Bachelor Thesis

Enhancing a Static Analysis Framework for Android Security Analysis

Jonathan-Can Kleiner

783058

B.Sc. Wirtschaftsinformatik

Potsdam, XX.09.2019

1. Supervisor **Prof. Dr.-Ing. Christian Hammer**

2. Supervisor **Dr.-Ing. Abhishek Tiwari**

GLIEDERUNG – OUTLINE – TABLE OF CONTENTS (Inhaltsverzeichnis)

Right align a line: Select line → Format → Paragraph → Tabs → Enter Position, click right, click dots

Same for removal of right alignment because now indentation wont work

Abkürzungs-, Abbildungs- und Tabellenverzeichnis

SUMMARY

Abstract in German

This work has the focus on reproducing the paper “Slicing Droids...” published in 2012? They proposed a framework which does Slicing. But since Android versions, techniques and tools are rapidly evolving, these kind of frameworks becomes obsolete quickly. Therefore we enhanced SAAF and did an updated analysis on more recent Market and Malicious Android sample set.

Abstract in English

What is the problem / motivation?
What do we propose, what we are presenting in this paper which solves that problem?

This work has the focus on reproducing the paper “Slicing Droids...” published in 2012? They proposed a framework which does Slicing. But since Android versions, techniques and tools are rapidly evolving, these kind of frameworks becomes obsolete quickly. Therefore we enhanced SAAF and did an updated analysis on more recent Market and Malicious Android sample set.

TODO: text till line endings, test

1 Introduction

The impact that smartphones and other kinds of mobile devices have in our day-to-day life is undeniable. This impact is destined to become larger since the usage of mobile devices is rapidly growing. Among all mobile operating systems, Android has the largest global market share with 86% [1]. As of May 2011, 4.5 billion apps have been downloaded from the centralized application marketplace Google Play Store and over 200,000 apps were published [3]. In 2016, these numbers increased to 82 billion downloaded apps and in 2017, the number of published Android apps reached over 3.5 million [2]. There are also a significant amount of unofficial third-party marketplaces where a user has access to millions of Android applications besides the official Google Play Store.

Android relies on a permission-based security model where the user is asked if the application can have certain permissions, either before downloading the application or during runtime, when the app needs a permission to access a resource or execute a functionality. One downside of this model is that it might lead to an application with more permissions granted then they actually need, which is generally called a “permission gap” [4]. Another downside is, once a permission is granted, there are no other security mechanisms which prevents a malicious or vulnerable app to leak sensitive information or execute malicious instructions such as sending premium SMS messages.

This makes Android applications a perfect target for attackers with malicious intent.. According to Check Point Research, banking malware which became one of the most popular malware type in the recent years had a rise of more than 50% compared to last year [5]. These types of banking malware aims to steal credentials, transaction data or even transfer money from the victims account. Another newly discovered mobile malware is called “Agent Smith” and it already infected around 25 million devices[6]. It can replace already installed applications with the malicious versions using various Android vulnerabilities.

The number of malicious software is increasing with time and the only barrier for publishing new applications on the official Google store is the Google security system while the unofficial third-party market places usually does not have any security measures [7, 8]. Google Play uses the bouncer system, where an application will undergo security testing before it is allowed to be uploaded but even the Google Play security can be circumvented [10]. In 2017, Rahman et al. [9] analyzed 756 Google Play apps, 12% were detected as malware by at least one anti-virus tool and 2% were detected as malware by more than 10 tools.

Having these points in mind, it is crucial to develop efficient and effective ways to detect suspicious behavior patterns in mobile applications. These tools, which analyzes applications in an automated way can give us a detailed insight about the different parts of an app. This kind of tool can be then extended and developed even more to satisfy different needs of different domain requirements.

In 2012, a study done by Hoffmann et al. [11] proposed SAAF, a static malware analysis framework for Android applications which is capable of disassembling and analyzing Android applications in an automated way. This framework analyzes smali code with the help of apktool [12]. Smali code is a disassembled version of the Dalvik Executable (DEX) file which is used by the Android's Java virtual machine implementation. This form of disassembling is chosen over Java decoders because it is more robust in nature[13].

SAAF is a static source-code analyzer with the main focus on performing backward slicing on certain parameters of a given method to find out how the data flows inside the application. This way, a more precise understanding of the parts and functionality of an application can be achieved and more importantly suspicious behavior patterns will be detected in an automated way. Other analysis techniques are also part of SAAF, such as control flow graph visualization, ad-related code identification and an Android Manifest parser.

In their study, [Hoffmann et al.(?)] (we will use they or their study for the sake of brevity???) more than 136.000 benign market apps and around 6.100 malicious apps were analyzed by SAAF. They reported …:…...

The Android platform had major updates over the years which changed the internals of the platform. This led to state-of-the-art analysis tools not being able to analyze newer applications mostly because the usage of an older version of the apktool. A recent study, published in 2018 [14] shows, between the six prominent Android analysis tools evaluated, only one of them was able to analyze applications with an API level 19 or higher which is supported by 95,3% of all Android devices according to Android Studio. All the other tools crashed during analysis because older versions of apktool failed to decompile newer versions of Android applications. This was also the case with SAAF in our tests, therefore SAAF was not able to analyze any applications from our sample set, even the older versions.

In this thesis, we are revisiting the paper by Hoffmann et al. [11] with the goal of enhancing the static analysis tool SAAF for enabling its usage on never versions of Android applications, fixing present bugs, adding additional search parameters and reproducing their results using a newer Android application sample set containing 2000 benign Android market applications and 436 malware applications, collected between the years 2015 – 2017.

In summary, we present the following contributions:

- An updated version of SAAF presented in 2012 by Hoffmann et al. [11], a static analysis framework for Android applications with many of the major bugs fixed, able to analyze newer versions of Android apps.

- A replication study, with the updated tool we analyzed a sample set collected between the years 2015 – 2017 with 2000 benign market apps and 436 malicious ones, comparing our results to the results from 2012.

2 Background

In this chapter, we provide some background information about the Android platform, disassembling and analysis techniques.

2.1 Android Architecture Overview

Android consists of four types of app components, namely *activities*, *services, broadcast receivers* and *content providers*. Even though Android applications are generally written in the Java programming language, they don't have a main method which acts as the only starting point for a Java program, instead every activity can act as the starting point. An *activity* defines a user interface and it represents a single screen where the user can interact with the application. Many of these activities are used in an application to provide the user with a pleasant user-experience. *Services* are components without a user-interface which will run in the background for more resource intensive or long lasting computations. *Broadcast receivers* are used for receiving system wide broadcast messages sent from the Android system or other applications running on the device. The application that receives such a message can then react accordingly, for instance, limiting access to certain resources after receiving a broadcast announcing that the battery of the device is under a certain percentage. *Content providers* uses a storage mechanism to provide data for future access. This way other applications are able to access and modify the data if they have the required permissions.

Android's components communicate via a mechanism which passes messages called *intents*. Intents are messages sent between components with optional data attached to them. Components can communicate, trigger certain parts of other components, transmit data as well as send and receive messages from the Android System [15].

All crucial information that constructs the backbone of an application is declared in the manifest file which has always the name *AndroidManifest.xml*. This file contains for instance, which versions of android will support the app, what kind of permissions does the app require to fulfill all its functionalities, which components are a part of the app and how are they configured. Listing 1 shows a manifest file of a sample application.

As

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="de.example" android:versionCode="1" android:versionName="1.0" >

   <uses-sdk android:minSdkVersion="8" android:targetSdkVersion="17" />

   <uses-permission android:name="android.permission.SEND_SMS"/>
   <uses-permission-skd-23 android:name="android.permission.READ_PHONE_STATE"/>

   <application
      android:allowBackup="true" android:icon="@drawable/ic_launcher"
      android:label="@string/app_name" android:theme="@style/AppTheme" >
      <activity android:name="de.example.MainActivity" android:label="@string/app_name" >
         <intent-filter>
                  <action android:name="android.intent.action.MAIN" />
                  <category android:name="android.intent.category.LAUNCHER" />
         </intent-filter>
      </activity>
   </application>

</manifest>
```

Listing 1: Sample Android Manifest File

2.2 Android Permission System

Android provides an extensive API for developers to make use of different functionalities of the device or access sensitive user data, e.g. accessing location services in order to track your distance traveled, using the camera to take pictures and then immediately process them or accessing the user contacts etc. All of these API's are privacy- and security-relevant parts which are protected by permissions. If an application needs to access a certain protected functionality or data, it must first be granted the appropriate permission.

2.2.1 Protection Levels

Android categorizes permissions into four protection levels [16, 17]:

- **Normal**: These permissions are accepted as “low risk” permissions, meaning that the permissions reach and effect does not appose a high security risk. For instance, the permissions named INTERNET, BLUETOOTH and SET_WALLPAPER are inside the normal protection category, which are granted automatically when the user installs the application without explicitly asking for user approval.

- **Dangerous**: An app with these permissions can access private user data or control certain device functionalities which can be harmful for the user therefore dangerous permissions are not automatically granted. The user has to explicitly allow these types of permissions for a requesting application. READ_CONTACTS and SEND_SMS are examples for dangerous permissions. A malicious app with these permissions could potentially steal user contact information or send premium SMS messages.

- **Signature**: These types of permissions will automatically be granted to requesting applications which are signed with the same certificate as the application that defined the permission. These permissions are generally used to access shared data between applications developed by the same group.

- **Signature or System**: These permissions will be automatically granted only to applications inside the Android system image or with the same signature as the permission defining app. Generally used by device manufacturers or different vendors.

2.2.2 Install-time and Runtime Permission Requests

Since the first version of Android, the platform required the user to allow all the dangerous permissions (permissions which are contained in protection level “dangerous”) that an application might ever request at *install-time*. This meant an all-or-nothing situation where the user had to make a decision about allowing every single dangerous permission before even opening the app. The only way for app developers to

gain users trust was the app page on the Google Play Store. On this page, a user can read a longer description, look thru screenshots and videos about the application. If a permission seemed untrustworthy for the user, requiring too much access than necessary or access to sensitive information, the user could denied the permission request and the installation of the app was canceled [17]. maybe more...

This behavior was changed in October 2015 with the introduction of *runtime permissions*, a new feature in Android 6.0. (API level 23 or above). This was a significant change for the permission system. Users with devices supporting the correct Android version could now download applications without being asked to allow all requested permissions at once before the actual download. With this change users can download the application without getting a permission prompt screen, instead, when the application is running, each time the app needs a dangerous permission for any reason, the user is asked with an explanation from the developers. This allows developers explain their reason for requiring the permission. If the user denies this request, the part of the application that depends on that request cannot function anymore.

More infos about compiling to DEX, the smali format, apktool …

Static Analysis, backward slicing, dynamic analysis, taint analysis…

How the tool works in depth with examples

All the things that I fixed + updated, before I did the analysis

My analysis results compared to the one from 2012.

conclusion

related works?

Heuristic Patterns

asd

```
<heuristic-patterns>
        …
  <heuristic-pattern pattern="android/telephony/TelephonyManager->getDeviceId"
        type="INVOKE" hvalue="0" description="IMEI" active="true" />

  <heuristic-pattern pattern="android/telephony/SmsManager->sendTextMessage"
        type="INVOKE" hvalue="0" description="send SMS" active="true" />
        …
</heuristic-patterns>
```

Listing 2: Excerpt ? Heuristic Patterns

Backtrack Patterns

asd

```
<backtracking-patterns>
        …
  <backtracking-pattern active="true"class="android/telephony/SmsManager"
        method="sendTextMessage"
        parameters="Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
                PendingIntent;Landroid/app/PendingIntent;"
        description="Destination number of an SMS message" interesting="0" />

  <backtracking-pattern active="true" class="android/telephony/SmsManager"
        method="sendTextMessage"
        parameters="Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;Landroid/app/
                PendingIntent;Landroid/app/PendingIntent;"
        description="Text of an SMS message" interesting="2" />
        …
</backtracking-patterns>
```

Listing 3: Excerpt ? Backtrack Patterns

Permissions

```
<permissions>
        …
  <permission name="android.permission.ACCESS_CHECKIN_PROPERTIES"
        type="platform" description="" />
  <permission name="android.permission.ACCESS_COARSE_LOCATION"
        type="platform" description="" />
        …
</permissions>
```

Listing 4: Excerpt ? Permissions.xml

Bug Fixes?

Database Version

When setting up SAAF with a new version of a MySQL database, it threw this exception:
java.sql.SQLException: java.lang.ClassCastException: java.math.BigInteger cannot be cast to java.lang.Long

This was due to an incompatibility of the MySQL database version and the MySQL connector for java. Therefore we changed the version of our MySQL database to 5.7.27.

Apktool Version

SAAF 0.8 uses apktool 1.5.2 which was not able to decode any applications we tested it with??
In our tests, the only version which could consistantly decode every apk was apktool version 2.4.0, so we included it in the project. They also have a new branch with apktool 2.0 but that analyzed apps from 2012-2014 fine but also had problems with newer apps. Then we tried 2.4.0 and it worked with every apk that we tested with

Memory Space

This gave then another exception where the memory was not enough, the recommended was 1024? After tests we found that around 8000 was good enough

SQL key too long exception

The output which causes the exception can be seen below. It consists of 271 characters which explains the error. Since the field is set as VARCHAR(255), changed the value to 755, fixed it. In some cases the path was longer than the specified 255 string therefore …
Maybe paste the example which has 271 characters???

Generated Reports File Path

Fixed a bug where the relative path for each found heuristic and backtrack result could not be determined and therefore in the generated report.xml the file element was empty. It was throwing a lot exceptions because of that.This was due to a misuse of regular expression and file separators and is fixed.

Negative Array Size Exception

In rare occasions … we were not exactly sure about the reason but we built exception handlers that would handle the error and this was the analysis was not stopped.

Big Problem

Only the folder with the exact name “smali” was being … but with never versions min21 and above, also smali_class2? Was created where the actual data is being held, we fixed it so that the tool also takes those smali classes in the analysis.

Update to the tool

Updated backtrack and heuristic patterns

added new things because of new methods, new API with never versions, older version didn’t contain it…

** Show with another listing which other heuristic or backtrack patterns I added for the analsis

Add <uses-perm-sdk-23>

This was not tracked, since 2015 there is an option of <sdk-23> which does … explaind what uses-p-sdk-23 is used for, see SO tab.

https://stackoverflow.com/questions/35570533/difference-between-uses-permission-sdk-23-and-uses-permission

4 Evaluation

We used SAAF to analyze 2000 market apps randomly selected from the Google Play Store between 2015 and 2017. For our malicious sample set we analyzed 436 applications. The analysis was performed on a Dell Precision M4800 with Windows 10 installed, a 2.5 GHz Intel Core i7-4710MQ CPU and 32 GB RAM.

4.1 Analysis Results

In this section our analysis results are presented and compared with the results from 2012.

Permissions: The results of our new analysis and the old analysis can be seen in Table X and Table X respectively. It is important to note that we also included the newer permission declaration tag *<uses-permission-sdk-23>* which was introduced in with Android 6.0 (in 2015). The first thing noticeable while looking at both tables is that the percentage difference of the total permission requests between both tables. The new table has a significantly more permissions requested.

Malware	%	**Market**	%
INTERNET	93.45	INTERNET	86.20
READ_PHONE_STATE	77.51	ACCESS_NETWORK_STATE	52.08
SEND_SMS	63.69	WRITE_EXTERNAL_STORAGE	34.01
ACCESS_NETWORK_STATE	52.63	READ_PHONE_STATE	32.46
WRITE_EXTERNAL_STORAGE	48.07	ACCESS_COARSE_LOCATION	22.98
RECEIVE_SMS	40.32	ACCESS_FINE_LOCATION	22.42
RECEIVE_BOOT_COMPLETED	36.90	VIBRATE	18.05
READ_SMS	22.42	WAKE_LOCK	12.36
ACCESS_WIFI_STATE	21.45	ACCESS_WIFI_STATE	11.01
VIBRATE	20.36	CALL_PHONE	8.99

Table X: Top 10 Permissions Old

Malware	%	**Market**	%
INTERNET	98.83	INTERNET	96.86
ACCESS_NETWORK_STATE	91.84	ACCESS_NETWORK_STATE	89.27
READ_PHONE_STATE	83.68	WRITE_EXTERNAL_STORAGE	58.14
WRITE_EXTERNAL_STORAGE	77.16	WAKE_LOCK	39.35
ACCESS_WIFI_STATE	63.64	ACCESS_WIFI_STATE	34.33
WAKE_LOCK	62.00	READ_PHONE_STATE	30.40
RECEIVE_BOOT_COMPLETED	61.54	RECEIVE	28.94
GET_TASKS	49.42	VIBRATE	25.95
ACCESS_COARSE_LOCATION	44.76	ACCESS_COARSE_LOCATION	24.28
VIBRATE	44.29	ACCESS_FINE_LOCATION	23.08

Table X: Top 10 Permissions New

References – Need to be sorted alphabetically...

1. IDC 2019: Smartphone Market Share, https://www.idc.com/promo/smartphone-market-share/os (Last accessed on 1 September 2019)
2. Statista: Number of available applications in the Google Play Store from December 2009 to June 2019, https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (Last accessed on 1 September 2019)
3. Barra, Hugo: Android: momentum, mobile and more at Google I/O. Official Google Blog. Google. https://googleblog.blogspot.com/2011/05/android-momentum-mobile-and-more-at.html (Last accessed on 1 September 2019)
4. A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Automaticallysecuring permission-based software by reducing the attack surface: an application to android. https://hal.archives-ouvertes.fr/hal-00726196/document (Last accessed on 1 September 2019)
5. Check Point Research: Cyber Attack Trends: 2019 Mid-Year Report, https://www.ispin.ch/fileadmin/user_upload/partner/pdf/CP-mid-year-report-2019.pdf (Last accessed on 4 September 2019)
6. Check Point Research: Agent Smith: A New Species of Mobile Malware, https://research.checkpoint.com/agent-smith-a-new-species-of-mobile-malware/ (Last accessed on 4 September 2019)
7. (Malware is increasing 1) Name: A Survey on Smartphones Security: Software Vulnerabilities, Malware, and Attacks, NO NEED URL or last accessed FOR PAPER??? https://pdfs.semanticscholar.org/57ca/94653a5d440a7d5574b8d400f4e055eea7f5.pdf (Last accessed on 4 September 2019)
8. (Malware is increasing 2) NAME: Security Threats on Mobile Devices and their Effects: Estimations for the Future https://www.researchgate.net/publication/297746368_Security_Threats_on_Mobile_Devices_and_their_Effects_Estimations_for_the_Future (Last accessed on 4 September 2019)

9. (Analysis that Gplay security can be circumvented) Name: Search Rank Fraud and Malware Detection in Google Play https://poloclub.github.io/polochau/papers/17-tkde-googleplay.pdf (Last accessed on 4 September 2019), OTHER PAPER FROM SAME GUYS: https://epubs.siam.org/doi/pdf/10.1137/1.9781611974348.12
10. (Google play security was circumwented) Jon Oberheide and Charlie Miller. Dissecting the android bouncer. Sum-merCon2012, New York, 2012. NO URL/PRESENTATION
11. (Slicing Droids) Name: Slicing Droids: Program Slicing for Smali Code https://www.syssec.ruhr-uni-bochum.de/media/emma/veroeffentlichungen/2013/11/12/slicing_droids_sac13.pdf (Last accessed on 4 September 2019)
12. Apktool: A tool for reverse engineering Android apk files, https://ibotpeaches.github.io/Apktool/ (Last accessed on 5 September 2019)
13. (Smali disassemble is more robust) name: W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. AStudy of Android Application Security. In USENIX SecuritySymposium, 2011.
14. (do tools keep their promises) name: Do Android Taint Analysis Tools Keep Their Promises? (Last accessed on 5 September 2019)
15. Google Android Developers: Intent Documentation, https://developer.android.com/reference/android/content/Intent.html (Last accessed on 5 September 2019)
16. (android permission protection levels) Android DocumentaAtion, <permission> Tag: https://developer.android.com/guide/topics/manifest/permission-element.html#plevel (Last accessed on 7 September 2019)
17. (android permission protection levels) Android Documentation, Permission Overview: https://developer.android.com/guide/topics/permissions/overview.html (Last accessed on 7 September 2019)
18. (INTERNET downgraded from dangerous to normal) (delete) https://medium.com/finbox/android-runtime-permissions-recent-policy-changes-and-security-vulnerabilities-935c5fc88f3d
19.