dummy Main method and build a call graph as an input to the taint analysis engine.Flowdroid provides a good taint tracking mechanism from control flow graph, but it lacks taint tracking mechanism for variables declared as public static. These variables can be accessed from any part of the program, and so they can be used in taint propagation. Moreover, Flowdroid only discusses taint analysis and does not cover the sinks used in the information leakage such as URLs.

William [5] combined Flowdroid and Epicc analyzers to precisely track both inter-component and intracomponent data flow. They broke down the analysis into two phases. The first phase determines flows enabled by applications and the possible conditions for these data flows. The second phase takes the first phase as an input to enumerate more dangerous data flows by considering the application as a whole.This model also fails in

covering variables declared public static. So there is a high chance of false negative rate in the system.

Leakminer developed by ZheMin Yang [6] detects information leakage through static taint analysis. The Authors use the pre-processing methodology of the application to transform the DEX bytecode and loading of the metadata. The preprocessing involves converting DEX bytecode to Java bytecode and extracting app-rquired permissions from the metadata. The authors use these selected permissions to eliminate unuseful API calls to identify the legitimate method calls to the sensitive data. Using the transformed DEX code, they build a call graph detailing activity lifecycle callbacks and service callbacks. Now both sensitive data identified through permissions and call graph are given to taint analysis engine to detect information leakage.

Dflow and Dinfer are context-sensitive information type inference system developed by Wei Huang [7] to detect information leakage. The authors proposed context flow system and type inference system for taint analysis. The system first associate type qualifiers with the variables depending upon the taint status. The inference engine use typing rules set based solution and valid typing approach to detect if there are type errors. If there is no type error, then there is no flow from source to sink else it signals potential privacy leaks.

Anshul Arora [8] propose detection of Android malware by analyzing network traffic generated by the malware. The authors built a rule-based classifier for detecting Android malware. The authors used an emulator to run malicious binaries and create network traffic. The network traffic is then captured using TCP dump. The authors propose traffic features and rule-based classifier to identify malicious network traffic. The authors also discuss different risk levels depending on the characteristics matched.

Some research studies such as ScanDal by Jinyung Kim [9] discuss detection of privacy leaks in Android applications. The ScanDal uses path-insensitive, context-sensitive with a context depth and flow sensitive and flow insensitive hybrid analysis. The authors propose the semantics for Dalvik core an intermediate representation of the Dalvik byte-

code. Using the schematics and abstract syntax rules the ScanDal detects information leakage.

Xuetao Wei [10] developed Aura a system which can detect URLs an Android App communicates. The system uses both static and dynamic analysis to find URLs which App communicates. The static analysis component determines embedded URLs within the app bytecode. The dynamic analysis component detects URLs by capturing network traffic using tcpdump. The authors then categorize the URLs based on the malicious reputation of the URLs using VirusTotal.Although Aura can be used to detect URLs contacted by the malware sample, Aura does not implement information tracking mechanism for these URLs.

There are many research studies[11][12][13][14] which only focuses on Android permissions to detect malicious nature of the Android binary. Android permissions are the security feature provided by the Android to grant sensitive resource access to the application. If an App wants to access GPS location, then the App requires Location service permission to access GPS coordinates from the GPS receiver. Thus denying access to the sensitive resource through permissions can stop malware accessing sensitive information. Although these can be a useful starting point to identify malicious nature, many users tend to give permissions neglecting the security features provided by the Android operating system.

Although most of the studies focus on information leakage, none of them focus on the sensitive information leaked by the malware to the attacker-controlled URLs. From the information security perspective, it is necessary to detect both the leaked information and the malicious actor in the attack. Therefore in our research, we not only focus on the static taint analysis of information leakage but also focus on the information leaked to various malicious URLs and deriving the relationship between them.

Design

In this chapter, we describe the design of our Taint Analysis Framework. Our framework is made up of two modules. The first module is a reverse engineering component which converts android binary to intermediate representation and the second module perform static analysis on the decompiled files using the Parser. Figure 3.1 shows the architectural diagram of our taint analysis framework.

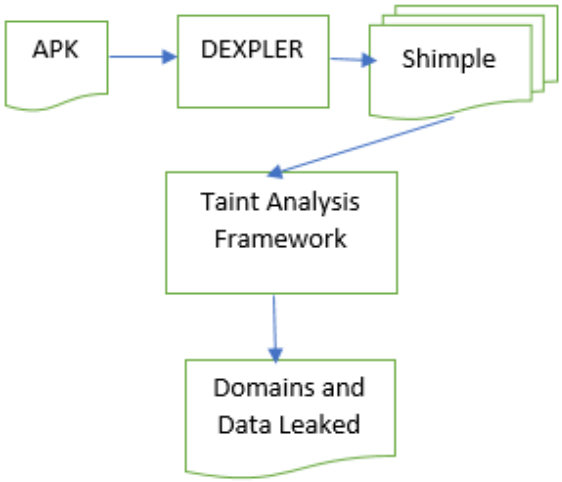

Figure 3.1: Simple Framework

The taint analysis framework is made up of following main components: 1) APK (Android Package File) 2) DEXPLER (Reverse Engineering tool) 3) SSA (Intermediate Representation) 4) Source Functions (Android sensitive source methods) 5) Sink Functions (Android functions responsible for leaking data) 6) Rules (Generic rules to handle

assignment operations of system defined functions) 7) Parser (Source code analysis tool) 8) Data leakage (Results)

3.1 APK (Android Package)

Android use APK (Application Package) format to install apps on the Android operating system. The Android Application Package (APK) is an archive which is a resultant of compilation of Android source code. Android programs are basically developed using Java programming language. To run the application, Android use the DEX file inside the APK which contains classes and methods necessary to execute the instructions. Apart from DEX file APK contains Androidmanifest.xml to contain additional information such as App information and access permission. To analyze the source code of the Android malware we pass the APK of the sample through reverse engineering tool to get intermediate representation of the source code. We now discuss the reverse engineering tool in our framework to decompile the Android Application Package into intermediate representation (IR).

3.2 DEXPLER

Although Android applications are developed in Java programming language they do not run on Java Bytecode but instead use Dalvik bytecode. The Java code is compiled to Java bytecode and then transformed to Dalvik bytecode using dx tool. To analyze Dalvik bytecode for static analysis, it first need to be converted into an intermediate form suitable for code analysis. Dexpler[15] a software package converts Dalvik bytecode to Jimple (IR). The Dexpler is built upon Dedexer (disassembler tool for DEX files) and Soot (framework for analyzing and transforming java and Android applications). Soot[16]converts Java bytecode to soots internal representation. The internal representation used in DEXPLER is Jimple which is three address intermediate representation suitable for optimization. In our research, we modified the DEXPLER to convert Dalvik Bytecode to Shimple (Single

static assignment version of Jimple). The next section describes Shimple (Single static assignment) and its usage in taint analysis.

3.3 SSA (Single static assignment) as Intermediate Representation

Single static assignment is a property of intermediate representation (IR) where each variable is defined exactly once. The uniqueness of variable names helps in taint tracking because the source code analyzers has to track information of variables on individual basis. We now discuss essential characteristics of single static assignment representation and how they can be used in taint analysis.

Listing 3.1: Single Static Assignment Variable Declaration

```
1 android.telephony.TelephonyManager $r8;
2 java.lang.String $r9, $r12, $r15, $r19, $r9_1, $r15_1, $r15_2;
3 char $c2, $c2_1, $c2_2;
4 android.location.LocationManager $r10;
5 android.location.Location $r11;
6 double $d0, $d0_1;
```

The above listing 3.1is a sample abstract of the single static form of the declared variables in the Android source code. From Figure 5 we can observe variables used by different Class objects. Lets discuss variables used by java.lang.String class. The java.lang.String class uses seven different variables having initial letter $. These variables are assigned value exactly once in the function where they are defined. For example, if $r9 is assigned some value in Function X, then the value of $r9 remains constant and is not changed till Function X cycle is complete. For taint analysis purpose this is an important characteristic as we just need only to track data contained in the variables.

Listing 3.2: SSA Variable assignment

```
1  $r11 = virtualinvoke $r10.<android.location.LocationManager: ...
       android.location.Location ...
       getLastKnownLocation(java.lang.String)>("gps");
2  $d0 = virtualinvoke $r11.<android.location.Location: double ...
      getLatitude()>();
3  $r9_1 = staticinvoke <java.lang.Double: java.lang.String ...
      toString(double)>($d0);
4  $d0_1 = virtualinvoke $r11.<android.location.Location: double ...
      getLongitude()>();
5  $r12 = staticinvoke <java.lang.Double: java.lang.String ...
      toString(double)>($d0_1);
```

Listing 3.2 shows the data assignment operation using SSA representation. We can observe that each assignment process use a separate variable for storing information. Thus, for taint tracking, this makes it easy as we do not have to consider external events which alter the information contained in the variable. [More]

3.4 Source Functions

Android Platform comes with different inbuilt APIs with their classes and methods. To access resources on Android phone, a call to system method is required. We researched on various system method calls which collect sensitive information from the phone and termed those system methods as sources. To give an example of source functions, getDeviceId() method can be used to get IMEI number of the device while getLastKnownLocaion() can collect GPS coordinates of the device.Listing 3.3 shows source function assignment operation.

Listing 3.3: Example of SSA represenation of Android source function

```
1 $r9 = virtualinvoke $r8.<android.telephony.TelephonyManager: ...
      java.lang.String getDeviceId()>();
2 $r11 = virtualinvoke $r10.<android.location.LocationManager: ...
      android.location.Location ...
      getLastKnownLocation(java.lang.String)>("gps");
3 $r14 = staticinvoke <android.os.Environment: java.io.File ...
      getExternalStorageDirectory()>();
```

We created a database of source methods to pass it to the parser so that parser can use those methods to search it in the decompiled code. The source APIs and their classes are taken from official Android Developers website[17]as show in Table 3.1, Table 3.2, and Table 3.3

Table 3.1: Source Functions

SOURCE FUNCTION	RETURN TYPE	INFORMATION CARRIED
getAllCellInfo	List	Information
getCellLocation	CellLocation	Cell tower information
getDeviceId	String	DeviceId
getDeviceSoftwareVersion	String	Software Version
getLine1Number	String	Sim1 number
getNetworkOperator	String	Network operator data
getNetworkOperatorName	String	Network operator name
getVoiceMailNumber	String	Voice Mail Number
getAllProviders	List	Network provider list
getLastKnownLocation	Location	GPS cordinates
getAltitude	double	GPS altitude
getLatitude	double	GPS latitude
getLongitude	double	GPS longitude
getSpeed	float	Device speed
getAccounts	Account[]	Account information
getAccountsByType	Account[]	Account information by type
getAuthToken	AccountManagerFuture	Authentication token
getPassword	String	Password
getPreviousName	String	Last user name information
getUserData	String	User Data
getAppTasks	List	Application Task Information
getDeviceConfiguration	ConfigurationInfo	Device configuration
getMemoryInfo	void	Memory information
getProcessInErrorState	List	Error process information
getRunningAppProcesses	List	Running process information
getRunningServices	List	Running services
getRunningTasks	List	Running taks

Table 3.2: Source Functions

SOURCE FUNCTION	RETURN TYPE	INFORMATION CARRIED
query	Cursor	Database table information
getPackageName	String	Package name
getActiveAdmins	List	Admin information
getApplicationRestrictions	Bundle	Application restriction info
getCurrentFailedPasswordAttempt	int	Failed password attempts
getDeviceOwnerLockScreenInfo	CharSequence	Lock screen info
getPasswordExpiration	long	Password expiration
getPasswordHistoryLength	int	Password length
getPasswordMaximumLength	int	Password maximum length
getPasswordMinimumLength	int	Password minimum length
getSystemUpdatePolicy	SystemUpdatePolicy	Update Policy Information
getUserRestrictions	Bundle	User restriction info
getWifiMacAddress	String	Wifi Mac Address
getConnectedDevices	List	Connected Devices
getAddress	String	Device Address
getName	String	Device Name
getRemoteDevice	BluetoothDevice	Remote Device Name
getDescription	ClipDescription	Clipboard info
getItemAt	ClipData.Item	Clipboard item
getColumnIndex	String	Database column information
getColumnName	String	Database table information
getColumnNames	String[]	Database table information
getAttachedDbs	List	Databases attached info
getPath	String	File path
getVersion	int	Version Info
acquireDrmInfo	DrmInfo	DRM info
getMetadata	ContentValues	Meta-data Information
getSensorList	List	Sensor List

Table 3.3: Source Functions

getCipher	Cipher	Cipher info
getSignature	Signature	Signature Information
getActiveNetworkInfo	NetworkInfo	Network Information
getAllNetworkInfo	NetworkINfo[]	List Networks Information
getAllNetorks	Network[]	List of all Networks
getDefaultProxy	ProxyInfo	Default Proxy Information
getDetailedState	NetworkInfo.DetailedState	Phone state info
getHost	String	Host Address
getPort	int	Port Number
getKey	SecretKey	Key
getCertificate	SslCertificate	SSL Certificate Information
getBSSID	String	Wifi BSSID Information
getIpAddress	int	IP address
getMacAddress	String	MAC address
getSSID	String	Wifi SSID Information
getConnectionInfo	WifiInfo	Wifi Connection Information
getScanResults	List	Wifi Scan Result Information
getInetAddresses	String[]	Network Inet Information
getHostAddress	String[]	Network Host Address
getSimOperator	String[]	Sim card provider
getSimOperatorName()	String[]	Sim card operator name
getSimSerialNumber()	String[]	Sim card serial number
getSubscriberId	String[]	Sim card Subscriber Info
getVoiceMailNumber	String[]	Voice Mail Number

3.4.1 Design Challenge

There are a lot of input vectors that can act as SOURCE for sensitive information. We try to cover as much as possible the SOURCE function from both Android and Java class methods. Many Android Apps store information into a local database including contacts and other details which can be queried by an App if it has appropriate permissions. The database tables of different Apps also act as SOURCE for our framework. The major challenge is the data represented in tables is unknow and varies from App to App. As

there is no particular SOURCE function for representing data in tables we consider any operation to the local database as sensitive information extraction from the smartphone. Many Classes have same method names, so we have only studied those methods which can act as SOURCE in information leakage.

3.5 SINK FUNCTIONS

Like source functions, Android Platforms comes with APIs which can be used to transfer data to external network or outside application memory space. A malware can use one of these system APIs to leak sensitive information to the outside world. We researched on various system method calls which outputs information outside application memory space and termed those system methods as sinks (e.g. put() method of org.json.JSONObject class can be used to store sensitive information in the form of Json object and leak information to the URL.Listing 3.4 shows an example of sink functions.

Listing 3.4: Example of SSA represenation of Android source function

```
1 virtualinvoke $r1_1.<android.content.Intent: ...
      android.content.Intent ...
      putExtra(java.lang.String,java.lang.String)>("pkgname", $r2_3);
2 staticinvoke <android.util.Log: int ...
      d(java.lang.String,java.lang.String)>($r3, $r2_2);
3 virtualinvoke $r1.<org.json.JSONObject: org.json.JSONObject ...
      put(java.lang.String,java.lang.Object)>("id", $r3);
```

We created a database of sink methods to pass it to the parser so that parser can use those methods to search it in the decompiled code for possible leaks. The sink APIs and their classes are taken from official Android Developers website as shown in Table 3.4.

Table 3.4: Sink Functions

SINK FUNCTION CLASS	SINK FUNCTION
android.media.ExifInterface	setAttribute
org.json.JSONObject	put
org.json.JSONObject	accumulate
org.apache.http.client.methods.HttpPost	execute
java.util.Map	put
org.apache.http.message.BasicNameValuePair	>
java.io.OutputStream	write
android.graphics.Canvas	drawText
android.graphics.Bitmap	setPixel
android.os.Bundle	putBinder
android.util.Log	int d
java.io.OutputStreamWriter	append
android.content.Intent	putExtra
android.telephony.SmsManager	sendTextMessage
android.content.Context	sendBroadcast
android.media.MediaRecorder	start

3.5.1 Design Challenge

There are a lot of output vectors that can act as SINK for sensitive information. We try to cover as much as possible the SINK function from both Android and Java class methods. We only cover the most popular SINK functions that are commonly used in Android and Java programming to transfer through different App functionality.

3.6 Rules

Other than source functions and sink functions Android Platform comes with inbuilt classes and methods that can have influence on taint propagation. As Android programs are developed using Java Programming language, Java class methods can also have influence on taint propagation.For taint tracking, we need to track various operations which affect

taint propagation. Table 3.5 gives a list of operation which can influence taint propagation. The taint status of the destination variable depends upon the taint status of the calling object as well as the return value of the function call.

Table 3.5: Rule Categories

NO	OPERATION	COMMENT
1	x= y	Simple data assignment operation.(e.g. if y is tainted then x is tainted)
2	X = y1.functionP()	Operation containing function calls tosystem methods with no parameters.(e.g. if y1 is tainted then x is tainted)
3	x = y1.functionP(a)	Operation containing function calls tosystem defined methods with parameters.(e.g. if a is tainted then x is tainted)
4	x = y1.functionQ()	Operation containing function calls touser defined methods with no parameters.(e.g. if y1 is tainted then x is tainted)
5	x = y1.functionQ(a)	Operation containing function calls touser defined methods with parameters.(e.g. if y1 is tainted then x is tainted)

We researched on common system methods used in Android programming and developed rules to check their influence on taint propagation. Table 3.6 shows sample example of the rule table

Table 3.6: Abstract of Rule Checking Mechanism

Function Name	Any Calling Object	Does calling object influence taint propagation	Indexes of Parameters that affect taint propagation
toString	Yes	yes	none
append	Yes	yes	any

We take an example to understand the logic of the rule features in column 2,3 &4.Listing 3.5 shows an example of taint tracking through rules.

Listing 3.5: Single Static Assignment Declaration

```
1 $r11=virtualinvoke$r10.<android.location.LocationManager:
2 android.location.Location ...
     getLastKnownLocation(java.lang.String)>("gps")
3 $r15=virtualinvoke$r11.<java.lang.StringBuilder:
4 java.lang.String  toString()>();
5 $r13_1=virtualinvoke$r13.<java.lang.StringBuilder:
6 java.lang.StringBuilder append(java.lang.Object)>($r15)
```

To understand the rule features we break Line 1 of the code into different parts of SSA statement. Listing 3.6 shows code breakdown of assignment operation.

Listing 3.6: Single Static Assignment Declaration

```
a.  \$r11:- destination variable\/object
b.  virtualinvoke:- interface call to object
c.  \$r10:- calling object
d.  android.location.LocationManager:- Calling Object Class
e.  android.location.Location: Calling Object subclass
f.  getLastKnownLocation:- Method\/function on which operation is ...
    performed
g.  java.lang.String:- return type of the function (9)< () :- ...
    contains parameter passed to the function
```

The first column in rule table 3.6 checks for Android system method in data assignment operation. The second column in the table 3.6 verifies if there are any calling objects. The third column checks if the calling object affects taint propagation. The fourth column checks for parameters passed to the Android function. Based on the parameters passed, the fourth column verifies the influence of the parameters on taint propagation. In (1) $r11 is assigned sensitive value from Android Source function getLastKnownLocation(). As $r11 now contains sensitive value we mark $r11 as tainted. We apply our rules in column2 and column3 to (2). Column2 checks whether (2) has any calling object. As (2) contains $r11 as calling object our rule evaluates to true. We then check if $r11 has any influence in taint propagation. This condition also evaluates to true as $r11 was tainted previously in (1). Therefore, we now mark $r15 as tainted. In (3), append() function used for data assignment operation. This append() takes one parameter $r11. The fourth column in rule table 3.6 checks if any parameters are passed to function influence taint propagation. Let us assume that $r13 in (3) is not tainted. According to our 2nd rule calling object has no influence on taint propagation for function append(). However, as the parameter passed i.e. $r11 is tainted therefore fourth column of rule table evaluates to true. Thus $13_1 is marked tainted. The rule table 3.6 discussed only covers non-sensitive methods in Android programming. The rule table helps speeds up the analysis process instead of creating environment for functions for analysis.

3.7 Parser Design

The goal of the parser is to correctly parse the single static assignment intermediate represenation of the Android source code. We develop Parser using Python which makes it easy to parse decompiled Shimple file line by line and interpret results conveniently. The parser uses the Regex string matching mechanism to parse the line of the source code into different parts needed for taint analysis. Following are the inputs to the Parser: 1) Shimple (SSA) file 2) SOURCE functions 3) SINK functions 4) Rule database

SOURCE, SINK, and Rule database act as core input to the parser in tracking information flow from source to sink. The main design challenge of the Parser is to develop taint analysis intelligence using best possible data structures to improve efficiency and reduce time complexity. The parser also implements URL tracking. The main challenge for the URL tracking is the URL obfuscation techniques used by the malware writer. In our parser, we have use data structure from external libraries such as Networkx[18]to develop information flow intelligence. Following are the advanced features supported by the parser: 1) Control flow analysis using Graph structure 2) Taint analysis of user defined function calls using Environment structure 3) Additional taint tracking mechanism for URLs

Implementation

The parser is the main component of our taint analysis framework. The parser is developed in Python to parse the Shimple files line by line and provide information flow analysis of different variables in the source code. The parsers leverage the previous inputs from source functions, sink functions, rules to track information flow from source functions to sink functions. We now discuss the subcomponents of the parsers used in tracking information flow in the source code.

4.1 Data Structure

To facilitate information tracking, we need a proper way to track the taint status of the variables. When a variable takes sensitive information, that variable gets tainted. When information of this tainted variable assigns to another variable, that variable also gets tainted. Thus in an entire function or method there can be a chain of tainted variables containing various sensitive information. When the static analysis program encounters the sink method, the program needs to determine whether sink method takes any tainted variable as input. To assess the status of the input variable, the program needs to check the taint status of the input and determine the possible information leakage. In our static analysis framework, we use bidirectional graph structure provided by Networkx a third party python library to store the status of the taint variables. Networkx offers different functionalities such as finding successor and predecessor of nodes. Using these features, a graph can be analyzed easily to determine the root and adjacent nodes. The following illustration shows how networkx

graph can be used to maintain taint status of the variables.

Listing 4.1: Sample Pseudocode to demonstrate Networkx

```
1 var1 = source(); // source is GPS coordinates
2 var2 = var1;
3 sink(var2);
```

In the Listing 4.1 var1 takes value from the source function. In the Pseudocode example, we assume the source is GPS coordinates. The var1 now contains sensitive information, i.e., the var1 is tainted. So we create a root node representing the variable var1 with attributes status = tainted and data= GPS coordinates. In the second line of the Pseudocode value of var1 is assigned to var2. The static analysis parser now checks if var1 is present in the graph and if its status is 'tainted.' If the parser finds var1 in the graph, the parser checks if the operation has any taint influence. As the var2 gets data from var1, there is taint propagation. Therefore the parser now adds a new node to the graph representing var2 as node and attributes as tainted. The parser then creates a bidirectional link between the var1 node and var2 node. The following figure illustrates the Networkx graph structure.

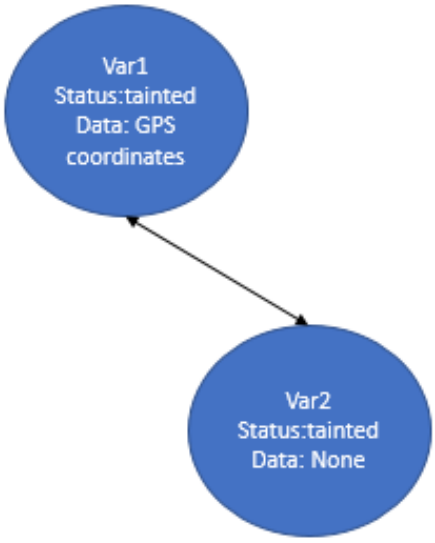

Figure 4.1: Example Networkx representation

(Note: If there is no new assignment of sensitive data then the node in the graph has attribute data=none). In the third line of the pseudocode, the parser encounters sink method which takes var2 as an input parameter. The parser now consults the graph to check if var2 is present. As variable 2 is present and the status of var2 is tainted, the parser analyzes the graph by backtracking the var2 node using predecessor functionality provided by the NetworkX of the var2 node. As the predecessor of the var2 node is var1, the parser extract the data associated with the var1 node. The parser now knows that the parameter var2 is tainted and it contains sensitive information (GPS coordinates obtained from var1 node).

4.2 Regex Strings to parse SSA representation

The parser uses the Regex string matching mechanism to parse the given line of the code. We use an online tool[19] in creating regex strings to aid in our analysis. Following are the sample regex string implementations for various source code implementation.

4.2.1 Detecting start of the Class Method

Detecting start of the function methods is necessary to update the data structure and its information. Most of the method names start with access identifiers (either private, public, static) The Listing 4.2 shows regex string to find access identifiers along with start and end of input parameters to the method.

Listing 4.2: Finding Function Methods

```
1 re.search("(?:(?:public\\b)|(?:private\\b)|(?:static\\b)|
2 (?:protected\\b)\s+)(.*)(\\()",line)
```

4.2.2 Regex Strings to search function name

Following regex string in Listing 4.3 is used to find function names in the code. The group 2 give the name of the called method. Using the function name we can verify whether the function is SOURCE function, a system defined method or a user function.

Listing 4.3: Finding Function Name

```
search = re.search('(.*)\s(.*)(\\()(.*)(>)', line)
if search:
    check_fun1 = search.group(2)
```

4.2.3 Regex Strings to search $Variable = $Variable assignment operation

The following regex string in Listing 4.2.3 finds simple variable assignment operation. It helps in the search for the source and destination variable in the information transfer.

```
$Variable = $Variable
var_var = re.match('(.*)(\\$)((?:[a-z][a-z0-9_]*))(\\s+)(=)(\\s+)
(\\$)((?:[a-z][a-z0-9_]*))(;)',line)
if var_var:
    desvar = "$"+var_var.group(3)
```

4.2.4 Finding Phi Statement and parsing the parameters in the Phi statement

Shimple uses ”Phi” Statements to represent control flow of branch instructions. Parsing the parameters to the ”Phi” statement gives an analysis of different variable assignment as a result of branching operation. Listing 4.4

Listing 4.4: Finding conditional statement

```
loop_assignment = re.search('(.*)(Phi\()(.*)(\)\;)',part2,flags=0)
if loop_assignment:
    varstring = loop_assignment.group(3).split(",")
    for var in varstring:
        var,label = var.split("#")
```

4.2.5 Finding public static variable name in data assignment operation in the code

A method can access public static variables declared in any part of the code which makes public static variable as a vulnerable source for sensitive information.Listing 4.5 shows regex string to find public static variables in the assignment operation.

Listing 4.5: Public static variables

```
var_psv_assignment = ...
    re.match('(\\$)((?:[a-z][a-z0-9_]*))(\\s+)(=)(\\s+)(\\$)
((?:[a-z][a-z0-9_]*))(\\.)(<)(.*?)(:)(\\s+)(.*?)
(\\s+)(.*?)(>)(;)',line)
if var_psv_assignment:
    psvt = ...
        var_psv_assignment.group(13)+"#"+var_psv_assignment.group(15)
```

4.2.6 Finding URLs and extracting domain name

Following regex string is a simple example of finding URLs within the source code and extracting domain name.Listing 4.6 shows regex string to find URLs embedded in the code.

Listing 4.6: Finding URLs in the code

```
search_url1 = re.search("(?P<url>https?://[^\s]+)", line)
if search_url1:
    try:
        parsedurl = urlparse(search_url1.group(0))
```

```
5           domainname = parsedurl.netloc
```

4.2.7 Finding passed parameters and destination variable in a function call

Following regex string finds different parameters passed to the function call and the destination variable. Applying taint analysis intelligence to parameters can track the influence of parameter in taint propagation. Listing 4.7 shows regex to find passed parameters to the functions.

Listing 4.7: Finding parameters to the function

```
1 var_static_invoke_assignment = ...
      re.match('(.*)(\\$)((?:[a-z][a-z0-9_]*)
2 (\\s+)(=)(\\s+)(staticinvoke)(\\s+)(<)(.*?)(:)(\\s+)(.*?)(\\s+)(.*?)
3 (\\()(.*?)(\\))(>)(\\()(.*?)(\\))(;)',line)
4 if var_static_invoke_assignment:
5      param_passed = var_static_invoke_assignment.group(20)
6      desvar = "$" + var_static_invoke_assignment.group(3)
```

4.2.8 Analyzing Source Functions and Information Tracking

Source functions act as a starting point for taint analysis. A malicious programmer can use these functions to extract sensitive information and store in variables. We term these variables as tainted variables as they contain sensitive information. When a variable is first initialized or assigned information from source function we call it as root variable as it is the starting point for taint analysis. The main design challenge to track and analyze information flow from these root variable is selecting best data structure to represent tainted variable. In our parser, we used object of NetworkX library in Python to represent tainted variable as a node in Graph. Representing a tainted variable as a node in Graph makes it easy to traverse graph nodes, access information in the nodes and apply special intelligence.

Determining successors and predecessors of graph nodes (in our case tainted variables) can help significantly in reducing complexity and increasing speed in taint tracking. We feed the parser with the source function database which parser can use to track sensitive functions in data assignment operations. The source function database contains the function names and the associated data. So, whenever the parser encounters the source function, it creates a node in graph (we call it as taint graph) with a variable name and data associated with source function. Following is the implementation of the taint graph to store tainted variables from source function.

4.3 Taint Propagation

In the first step of taint analysis, we determined root variables containing sensitive information and created nodes of each variable in the graph. Assignment operations can propagate taint from one variable to another variable. Therefore, we consider following test cases for taint propagation.

4.3.1 Simple assignment operation influence

$Variable2 = $Variable 1 where $Variable 1 is root variable(tainted)

The above assignment operation is the simplest form of taint propagation. $Variable 2 gets data from $Variable 1 which makes $Variable2 as tainted. The parser checks for such operations and adds the node to the Graph. In the above scenario, we add new node $Variable2to the graph and bi-directional link between $Variable1 and $Variable2. Adding bi-directional link makes it easy to analyze the graph by determining predecessors and successor of a node. Determining predecessor of $variable2 we can get all those nodes which are tainted.

4.3.2 Class object assignment influence

$Variable 2 = (Class) $Variable 1 where $Variable 1 is object variable of a Class

The above assignment operation is the representation of taint propagation from Class object variables. We check whether the object (we term all objects as variables containing some information) of a particular class in the taint graph. If $Variable 1 is in the taint graph, then we term above operation as taint propagation. Thus, we add $Variable2 in the taint graph and create a bi-directional link between the two variables. The reverse scenario is also possible where we assign some variable data to a Class object. [Code]

4.3.3 Public static variable as the tainted source

To access the data of a particular variable declared in different class we use public static variables. Public static variables thus act as a vulnerable source for taint propagation. There can be two distinct operations on the public static variables. 1) Assignment of data to the public static variable

2) Transfer of data via public static variable

In the first case, we check whether there is a propagation of sensitive data to the public static variables. We can use public static variables in the different class. During static analysis, we cannot determine what class operation triggers sensitive information to the public static variables. For example: Class A, Class B, Class C use the public static variable ”X.” In Class A, X assigns to some non-sensitive information or a variable which is not in taint graph (e.g. x = $Variable 1). In Class B, X assigns to sensitive information or a variable which is taint graph (e.g. x = $Variable 2). Class C use variable X to assign data from X to some other variable (e.g. $Variable 3 = x). An Android App can use multiple independent activities running simultaneously. So, it is not possible to determine which Class (A or B) can trigger value of public static variable X in Class C. So, we consider the worst case where Class B activity triggers taint propagation. Pre-processing of public static variables is necessary as the parsers check single file at a time. The pre-

processing function checks all occurrences of public static variables in decompiled files and data assignment operation. We create a new graph for the public static variables with nodes as public static variables. These nodes have attributes ”taint status” and ”URL” (discussed in the later sections). During the parsing of Class A, the parser adds variable ”X” to the public static variable graph and sets taint status as ”None.” When the parser encounters Class B assignment operation, it checks whether ”X” is in the graph and updates status only if there is taint propagation. Thus when parser starts checking Class C file operation ”$Variable 3 = x”, the parser knows what the status of variable X (tainted or not tainted) by looking into the public static variable graph. Depending upon the state of the public static variable the parser updates the taint graph.

4.3.4 Handling Branch instructions for taint propagation

Branching can induce taint propagation through different instructions executed in different branches. The Shimple (single assignment operation) simplifies the taint analysis on the branch instruction by representing entire branch instruction into single assignment operation using ”Phi” statements. Following test case from Soot's documentation helps us to analyze how soot handles data transfer operation during the transformation of Android source code to single static assignment. Listing 4.8 shows an example from soot official documentation [20] for conditional statements. Listing 4.9 shows Shimple representation of conditional statmenent.

Listing 4.8: Example from Soot Documenation for Conditional Statements

```
1 public int test()
2 {
3     int x = 100;
4
5     while(doIt){
6         if(x < 200)
7             x = 100;
8         else
```

```
9               x = 200;
10      }
11
12      return x;
13  }
```

Listing 4.9: Shimple (SSA) Represenation for conditional statements

```
1  The shimple output of the above code snippet :
2
3           i0 = 100;
4  (0)      goto label2;
5
6       label0:
7           if i0_1 ≥ 200 goto label1;
8
9           i0_2 = 100;
10 (1)      goto label2;
11
12      label1:
13 (2)      i0_3 = 200;
14
15      label2:
16          i0_1 = Phi(i0 #0, i0_3 #2, i0_2 #1);
17          if $z0 != 0 goto label0;
18
19          return i0_1;
```

We first try to analyze the source code. We can infer from the code that return value of ”x” is 100 if the value of ”x” is less than 200 and value of ”x” is 200 if the value of ”x” is equal or greater than 200. Variable ”x” thus takes two different values during the execution of the branch instruction. Now we try to analyze the Shimple representation of the source code for the branch instruction. Listing 4.10 shows phi representation of the conditional statement.

Listing 4.10: Phi Statement

```
1  i0_1 = Phi (i0 #0, i0_3 #2, i0_2 #1);
```

The Shimple use ”Phi” to track control of flow. The value ”i0_1” gets assigned to ”i0” or ”i0_3” or ”i0_2” depending on the control flow from different control statements indicated by labels. Thus ”i0_1” gets assigned value 100 or 200. Evaluating the ”Phi” statement helps us to analyze the possible data transfer from different branch instructions to the destination variable. For taint analysis, this is critical as different branch instructions can have an impact on taint propagation. We now consider a simple example in Listing 4.11 to understand taint propagation through various branch instruction.

Listing 4.11: Branch taint influence

```
1 $Variable 1 = sensitive information
2 $Variable 2 = non-sensitive information
3 if (condition):
4 (0) $Variable 3 = $Variable 1
5 else:
6 (1) $Variable 3 = $Variable 2
```

The ”Phi” statement for the above code is shown in Listing 4.12

Listing 4.12: Phi Statement for branch instruction

```
1 $Variable 3 = Phi ($Variable 1 #0, $Variable 2 # 1)
```

We consider OR Boolean logic for the variables in the ”Phi” statement. If either variable induces taint, then the resultant operation induces taint propagation. Thus, even if $Variable 2 does not contain sensitive information, $ Variable1 governs taint propagation. Parser thus add $Variable 3 into the taint graph and add bi-directional link between $Variable 3 and $Variable 1.

4.3.5 Taint propagation through system defined functions

Non-sensitive methods including Java class methods can also induce taint propagation. For example, a string object can have different operations such as append(), toString() to add or convert some value of a variable object. We use the rule database to check system methods which induce taint propagation. Let's take an example to understand the taint propagation through system defined methods, e.g., $Variable3 = $Variable1. append($Variable 3)

The above operation attaches values of $Variable 3 to $Variable 1. We know append() function attach data as it is system defined. So instead of executing the core instructions of append() function, we can directly infer that append() can influence taint propagation. Thus using rules for system methods, we can speed up taint analysis operation.

4.3.6 Taint Influence through User Defined Functions

Data assignment operation through user defined function calls can also affect taint propagation if the return value of the function returns sensitive information. We take following in Listing 4.13 the simple example to illustrate the taint tracking mechanism for user defined functions.

Listing 4.13: User Defined Function Call

```
1 $Variable1 = $Variable2.getIMEI();  where \$Variable2 is object ...
      of some user defined class and getIMEI() is the class method.
```

As getIMEI() method is user defined the return value of the method operation is unknown. To evaluate the user defined function calls we leverage the implementation of environment structure. Whenever the parser encounters user defined function calls, it creates an environment for the called method. The parser then analyzes the instructions in that function method and determines whether the method returns sensitive value. After the

parser examines method entirely, the parser creates a profile of that method and stores it in memory. The profile determines whether the function returns any tainted variable. So, next time whenever there is a call to the same function, the parser checks the profile of the function and determines if there is any taint influence. There are three possible taint influence cases in function call assignment as shown in Listing 4.14.

Listing 4.14: Single Static Assignment Declaration

```
1 $Variable1 = $Variable2.getIMEI();  where $Variable2 is tainted
2 $Variable1 = $Variable2.getIMEI(); where $Variable 2 is not tainted
3 $Variable1 = $Variable2.getIMEI($Variable3);  where $Variable 3 ...
     is tainted
```

In the first case (a) $Variable2 is user defined Class object which was previously tainted. In this case, we do not need to analyze the instructions within the getIMEI() method as the calling object influence taint propagation. In the second case (b) we require to investigate the code within the getIMEI() method to determine possible taint propagation. The third case is a particular case where getIMEI() method takes parameters. If getIMEI() method takes tainted input parameters then it is necessary to analyze the getIMEI() method as the method can use the input parameter to pass it to other methods or leak the information through sink functions.

4.4 URL Tracking Taint Mechanism

We added additional taint tracking mechanism for URLs. A malware writer can use hard coded malicious URLs or IP addresses in the code to send the sensitive information. The main goal of the URL tracking mechanism is handle as much as possible different ways an attacker can code to attach sensitive information to the URLs. The major challenge to this mechanism is the URL obfuscation which is difficult to track using static analysis.

We now discuss different test cases an attacker can use in the code to attach sensitive information

4.4.1 Test Case: 1 Appending URL to taint variable through append function

In the first case we need a data structure which can store variable name and the URL contained in the variable. We use the same Networkx graph structure we used for taint tracking to tracking URLs. Whenever Parser detects URL assignment, it creates a new node in the graph (we term it as link graph data structure in the code) containing the variable name and URL attached to the variable.Listing 4.15

Listing 4.15: URL through variable

```
1 $var1 = http://abc.com;
2 $var2 = getSensitiveInfo();
3 $var3 = $var1.append($var2);
```

After parsing the last instruction code, the link graph contains two nodes $var1 and $var3 and a bi-directional link between the two variables. Representing variables as nodes in graph simplifies the analysis and gives a good overview of the connecting nodes and their relation.

4.4.2 Test Case 2: Variable assignment

Listing 4.16: URL through variable

```
1 $var1 = http://abc.com;
2 $var2 = $var1
3 $var3 = getSensitiveInfo();
```

```
4 $var4 = $var2.append($var3)
```

4.4.3 Test Case 3: URL assigned through Public static variables

```
1 public static String x;
2 x = "http://abc.com"
3 $var2 = getSensitiveInfo();
4 $var3 = x.append($var2);
```

4.4.4 Test Case 4: URL assigned through user defined function

The compose URL function is the user defined function which prepares malicious URL and returns URL to the called function.

```
1 $var1 = composeURL();
2 $var2 = getSensitiveInfo();
3 $var3 = $var1.append($var2)
```

4.4.5 Test Case 5: Passing URL and taint variable to a Class method

The send data function takes two parameters: URL (or a variable containing URL) and the tainted variable. The method then uses both parameters to form JSON object to send it to the control server.

```
1 $var1 = "http://abc.com"
2 $var2 = getSensitiveInfo();
3 sendData($var1,$var2);
```

4.4.6 Test Case 6: Monitoring return values of the funiton methods

The return value can contain the URL composed by the function. So URL tracking mechanism also has to track the return value from the function. Function can either return URL directly or though the variable.

```
1 $var1 = "http://abc.com"
2 return $var1;
3
4 OR
5
6 return "http://abc.com";
```

4.4.7 Test Case 7: Monitoring parametes in the Phi Statement

Conditional statements can also assign URLs to the variables. Monitoring the parameters to the Phi statements can help to track such operations. It is possible that attacker can send the same information to different URLs depending upon the condition (status of information harvesting server)

```
1 if (condition)
2 (0) $var1 = "http://xyz.com"
3 else
4 (1) $var2 = "http://abac.com"
5
6 $var3 = Phi($var1 #0, $var2 #1)
7 $var4 = getSensitiveInfo();
8 sendData($var3, $var3)
```

Evaluation

In the previous sections, we discussed the static taint analysis parser to track the sensitive information flow. To determine the sensitive information targetted by the malware binaries and the sinks used in information leakage, we use the static taint analysis framework on a large dataset of the malicious Android binaries. In the first part of the Evaluation results, we discuss different sources and sinks used in information leakage by the malware families. The second part of the evaluation section discusses malicious domains, IPs, and various methods used by the attacker to exfiltrate data from the Android phone with the help of malicious binaries to the attacker-controlled URLs. In the final section of evaluation results, we try to derive essential relationships between the leaked data, malware family and the malicious URLs. The thesis work was evaluated using the malware samples provided by the Trustlook Company and some experimental Apps which leak sensitive information. The output of the program was compared with the network traffic generated by the samples to assess the accuracy of the experimental results. We now discuss the results generated by the framework.

5.1 Sensitive Sources Identified

After running our framework over large data set of malicious Android samples, we found 26 unique, sensitive sources in information leakage. The Table 5.1 gives a list of sensitive source identified in information leakage.

Table 5.1: Sensitive Information Identified

DeviceName	Wi-Fi ConnectionInformation
Packagename	FilePath
DeviceId	Simcardserialnumber
SimcardSubscriberInfo	Runningtask
Celltowerinformation	GPSaltitude
Databasetableinformation	WifiBSSIDInformation
MAC Address	PortNumber
Key	Devicespeed
Sim1 number	IPaddress
Network Information	GPSlatitude
SW Version Info	GPSlongitude
Simcardprovider	Wi-Fi SSID Info

Following is the sensitive information carrried by each SOURCE:

1) DeviceName: Device Name provided Android device user's name from configuration settings

2) WifiConnectionInformation: Wifi connection information about wireless configuration, strength and other details

3) Packagename: Information about the installed application package

4) FilePath: Information about the file location information

5) DeviceId: IMEI number of the phone. IMEI number is unique number to each device

6) Simcardserialnumber: IMSI number information

7) SimcardSubscriberInfo: Other sensitive details of the sim card

8) Running task: List of running tasks or applications on the phone

9) Cell tower information: GSM network information such as MCC, MNC, and LAC

10) GPS altitude: GPS location information

11) Databasetableinformation: Information stored in local SQL database. App can use local database to store and retrieve information.

12) WifiBSSIDInformation: Wireless BSSID information

13) MACaddress: Hardware address. MAC address is unique to the hardware

14) PortNumber: Port Number used by the application. Apps may use different port num-

bers to communicate with the outside world

15) Device speed: Android device speed details

16) Sim1number: Line1 number used by the Android user

17) GPS Latitude: GPS Latitude coordinates. GPS information can be used to track user's location on the Maps

18) GPS Longitude: GPS Longiutde coordinated. GPS information can be used to track user's location on the Maps

19) WifiSSIDInformaiton: Wireless Network Name

5.2 SINKS Identified

The static taint analysis of the malware samples identified sinks shown in Table 5.2 in the information leakage.

Table 5.2: Sink Functions Identified

org.json.JSONObject
android.util.Log
android.telephony.SmsManager
com.android.netutil.utils.HTTPUploader

Following is the description of the Sinks and the ways an attacker can use these sink for information leakage.

5.2.1 org.json.JsonObject

This sink has the largest detection rate in information leakage. or.json.JSON object is widely used by Apps to send to command and control server. Many Apps use HTPP connection class to set up the connection between client and server. Json object and URL are used as an input parameter to the HTTP connection.

5.2.2 android.util.log

Many app developers use Logs to dump runtime information. This error information then can be fetched to detect error and bugs. Due to poor coding practice, some apps can dump sensitive information via logs. This makes Logs a potential Sink as well as the SOURCE for the sensitive information.

5.2.3 android.telephony.SmsManager

The API can be used by Apps to send text messages over GSM. The same API can be used by malicious applications to send information to the attacker. Many SMS banking trojans use this API to leak sensitive information of the user.

5.2.4 com.android.netutils.HTTPUploader

This API can also be used to send data via URLs.

5.3 Suspicious Domains

To transfer data from App to the server, many Apps use hardcoded URLs in the source code. These URLs can be either domains or IPs of the control server. After analyzing the malware samples using static taint analysis, we found suspicious domains which accounted for 82 percent of overall suspicious URLs observed. The taint analysis helped us to analyze what information is attached to these domain names. In section V we discuss the sensitive data attached to these domains, section VI discusses the geographic location of the domain registration and section VIII discuss the relation of malware samples with the suspicious domains.Table 5.3 shows the identified Suspicious URLs.

Table 5.3: Suspicious URLs

www.toptools100.com	service.sm-adoss.com
ws.sd4face.com	adm.kemoge.net
interface.kokmobi.com	app.wapx.cn
interface.madpush.com	api.blueskysz.com
m.360dream.net	mmsc.monternet.com
s.iyd.cn	log.appsolo.net

5.4 Suspicious Ips

We also found hardcoded IP addresses in the information leakage. Out of the suspicious URLs analyzed, we found 11 percent suspicious IPs involved in information leakage. We also found local IP address which can also become a potential threat if other malicious apps use the local IP address to receive sensitive information. Unlike domains which can have multiple IP addresses or dynamic IP address, the hardcoded IP addresses in the malicious code are static IP addresses. Some IP address also has port numbers in the hardcoded URLs which we discuss in next section. It is important to note that the IPs can also be registered cloud services used by an attacker to gather sensitive information. In Section IV, we discuss the Cloud IPs observed in the URLs.

5.5 Suspicious Port Numbers

We found specific port numbers in the hard-coded URLs. The port number accounts to both suspicious domains and IPs. Unlike many domains which use default port 80 or well know port numbers to communicate we found port numbers which are defined outside well-known port numbers. The attacker can run any malicious service on these port numbers to gather sensitive information of the target. The Table 5.4 shows suspicious port numbers and their percentage count. Note the observed port numbers are TCP port numbers as our study is based on malware which use HTPP method to send information. It is important to note that some port numbers such as 8080 or 8088 can also be HTTP proxies. As an attacker can run malicious service to any port numbers, any specific port number does not count to any particular service. But, using the observed port numbers, we can filter our suspicious ones to reduce the attack surface and mitigate the risk.

Table 5.4: Suspicious Port Numbers

8088	8094	8023
8081	8091	8125
8888	8090	8007
9898	7004	8001
8000	18080	8002
8003	18081	8084
8079	9081	5222
4300	8061	8181
10003	9004	9884
9092	8004	7428

5.6 Using Cloud to collect sensitive data

Some cloud services offer free service to host personal web services. An attacker can use the free service to run multiple malicious web services to harvest information from the infected Android users. The additional benefit cloud can offer to the attacker is that most traditional firewall or ex-filtration systems only block malicious websites and allow legitimate domains including cloud services. From the URLs collected, we found 11 percent URLs as IP addresses. On further analysis of IP addresses, we found 60 percent of IPs registered to cloud computing services. Table 5.5 shows ISP registration of the IPs.

Table 5.5: Sample Cloud IPs observed and corresponding ISP

106.75.199.154	Shanghai UCloud Information Technology Company Lim
114.113.225.163	International Pioneering Park
113.31.81.147	BeiJing QianJingShiJi Co.
115.236.18.198	China Telecom Zhejiang
115.28.222.200	Aliyun Computing Co.
117.121.21.92	CNLink Network Technology
119.29.19.163	Tencent cloud computing
120.26.106.206	Aliyun Computing Co.
121.40.89.73	Aliyun Computing Co.
121.42.14.182	Aliyun Computing Co.

5.7 Top Domain Registrants for suspicious URLs

The Table 5.6 shows top domain registrants of the suspicious URLs.

Table 5.6: Domain Registration

Domain Registrant	Number of suspicious domain registered
IRT-CNNIC-CN	58
IRT-CHINANETCN	56
Amazon Technologies Inc	17

5.8 URL registration and Geographic location

We also studied the geographic locations of the suspicious URLs involved in information leakage. The table shows a breakdown of different geographic location codes and number of sensitive information sent to these areas. Although an attacker can register to any malicious service in the world, the geographic location information is necessary to derive intelligence on network traffic and block traffic if it matches a particular pattern. Table 5.7 and Table 5.8 shows which information leaked to different countries.

Table 5.7: Number of different sensitive source leaked to different geographic locations of URLs

Country Code	No. of Sensitive Source Attached
CN	24
US	19
SG	14
HK	7
VG	3
CZ	3
IE	2
CA	2
IN	2
AU	1

Table 5.8: Data Leaked to different Countries

China	United States	Singapore
Simcardprovider	Simcardprovider	Runningtask
Celltowerinformation	Celltowerinformation	Celltowerinformation
Databasecolumninformation	Databasetableinformation	DeviceId
Databasetableinformation	DeviceId	DeviceName
DeviceId	DeviceName	Key
DeviceName	GPSaltitude	MACaddress
FilePath	GPSlatitude	Networkoperatorname
GPSaltitude	GPSlongitude	Packagename
GPSlongitude	Key	Simcardprovider
IPaddress	MACaddress	SimcardSubscriberInfo
Key	NetworkInformation	VersionInfo
MACaddress	Networkoperatorname	WifiConnectionInformation
NetworkInformation	Packagename	WifiSSIDInformation
Networkoperatorname	Runningtask	
Packagename	Sim1number	
PortNumber	Simcardserialnumber	
Sim1number	SimcardSubscriberInfo	
Simcardserialnumber	VersionInfo	
SimcardSubscriberInfo	WifiSSIDInformation	
VersionInfo		
WifiBSSIDInformation		
WifiConnectionInformation		
WifiSSIDInformation		

5.9 Sensitive Data Attached to the URLs

In previous sections, we observed URLs involved in the information leakage. This section, discusses the sensitive information sent over these URLs. The study helps us to analyze more details of the source of the information and destination of information in the information leakage. In this paper, we mostly focused on the malicious URLs in the information leakage, but we also found other sinks in the malware samples. In the latter part, we discuss these sink to understand more details of the potential threat. The table shows a list of sensitive source targeted by the malware samples and percentage of suspicious URLs carrying these sensitive sources. The Table 5.9 shows sensitive information source targeted by the suspicious URLs. We can use the table information to determine the most popular and potential sensitive source. Note the table shows only the count for URLs involved in the information leakage. The number varies depending upon the features or category of the malicious samples. Studying these number, we can determine what sensitive information attacker is interested. Device Name and Package Name which accounts to large count gives specifics of the Android user and the malware details. The Simcard information provides GSM related information such as MCC, MNC and LAC information. DeviceId provides IMEI number of the infected phone which is unique to a particular device. Database table information is the information stored by the Apps in the local SQLite database. We use the local database as sensitive source as it can contain any sensitive information which apps can store. Note accessing SQL lite database of other Apps require special read permission. We assume the malicious samples have all required permissions to access sensitive sources. GPS longitude, altitude, latitude provides device coordinates for locating the device on the map. There are a lot of information security threat concerns with the sensitive sources which we discuss in section X of the paper.

Table 5.9: Percentage of suspicious domains leaking sensitive information

Data Leaked	% Leaked
DeviceName	65.68
Packagename	47.92
DeviceId	29.88
SimcardSubscriberInfo	21.00
Celltowerinformation	16.86
Databasetableinformation	13.90
MACaddress	11.24
Key	10.65
Sim1number	8.28
NetworkInformation	5.32
VersionInfo	4.73
Simcardprovider	3.55
Networkoperatorname	2.95
GPSlatitude	2.95
GPSlongitude	2.66
WifiSSIDInformation	2.07
WifiConnectionInformation	1.77
FilePath	1.77
Simcardserialnumber	1.47
Runningtask	1.18
GPSaltitude	1.18
WifiBSSIDInformation	0.88
PortNumber	0.88
Devicespeed	0.59
IPaddress	0.29
Databasecolumninformation	0.29

5.10 Malware Name and Sensitive Information

We also studied the malware sample categories used in information leakage. We used Virus Total[21]to extract specifics of malware samples such as malware family name. We then tried to relate specific sensitive source attacked by each malware family. The Table 5.10 shows the malware family name and number of sensitive source attacked by the malware family. The number of sensitive sources is the sum of all unique, sensitive source observed in malware samples belonging to a malware family. Note the count is not just

specific to URLs but also contains samples which leak information to other sinks. Relating number of sensitive information targeted by each malware families can help us to interpret malicious nature of samples in information leakage. Table 5.11 shows suspicious URLs connected by different number of malware families. We use this information in the section IX where we discuss the relation between malware family and suspicious URLs involved in information leakage.

Table 5.10: Malware Family and Type of Sensitive Information

Malware Name	#Sensitive Info
Android/Deng.VSD	21
Android/Deng2.BS	20
Android/AdWo	19
Android/G2P.MJ.615155967189	19
Android_ctl2.DOP	19
Android/Deng.GLW	19
Android_dc.ARTK	19
Android/G2P.BR.18C97926AF2F	18
Android/Deng.UPJ	18
Android/Deng.WEG	18
Android/Deng.GKW	18
Android/Fakengr	17
Android:Agent-KTX [PUP]	17
Android/G2P.BO.405F386BD6B4	17
Android_ctl2.SPC	17
Android/G3M.AB.38027EF69C4C	16
Android/Deng.LYF	16
Android/G2P.DD.AFBBC1192E24	16
Android/Deng.GMZ	15

5.11 Malicious Binary Type

Malware samples which leak information to suspicious URLs can have different sink count. We studied what percentage of malware samples leak a particular number of sensitive information to the URLs. The Table 5.12 shows binary type and percentage of samples

Table 5.11: My caption

suspicious URLs	Malware Family Count
oc.umeng.com	27
adm.kemoge.net	12
interface.kokmobi.com	12
www.toptools100.com	11
feedback.whalecloud.com	11
sdks.zy333.cn	9
se.nisemono.info	9
sdkj.zy333.cn	9
m.irs01.com	9
sdkm.w.inmobi.com	8
www.myapp.com	7
dock.inmobi.com	7
ad.flurry.com	7
alogus.umeng.com	7
alog.umeng.com	7
i.w.inmobi.com	7
ad.adspoo.com	6
app.wapx.cn	6
mmsc.myuni.com.cn	6
data.flurry.com	6

of that binary type. We found 12 percent of samples of the total samples which leak information to the suspicious URLs leak only one kind of information. We also calculated the percentage of samples which leak different kinds of information. The Table 5.12 helps us derive intelligence from samples leaking sensitive information.

5.12 Malware samples and suspicious URLs

A malware sample can connect to n number of URLs to leak information. Some samples have large count than others which make them more suspicious in information harvesting. It is possible that malware samples can connect to benign domains through inbuilt or 3rd party libraries. In this paper, we only consider the suspicious URLs the malware connects to leak information. The Table 5.13 shows the malware family names

Table 5.12: Malicious Binaries Type and Sensitive Information Count

Binary Type # of sensitive info	% Count
1	12.42
2	7.69
3	7.1
4	9.46
5	4.73
6	9.46
7	2.95
8	5.32
9	5.91
10	4.14
11	2.36
12	8.28
13	2.95
14	2.95
15	2.95
16	1.77
17	2.36
18	2.36
19	2.95
20	0.59
21	0.59

and number of suspicious URLs connected by the malware family. The reverse is also possible where multiple malware samples use same suspicious URLs to send information. The analysis gives the malicious nature of both malware family and suspicious URLs in the information leakage.

Table 5.13: My caption

Malware Family	No.Of Suspicious URLs
Android/Deng.UPJ	23
Android/G2P.BR.18C97926AF2F	21
Android/G2P.MJ.615155967189	19
Android/Deng2.BS	19
Android/Deng.VSD	16
Android/Deng.TC	14
Android/Fakengr	13
Android/Deng.ME	13
Android/AdWo	13
Android/Deng.WEG	11
Android/Mseg	10

5.13 Information Security Threat

Table III shows what type of information Android malware leaks to URLs. Studying the particular of the information is necessary as an Attacker can use information in different attacking scenarios. Device Name and Package Name at first seems legitimate for an App to send to the server. Android banking Trojan[22] is one of the Android malware family who use package name to identify the Package information installed on the infected Android user. The malware uses the Package name to detect compatibility and sends a fake log-in page to the infected device to steal other credentials. Cell tower information also acts as a sensitive source when it comes to spying or setting device configuration to undertake other attacks. TrojanSMS.Boxer[23] is one of the Android malware family who steals information related to Cell Tower such as MCC and MNC to retrieve configuration settings from command and control server to send messages to premium numbers and earn money. An attacker can extract file path information from the infected phone to retrieve the list of files under a particular directory. The file path information acts as a sensitive source where a user store files containing sensitive information to his device. An attacker can also use running task information[24] from the infected phone to check the running status of a

particular App on the infected phone. If the particular App is running, the malicious binary can download fake login page from command and control server and display an overlying window on the top of the App to trick victims into disclosing sensitive information.

5.14 Malware samples and information leaked

The previous sections all discussed measurements related to suspicious URLs as a sink in information leakage. There are different ways malware can leak information to the outside world other than URLs. Some of the other sinks could be through Android logs, messaging or broadcasting information through Android intent. After analyzing the samples through static taint analysis, we found x percent of samples use putExtra() a sink function used to broadcast intent information, y percent of samples used messaging to leak information, z percent of samples to dump sensitive information to the logs. Information security threat from these sinks is discussed in the later sections. The Table 5.14 shows the percentage of samples which leak different sensitive information. If we compare the table x with table y, we can observe the results are almost similar. The table more focuses on the sensitive source targeted by the malware rather than leaked to the suspicious URLs.

5.15 Using Logs to steal sensitive information

Another interesting case we studied while analyzing malware samples is 45.81 percentage of malware sample leak information to the logs. Many developers use Android.Util.Log API to log errors occurring in the execution. These errors can contain sensitive information related to the device. An App having read permission to logs can read sensitive information from the logs using the logcat command. The question we can ask is why malware samples would leak information to logs? Is it just poor programming practice or using logs as the Sensitive source for another malware.

Table 5.14: Malware Samples and Sensitive Information

Sensitive Data	%samples
Device Name	60.87
Database table information	60.87
Package name	51.67
DeviceId	27.09
Sim card Subscriber Info	20.74
MAC address	8.19
Sim1 number	7.19
Key	6.18
Cell tower information	6.02
GPS latitude	3.85
Version Info	3.18
Network Information	2.68
Wifi SSID Information	2.01
GPS longitude	2.01
Network operator name	1.67
Wifi BSSID Information	1.67
Sim card serial number	1.00
Wifi Connection Information	0.84
GPS altitude	0.50
Port Number	0.50
IP address	0.33
Running task	0.17
Device speed	0.17

5.16 Using Messaging to steal sensitive information

We also studied samples which leak information through Android.Telephony.SMSManager API. We found 4.71 percent of malware samples use Android.Telephony.SMSManager API as a sink in their information leakage. Existing network filtering mechanisms can stop malware leaking information through the internet but how do we stop malware samples leaking information through SMS raises a new information security concern.

5.17 Information Leakage Interesting Case

Ever wondered a camera picture could also leak sensitive information? We conducted a series of experiments where a malware inserts sensitive information into the metadata of the image. Many modern day camera applications embed GPS coordinates into the picture as a Geo tagging feature. Adding GPS coordinates to the image involves operation on the metadata of the picture. Metadata of the picture thus act as a potential sink where malware can add any sensitive information to the picture. We developed a malware sample which takes a picture and inserts IMEI and Contact number into the Exif data of the captured photo. Figure 5.1 shows how sensitive information can be embedded into the picture using EXIF attributes. Figure 5.2 shows the sensitive information IMEI number and Line1 number in the Author and Copyright section.

```
exif.setAttribute(ExifInterface.TAG_ARTIST,imei);
exif.setAttribute(ExifInterface.TAG_COPYRIGHT,line1);
exif.setAttribute(ExifInterface.TAG_GPS_LATITUDE,lat);
exif.setAttribute(ExifInterface.TAG_GPS_LONGITUDE,lon);
```

Figure 5.1: Setting EXIF attributes in malware sample

The experiment clearly shows that an image can also act as a potential sink in the information leakage.Although we did not find any traces in the malware samples provided, it remains a potential threat. The above screenshot shows the property details of the picture taken from our customized camera app. We can see from the picture that sensitive information IMEI number and Sim1 number attached to the image. We upload images to many social networking websites or store it in public cloud storage. Although some social networking sites remove EXIF data before they upload the picture to their servers, some sites retain the metadata information. This becomes a potential source for the attacker to extract sensitive information from the phone. Many exfiltration systems also fail in check-

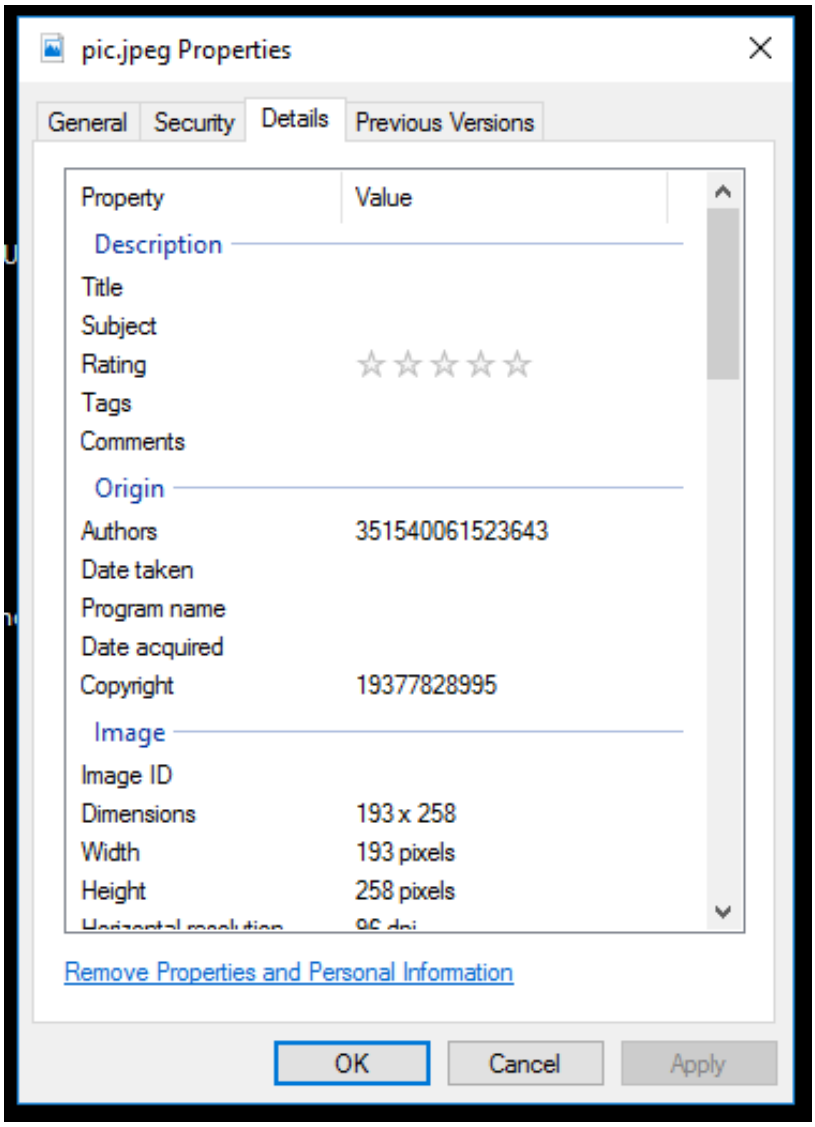

Figure 5.2: Image Capture Metadata Details showing IMEI and Line1 Number

ing the image as it appears to be legitimate. The taint analysis system we developed can successfully check such leaks by performing static analysis from source of the picture to the destination of the sink (here the EXIF data)

Discussion

The taint analysis system we developed shows an effective way of tracking information leakage in the Android binaries. Designing of the system was challenging, starting from selecting best intermediate representation for taint analysis and tracking the last data flow to the sink. The system we developed can be used on benign apps to detect information leakage vulnerabilities. A large number of developers do not follow secure coding practice guideline which leads to vulnerabilities in the application. According to CERT rules[25] for Android secure coding practice,dumping sensitive information to the logs can be detrimental. We conducted few experiments and found many applications do not comply with secure coding practice guideline. Many challenges that are not addressed in the static taint analysis such as URL obfuscation can be addressed in the dynamic analysis. The current taint system is only applicable for Android applications, but the methodology remains same for other smart-phone platforms also.

Conclusion

The thesis has discussed the static taint analysis to detect information leakage in Android binaries. We also presented several challenges in taint analysis and the ways malware can send information over different sinks. The thesis also addresses several problems in analyzing relations between the leaked data and the URLs. Unlike the prior approaches which only focus on specifics on information leakage or the suspicious domains contacted, we go further in analyzing the association of leaked data to several suspicious URLs and other sinks. We also discussed some interesting cases of how malware can leak information through different mediums such as pictures. With the help of Single Static Assignment representation in static analysis, we also demonstrated a simple taint analysis approach for Android applications. Implementing environment analysis in taint analysis is costly regarding time complexity, but it reduces the overall false positive rate. Although the research focuses on the malware binaries, the taint analysis framework can be used to detect information leakage vulnerabilities in the benign application.

Bibliography

[1] JOHN CALLAHAM. Google says there are now 1.4 billion active android devices worldwide, 09 2015. https://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide.

[2] Statista. Number of apps available in leading app stores as of march 2017, 2017. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[3] Danny Palmer. Hackers are using this android malware to spy on israeli soldiers, 2017. http://www.zdnet.com/article/hackers-are-using-this-android-malware-to-spy-on-israeli-soldiers/.

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

[5] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6. ACM, 2014.

[6] Yang M. AYang, Z. Leakminer: Detect information leakage on android with static taint analysis. *In Software Engineering (WCSE),Third World Congress,IEEE*, pages 101–104, 2012.

[7] *Scalable and precise taint analysis for android*, 2015, Author = Huang, W., Dong, Y., Milanova, A., Dolby, J. , Date = July, Pages = 106-117, Eventtitle = Proceedings of the 2015 International Symposium on Software Testing and Analysis.

[8] Garg S. Peddoju S. K. Arora, A. Malware detection using network traffic analysis in android based mobile devices. *Next generation mobile apps, services and technologies (NGMAST),eighth international conference,IEEE.*, pages 66–71, 2014.

[9] Yoon Y. Yi K. Shin J. Center S. W. R. D. Kim, J. Scandal: Static analyzer for detecting privacy leaks in android applications. *MoST*, page 12, 2012.

[10] Neamtiu I. Faloutsos M. Wei, X. Whom does your android app talk to? *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6, 2015.

[11] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[12] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012.

[13] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[14] Yang Wang, Jun Zheng, Chen Sun, and Srinivas Mukkamala. Quantitative security risk assessment of android permissions and applications. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 226–241. Springer, 2013.

[15] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[16] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[17] Google. Android developers. https://developer.android.com/index.html.

[18] Networkx. Networkx. https://networkx.github.io/documentation/networkx-1.10/tutorial/tutorial.html.

[19] txt2re. txt2re. https://txt2re.com/.

[20] Co P. Gagnon E. Hendren L. Lam P. Sundaresan V. Valle-Rai, R. A brief overview of shimple, 1999. https://github.com/Sable/soot/wiki/A-brief-overview-of-Shimple.

[21] Virustotal, 2017. https://www.virustotal.com//home/url.

[22] Symantec. Android banking trojan delivers customized phishing pages straight from the cloud, 2015.

[23] Pablo Ramos. Boxer sms trojan: Malware as a global service, 2012.

[24] Dinesh Venkatesan. How androids evolution has impacted the mobile threat landscape, 2012.

[25] Software Engineering Institute. Android secure coding standard, 2015. https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=161218578.