UTS UNIVERSITY OF TECHNOLOGY SYDNEY

Analysis of the Latest Trojans on Android Operating System

by Baodi Ning

Thesis submitted in fulfilment of the requirements for the degree of

Master of Analytics

under the supervision of Dr. Yulei Sui
and co-supervision of Dr. Jingling Xue

University of Technology Sydney
Faculty of Engineering and Information Technology

January 2021

CERTIFICATE OF ORIGINAL AUTHORSHIP

I, Baodi Ning declare that this thesis, is submitted in fulfilment of the requirements for the award of Master of analytics, in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program.

Signature: Production Note:
Signature removed prior to publication.

Date: January 6th, 2021

Abstract

With the rapid advancements of electronics, the mobile operating system can accommodate various applications, which greatly facilitates people's everyday life. With a user group of more than 2 billion, the Android platform provides a diverse ecosystem for developing and publishing all sorts of applications. Although Google's official application store, Google Play, contains over 2 million apps, such a huge market also attracts hackers to make profits through distributing malware.

Mobile malware has rocketed since 2009. As reported by Broadcom Inc., an industry-leading security company, 2017 witnessed an increase of new mobile malware strains, compared with the year of 2016. Additionally, more profit-driven malware emerged with the growth of underground markets. Due to the fragmentation problem of the Android platform, Android has long been the most targeted operating system suffering from attacks. To keep pace with the cutting-edge anti-malware countermeasures adopted by cyber-security businesses, malware developers have abused high-level obfuscation, virtual environment recognition, conditional execution (logic bomb), run-time payload dropping, etc., to fool their opponents (i.e., security defending products and reverse engineering tools). These techniques are usually more obvious to trace during the evolution and diversification of a malware family. In this thesis, we take a close look into both recent Android trojans and one specific family of Android banking trojan, that infiltrates banking applications to steal credentials or trick victims to type in their usernames and passwords through displaying fake login interfaces. This thesis focuses on both statically reverse engineering the samples and dissecting the programs to understand their internal logic and find the similar features that could be used to assist security analysts, and dynamically monitor their behaviors in emulators. From public and private sources, 2380 samples of trojans from 20 (sub)families have been collected. As a result of the analysis, a lucid overview and improved apprehension of Android trojans are provided. The results indicate that Android trojans evolves towards possessing more malicious capabilities and more diverse permutations without losing their core design, which would cause more limitations and ineffectiveness for modern security solutions.

Acknowledgement

Throughout my research life pursuing a master's degree, I have received a lot of support from a wide range of people, both online and offline, some are friends or relatives while some have never been in contact.

First, I would like to sincerely thank my supervisor, Sui, whose expertise and resources are significant in both giving recommendations for revising the published articles as well as this thesis and informing possible journals and/or conferences to submit potentially publishable drafts. His insightful suggestions make my work more coherent and professional.

I would like to acknowledge my colleagues Guanqin Zhang, Zexin Zhong and Yanxin Zhang, who have helped me with reviewing drafts. I want to thank them all for their advice on perfecting the work to a higher level.

Also, I would like to thank some of the security professionals I have followed from Twitter, who are very resourceful and let me know many useful tools for both reverse engineering and analysis. Among them, @xabc0 a.k.a. Ahmet Bilal Can is the one that leads me into the world of banking trojans and malware analysis by his posts. @pr3wtd a.k.a. Witold Precikowski is the one who founded Apkdetect [1], and helped me with answering my questions when I encountered some weird code in the reverse-engineered source code. @LukasStefanko a.k.a. Lukas Stefanko also provided me with good ideas when I had some confusion when analyzing some malicious payloads of Bankbot Anubis.

In addition, I would like to thank my parents. I appreciate their encouragement when I felt frustrated or disappointed during my research. Their support made me through the plight. Moreover, this dissertation would not be completed without the company and communication from my friends, Wei Liu and Zhe, who have given me helpful suggestions and provided informative discussions based on their experiences of pursuing a PhD degree. Besides, the entertainment with them refreshed my mind from intensive research.

Finally, I would like to thank Bilibili as well as Youtube for providing records of great courses as well as amusing content for cheering and charging me up during my off-work hours.

Contents

1 Introduction **7**

1.1 Android operating systems . . . 7
1.2 Android malware and Android banking trojan . . . 9
1.3 APK Structure . . . 14
1.4 Research Objectives . . . 15
1.5 Thesis Organization . . . 16

2 Literature Review **17**

2.1 Android Trojan Evolution . . . 17
2.2 Android Malware Families Analysis . . . 17
2.3 Android malware detection . . . 18
2.3.1 Static Approaches . . . 19
2.3.2 Dynamic Approaches and Hybrid Approaches . . . 20
2.4 Android Malware Clustering and Classification . . . 21

3 Deep Analysis of Recent Android Trojans **22**

3.1 Data Collection and Extraction . . . 22
3.2 Attributes of Recent Android Trojans . . . 27

4 Analysis of the Bankbot Anubis family **40**

4.1 **Anubis's Approach** . . . 41
4.1.1 Phase 1: Mobile Devices Cyber Attack . . . 41
4.1.2 Phase 2: Privilege Escalation . . . 42
4.1.3 Phase 3: C&C Server . . . 44
4.1.4 Phase 4: Decrypting the encrypted . . . 47
4.2 **Data and Analysis** . . . 47
4.2.1 Anubis Roadmap . . . 48
4.2.2 Development of Encryption . . . 50
4.2.3 AndroidManifest . . . 50
4.2.4 Hard-coded API classes and methods . . . 54
4.2.5 Opcode Sequences . . . 57

5 An Observation for Android Malware Detection **59**

6 Conclusion and Future Work **64**

Appendix A Publication List **78**

Appendix B Figures **78**

List of Figures

1 Market shares of different operating systems for mobile phones[2] 8
2 Number of applications available on Google's official store[3] 8
3 Percentage of Android devices (i.e. Cumulative distribution in the screenshot) that the application could run on w.r.t the minimum SDK version (i.e. Android platfrom version in the screenshot) 8
4 Distribution of Android versions based on the cumulative distribution data . . . 9
5 Banking trojans detected by Kaspersky from 2015 to 2019 14
6 Dissected file structure of an APK file 15
7 The screenshot that the hashtags added by an analyst's comment 23
8 The screenshot of the first five samples when searching with "tag: trojan" on Koodous platform 23
9 The workflow of generating family names of Android trojans through utilizing the Koodous platform 24
10 The captured brief information that matches the family name (e.g. Anubis) input in the search box in Apkdetect 25
11 The captured detailed information that either matches or is somehow relevant to the family name (e.g. Anubis) input in the search box in Apkdetect 25
12 The workflow of using Apkdetect for generating configuration information of the samples 27
13 Anubis Attack Procedures 41
14 Infection Process 42
15 Code Injection in Forged App 43
16 Interface of Anubis Accessibility Extraction 44
17 Anubis C&C Server Model 45
18 The breakdown of the Anubis samples regarding configurations and versions . . . 48
19 Emergence of different versions of Anubis from January 2018 to July 2019 49
20 Top 4 frequent API classes used by four versions of Anubis 56
21 The name of the newly implemented module for new versions 57

22 Top 3 frequent (including overlapping) documented Opcode sequences of four versions of Anubis . . . 58
23 The workflow of identifying suspicious and unsuspicious APKs . . . 61
24 The AUC-ROC curve of the results . . . 63
B.1 ”3458” in Rotexy . . . 78
B.2 ”393838” in Rotexy . . . 78
B.3 How Anubis displays toast . . . 79
B.4 How Anubis tailors the text based on the language . . . 79
B.5 Hard-coded ”Enable access for” in different languages . . . 79
B.6 Base64 decoded image from the source code of most higher versions of Anubis . . 80
B.7 A sharper image from double Base64-decoded source code snippets of an Anubis dropper . . . 80

List of Tables

1 Differences between different types of malware . . . 10
2 The timeline of 20 (taking sub-families into account) Android trojans in the collection . . . 26
3 The (abbreviated) dangerous and sensitive permissions requested by recent Android trojans . . . 29
4 An overview of recent Android trojan (sub)families in regard to the categories of permissions requested . . . 30
5 The (abbreviated) intents that are of interest to malware authors to trigger trojans 32
6 An overview of recent Android trojan (sub)families in regard to the categories of intent actions used . . . 33
7 An overview of recent Android trojan (sub)families in regard to the anti-analysis techniques . . . 35
8 An overview of recent Android trojan (sub)families in regard to the persistence techniques . . . 37
9 An overview of recent Android trojan (sub)families in regard to the communication methods with C&C . . . 39
10 Frequently used permissions by Anubis . . . 52
11 Shared intent filters by all versions of Anubis . . . 53
12 API classes abused by different versions of Anubis . . . 55

13 Top 3 documented API methods that are the most different in terms of their average frequency 56
14 Seven types of Dalvik Opcodes 57
15 The statistics of the APK samples in the dataset 59
16 The confusion matrix of the results 62

1 Introduction

1.1 Android operating systems

With the rapid improvement of integration techniques for microelectronics, mobile devices have gradually become an indispensable part of people's daily life, not only because of the portability but also thanks to the diversity of mobile software (a.k.a. applications) that people can have access to. As mobile devices integrate more convenient applications that can tackle real-life problems, people have become extremely dependent on such small devices, e.g. social networking, news reading, video & audio playing and watching, digital marketing, etc.

Google declared in 2019 [4] that more than 2 billion Android devices were active. In addition, Fig. 1 shows that Android has started to become dominant in the market share against other systems since 2012. The market share of Android came to its peak between 2018 and 2019, with more than 90 percent. The ever largest market share of either iOS or Symbian is less than 45 percent. In line with the popularity of the Android operating system, Fig.2 reflects an upward trend regarding the applications published by Google Play. The statistics of available applications in the official store, Google Play peaked at March 2018, with 3.6 million. Although the statistics fell from 3.6 to 2.6 million from March to September due to some policy, the volume of applications in Google play has never decreased since then. The wide acceptance of the Android system can be attributed to the open-sourced trait of its source code, which facilitates the API (short for Application Programming Interface) calling, testing and debugging process for developers. However, the side effect of such property stands out due to the severe fragmentation issues. Figure 3 is the screenshot of the distribution dashboard that comes from the official Integrated Development Environment (IDE) in April 2020, Android Studio for Android developers when a new project is created and click on the "Help me choose" link under the minimum SDK (short for Software Development Kit) dropdown. The cumulative distribution data reflect the percentage of devices that your application could be run on, according to the minimum SDK version selected. Based on these statistics, the distribution for each version is shown in Fig.4. Only less than 10 percent of Android mobile users have upgraded to the latest version, which makes modern anti-virus engines insensitive to those attacks that are aimed at older versions [5, 6, 7, 8, 9]. Although the OS framework fragmentation prevents general exploits from hackers, different vendors of Android OS customize their ecosystem by adding different native libraries, which malware authors can abuse to evade the generic detection of anti-virus engines [7, 8, 10, 11, 12, 13, 14, 15, 16]. Moreover, the speed for OEMs (Original Equipment Manufacturers) and their vendors to fix previous vulnerabilities is slow enough for attackers to

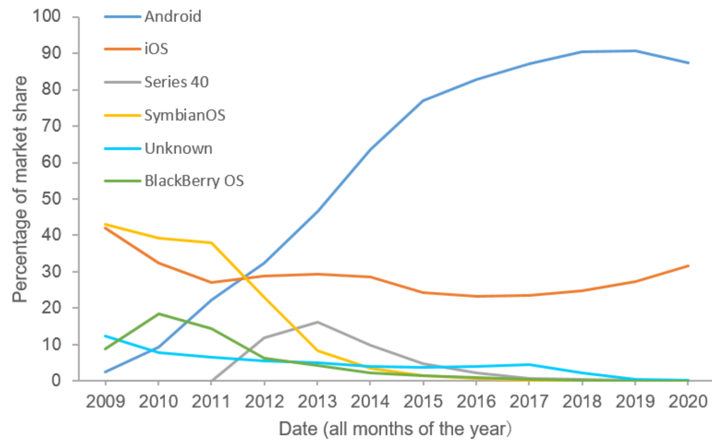


Figure 1: Market shares of different operating systems for mobile phones[2]

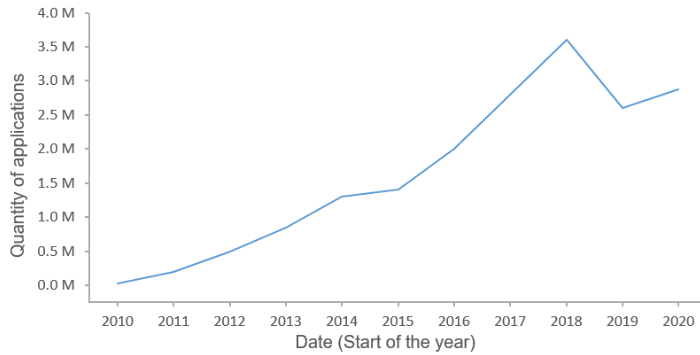
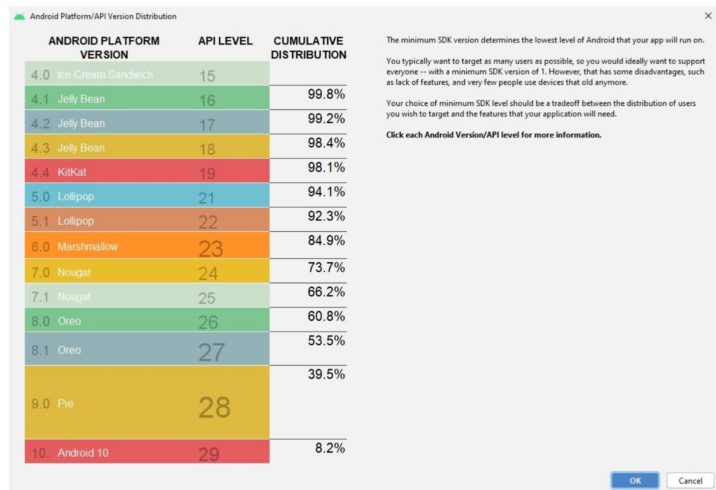


Figure 2: Number of applications available on Google's official store[3]

launch zero-day attacks based on the CVEs (Common Vulnerability and Exposures) found.



Android Platform/API Version Distribution

ANDROID PLATFORM VERSION		API LEVEL	CUMULATIVE DISTRIBUTION
4.0	Ice Cream Sandwich	15	
4.1	Jelly Bean	16	99.8%
4.2	Jelly Bean	17	99.2%
4.3	Jelly Bean	18	98.4%
4.4	KitKat	19	98.1%
5.0	Lollipop	21	94.1%
5.1	Lollipop	22	92.3%
6.0	Marshmallow	23	84.9%
7.0	Nougat	24	73.7%
7.1	Nougat	25	66.2%
8.0	Oreo	26	60.8%
8.1	Oreo	27	53.5%
9.0	Pie	28	39.5%
10.	Android 10	29	8.2%

Figure 3: Percentage of Android devices (i.e. Cumulative distribution in the screenshot) that the application could run on w.r.t the minimum SDK version (i.e. Android platfrom version in the screenshot)

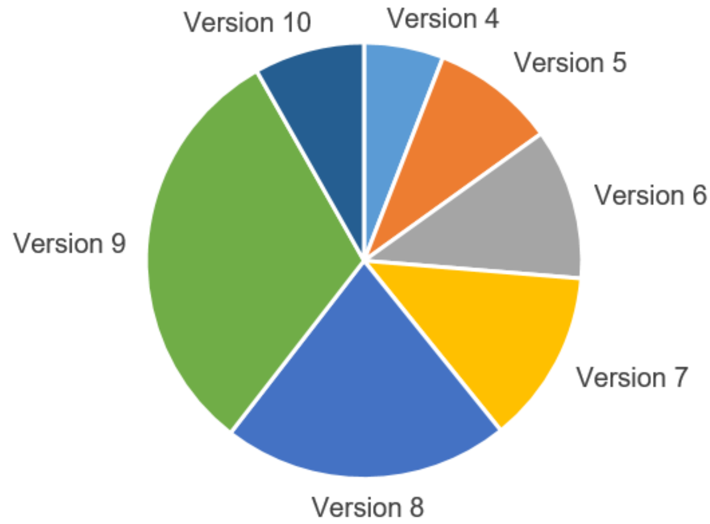


Figure 4: Distribution of Android versions based on the cumulative distribution data

1.2 Android malware and Android banking trojan

Malware is an umbrella word of **mal**icious soft**ware**. The term malicious refers to what could cause damage to the computer system or more intuitively, extremely hinder the user experience once being run. However, the boundary of benign and malicious software is actually not clear. For instance, some software are programmed to stealthily collect the device information (e.g. android_id, manufacturer, model, firmware version), keystroke whatever users are typing and/or record users' speeches. Finally all of these information or just the most representative or recognizable group (through some local processing) of information is sent to the server that has already been deployed for gathering user data. After being cleaned and classified, such data could be sold to business companies for more accurate advertisement shooting. Such software violate the privacy rights of users, but without other security applications, what is running at the backstage can not be easily notified by end users. Moreover, such functionalities sometimes exist in some innocuous applications which provides useful features, such as online shopping, chatting and social networking. Such software is usually called PUA (Potentially Unwanted Applicaitons), grayware or riskware. Since previous researches are inconsistent on the general taxonomy of malware and on whether or not such PUA should be considered as malware, only the categories that have been included most frequently are regarded as malware in this thesis.

Generally, Android malware can be classified into 7 categories. As shown in Table 1, the differences between seven types of malware regarding six properties are displayed. The meanings of the each sub-property is shown below:

1. Exisiting Form:

Table 1: Differences between different types of malware

		Virus	Worm	Trojan	Backdoor	Spyware	Rootkit	Botnet
Existing Form	Parasitic	✓						
	Masquerade		✓	✓	✓	✓	✓	✓
	Independent identity		✓					
Propagation mode	Repackaging	✓		✓	✓	✓	✓	✓
	Update attack			✓	✓	✓	✓	✓
	Sideload	✓	✓	✓	✓	✓	✓	✓
	Self-replicate	✓	✓					
Attack target	Local files	✓	✓	✓		✓		✓
	Network traffic	✓	✓					✓
	Operating system		✓	✓	✓	✓	✓	✓
Major risks	Data theft	✓	✓	✓	✓	✓	✓	✓
	Network paralysis	✓	✓					✓
	System damage	✓	✓					✓
Spreading speed		++	+++	+		+	++	+++
Difficulty of being detected		+	+++	++	++	++	++	++

(a) Parasitic: Attached itself to another executable in order to get executed whenever the host executable is triggered by victim's interaction.

(b) Masquerade: Disguised itself with legitimate apps' meta-data, especially the exterior icon and app name that are displayed to users during installation and in the home screen after successful installation.

(c) Independent identity: Reproduce itself continuously and spread the copies through networks or removable media running in the background without any user interaction

2. Propagation mode:

(a) Repackaging: A series of reverse engineering as well as editing, compiling and signing. Malware authors first look for and download popular applications on Google Play Store. After downloading, these APKs (Application Packages) are decompiled and disassembled for malware authors to analyze the source code and inject malicious code snippets to the source code, then add adequate invoking Android API (Application Programming Interface) methods to these malicious codes, add resources (e.g. images and strings) for phishing if necessary, as well as modify the manifest file so that these malicious actions can be executed during run-time. Afterwards, the integrated source code gets assembled and then gets compiled with

other resources into a package prior to signing with self-generated keys. Eventually, malware authors distribute the repackaged APKs to different third-party APK stores that can accelerate its spread, because the criteria for publishing uploaded APKs in such unofficial stores are relatively lenient, compared with the official store.

(b) Update attack: After a successful installation of the seemingly benign APK, a window that either notifies the user to agree on an update so that the application can satisfactorily perform its functionalities, or simply exhibits a window which tells the user that the installed APK is downloading updated content. Actually, the files to be downloaded are the payload of a malware, that acts as a hot-fix or plugin of the original APK.

(c) Sideload: In order to evade the detection engines [17] deployed in the official store but also attract as many potential victims as possible for monetization, malware authors first upload to the Google Play Store a simple APK like a calculator application which provides identical functionalities with other calculator applications. But the APK has already been injected with additional codes for communicating with a remote server from which the real malicious APK will be downloaded. Besides, some malware authors even encrypt the core malicious codes into a malformed file in the malicious APK. The core malicious portion would be decrypted into the memory and get executed during runtime, but such part is able to evade static detection on the device.

(d) Self-replicate: Malware copies itself continuously without any user interaction.

3. Major risks:

   (a) Data theft: Hackers exfiltrate valuable data to the remote server. The types of such data include but are not limited to SMSs, images, audios, videos, bank accounts and credentials. The sensitive information would be either sent to some interested companies for more accurate advertisement promotions or sadly end up in the hands of cyber criminals for monetization.

   (b) Network paralysis: If hackers can control the SMS and/or phone call module of the infected devices, they are able to launch DOS (Denial-of-service) attack through commanding the compromised devices to endlessly send SMSs or MMSs as well as dial premium-rated numbers. As a consequence, the network module would be overwhelmed by large amounts of traffic thus depleting the network resources of the compromised smartphones. Eventually, the victims would be hindered from accessing these modules.

(c) System damage: The damages that can be caused to the operating system include battery draining (e.g. constant process scanning), configurations altering (e.g. change in wallpaper) and functionalities invalidating (e.g. SMS redirecting, SMS blocking, call forwarding).

Virus [18, 19]: Virus refers to a kind of malicious program which parasitizes a benign application so as to avoid being recognized, and infects other files under the same system once being executed. Virus can land on the device either via a piggybacked software or through silently side-loading from a remote server. After landing successfully, virus then explores the file system to duplicate and inject itself to other files. Besides, virus can also perform other harmful activities based on how malware authors have programmed. The threat of virus includes data theft, software crash and/or denial-of-service (DOS) attacks.

Worm [20, 21, 22, 23, 24]: Worm is a type of malware that is notorious for its self-replication and high spreading speed through networks. Thesis malicious programs usually arrive at devices through social engineering like attachments in spam emails or instant messages. Once opened, the user would be directed to a malicious website or automatically download the worm payload. After installation, the worm runs silently at background and compromises the device without arousing attention of the user. Worms can even take advantage of the victim's restored email session to further spread the infection. Furthermore, worms can exploit the vulnerabilities of the operating system to commit malicious actions like modify and delete files. Most of time, the mission of a worm is to deplete the system resources by nearly infinite replication and to paralyze the network system, which is often followed by the attack of backdoor installation as well as rootkit installation for seizing full control of the infected device. And eventually the compromised device could just turn into a bot in the botnet.

Spyware [25, 26, 7]: Spyware is a type of malware that covertly gather information for more accurate advertisement shooting. Spyware mostly hides in some third party libraries used by the application and is relatively difficult to be detected, since the functionalities are developed under a third-party SDK, with their own native implementations. Spyware explores the operating system for any valuable or potentially valuable data, such as credit card numbers, netbank accounts and credentials, netbank balances, email addresses, web browsing histories, local audios, stored pictures and downloaded videos, chatting histories of social medias like twitter, facebook, and even keystroke logs. Afterwards, spyware send all or just part of the collected data that are considered worthy enough by malware authors to the remote server deployed by malware developers.

Backdoor [7, 27]: Backdoor is a type of malware that can be considered as an unauthorized path which can assist attackers to push other malware into the compromised device once

installed. Backdoor is usually generated with the help of rootkit to exploit undocumented processes in order to gain superuser privilege so that further attacks are able to bypass all security countermeasures and will not be notified by users. Backdoor is always abused by hackers to gain full control of a compromised device without user's knowledge.

Rootkit [28, 26, 25, 24, 29, 7]: Rootkit is a type of malware that is capable of obtaining remote access and control of the target device. Since rootkit exploits system-level vulnerabilities to seize administrative privilege, it is hereby difficult to be identified and removed due to some subsequent crucial alterations of the system configurations. Once rootkit successfully persists on the infected device, it executes along with every boot or reboot of the device, intended to open a portal (backdoor) for further attacks.

Botnet [22, 24, 29, 7]: Botnet is a kind of malware that is created to allow malware authors to remotely control and command the bot (penetrated device) to perform specific activities through a server. Such server usually controls a series of compromised bots which can be used for more aggressive purposes, i.e. launching DDoS (Distributed Denail of Service) attacks, deploying spiders for exfiltrating data that are stored in centralized servers, planting other malware (e.g. ransomware, adware) for monetization, etc..

Trojan [28, 22, 23, 25, 26, 29, 7]: As the word suggests, trojan represents one type of malicious programs that can be regarded as any malware that masquerades itself using the cover (exterior icon) of other benevolent applications with the intent of avoiding arousing suspicions from end users, while at the same time has the capability to commit aggressive activities against unauthorised local information. The term trojan is borrowed from the Ancient Greek story of the deceptive Trojan horse that led to the occupation of the City Troy. Due to the fact that most banks have developed their own applications for more convenient card management, trojan developers have considered not only phishing victims into inputting their bank credentials to an imitated fake bank login webpage, but also exploiting call forwarding as well as message forwarding so as to tackle the call of transfer challenge from banks and capture the mTANs (Mobile Transaction Authentication Numbers) generated by TFA (two-factor authentication) policy. According to 5 latest annual reports of mobile malware evolution from Kaspersky [30, 31, 32, 33, 34], Fig.5 shows that the number of new banking trojans detected by Kaspersky kept increasing from less than 0.5% in 2015 to around 3% in 2018. Although the proportion of new banking trojans detected in 2019 has decreased, compared with the percentage in 2018, they still comprised 2% of all the malicious Android applications detected.

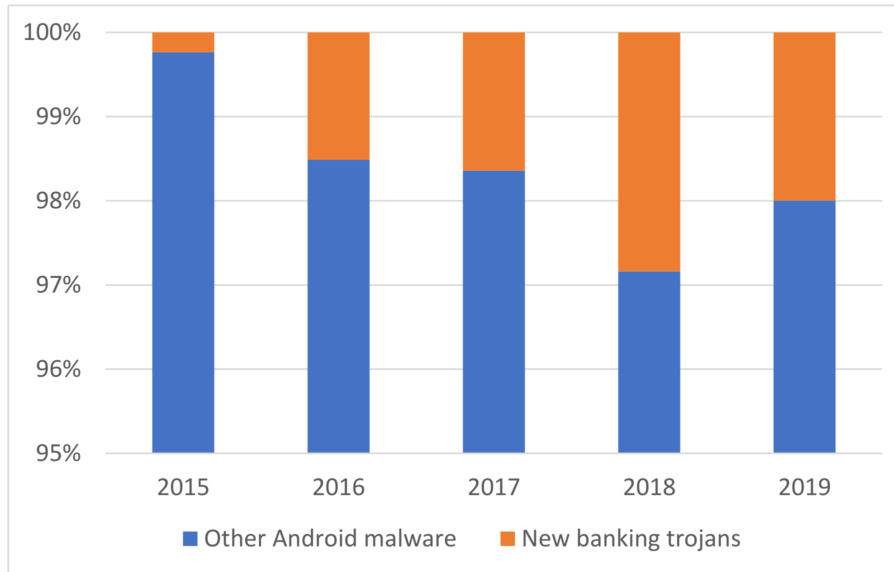


Figure 5: Banking trojans detected by Kaspersky from 2015 to 2019

1.3 APK Structure

As a package, an APK file is saved in the ZIP format as an archive file format. To view the contents inside an APK file, we simply need to change the .apk extension to .zip extension and then extract the files inside with a archive extractor. As Fig. 6 illustrates, the files compressed within an APK file typically include a directory named **res** storing necessary resources, a directory named **META-INF** storing the manifest information and other metadata about the java classes carried by an APK, a file named **AndroidManfiest.xml** storing the structure, metadata, components and requirements of an APK and a file named **classes.dex** storing compiled source codes. Besides, some APKs could contain a **libs** folder where Android developers implement additional native C or C++ codes in the format of shared objects (.so files); some APKs may also contain a **assets** folder for storing raw resources like databases, audios or videos, in which the files can be grouped into sub-folders and have flexible filenames[35]. Some malware would store the encrypted malicious .dex file in the **assets** directory.

Under the **res** directory, there generally exist the following directories including:

- **drawable** and/or **drawable-...dpi-vxx**: These directories store the icons and images used by the APK at runtime in PNG format with different pixel densities (DPI, dots per inch) to fit different devices at various resolutions for different API levels.
- **layout** and/or **layout-xxx**: These directories store the layout xml resources that define how an UI widget would be displayed.
- **mipnap** and/or **mipmap-...dpi-vxx**: These directories store the launcher icons which are shown on the homescreen with different DPIs to fit different devices at various reso-

lutions.

- **values**: This directory stores the values for the resources used in the APK, such as colors, dimensions, styles, integers, boolean values, resource IDs (in public.xml) to be referenced and most importantly strings like APK name, firebase URL (Uniform Resource Locator) and Google web service API keys.
- **values-...**: The **values-vxx**, **values-...-vxx** (e.g. values-v21, values-ldltr-v21) and/or **values-...dpi** directories store style xml resources, dimension xml resources and/or drawable xml resources for specific API level (21 for the examples given) and above and for device with specific DPI. The **values-xx** and/or **values-xx-rxx** (e.g. values-ca, values-zh-rCN) directories store different string resources based on different languages (ca and zh for the examples given).

The **META-INF** directory always contains three files, i.e. **MANIFEST.MF**, **CERT.SF** and **CERT.RSA**. The **MANIFEST.MF** writes various information used in the runtime, for instance, the list of filenames in the APK and the SHA1 digest encoded with base64 algorithm of each file. The **CERT.SF** contains the list of filenames in the APK and the SHA1 digest enc
of t
cont

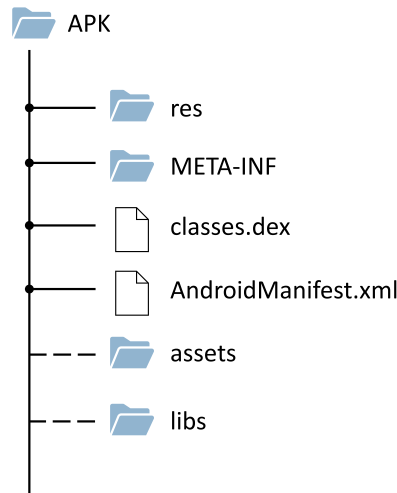


Figure 6: Dissected file structure of an APK file

1.4 Research Objectives

Due to the fact that the normal software and even the Android operating system are continuously refreshing to fix the vulnerabilities and/or add more features, the techniques abused by the

malware authors are also constantly being upgraded to become compatible with both the old environments and the new environments. This is because the new environments may easily neutralize some functionalities of the previous tools used by the malware developers. Thus the first aim of the project is to conduct studies of the techniques abused by recent popular Android trojans. Besides, most of the state-of-the-art approaches for identifying these trojans, like some machine-learning based approaches, are based on some previous knowledge (i.e. extracting useful meta-data from both benign and malicious applications and generating those features that are more distinctive than others when comparing the data from the benign with the data from the malicious). This project also aims to propose an approach to identify suspicious Android applications without utilizing prior knowledge. At last, to understand the core logic of the Android trojans, a thorough analysis of an Android trojan family would be conducted. What the readers can take away from this thesis are:

i. to know the core modules that are frequently abused by recent popular Android trojans to perform dangerous behaviors.

ii. to understand how to identify suspicious Android applications without much prior knowledge.

iii. to understand how a real-world Android trojan works.

1.5 Thesis Organization

This thesis is organised as follows:

- *Section 2*: This section presents a general of review of the literature of Android malware analysis.
- *Section 3*: This section profiles the recent Android trojans in different perspectives.
- *Section 4*: This section presents a case study of a specific trojan family in several perspectives.
- *Section 5*: This section presents a simple observation that can be used for identifying suspicious (malicious) applications.
- *Section 6*: A brief summary of the thesis contents and its contributions are given in the final chapter. Recommendation for future works is given as well.

2 Literature Review

2.1 Android Trojan Evolution

Having existed for more than 10 years, Android platform has progressively gained attentions from both industries and malware authors. In spite of the implementations of different countermeasures against malware, Android keeps attracting new malware authors to attack, because of the expanding groups of users and its powerful functionalities. Although the first Android trojan appeared in 2010, its functionality is restricted by the limitations of the platform. In 2017, Malwarebytes Labs announces that Android has possessed the capability of accommodating more composite applications, thus allowing for more vigorous malware [36]. The evidence of the improving functionalities of trojans is stated in [37]. The addition of botnet functionalities is what helps trojans to thrive [38]. Trojan Carberp only steals and collects SMS messages [39]; Trojan Hesperbot is capable of SMS spoofing and screen capturing [40]; GMBot first introduced the overlay attacking technique for phishing users to enter credentials in fake windows; The family Bankbot first came into sight in 2016, whose code takes the advantages of other Android trojan families [41].

2.2 Android Malware Families Analysis

Up to now, the researches on Android malware mostly concentrate on generic analysis, for instance, collecting different sets of samples and trying to categorize them into groups (detection, clustering or classification), which neglects the specifics that a single family can present. As far as we know, only two studies have comprehensively conducted analysis concerning a specific malware family. Both R. Yu [42] and M. Alejandro et al. [43] have technically analysed the variants of a ransomeware family and described the refinements that later generations introduce. Regarding trojan, DroydSeuss both statically and dynamically dissects Banking Trojan to find the shared artifacts of ABT like endpoint names with frequent itemset mining [44], which could be used to correlate active campaigns. Y. Hu et al. take a deep look into the fraudulent dating applications from many aspects (e.g. distribution, business model), these applications lure the end users to purchase a premium services with fake/bot accounts [45]. Even though these applications themselves do not target the device itself, they are used for bad purposes. N. I. AMINuddIN et al. detect Android trojan based on dynamically extracted system calls [46]. However, some trojans that implement environment-aware techniques are able to evade such approach. A detection framework regarding Android Banking Trojans (ABTs) is proposed by C. Bai et al. [47]. In that study, they develop a suspicion graph which relates goodware and

ABT with API packages and create a feature space based on suspicion ranks and suspicion scores obtained from the suspicion graph. Although they conduct analysis on 5 major ABT families, their main goal is to find the features that can distinguish each family from other malware and goodware, while our work is to dissect the technical details of trojan families. Y. Zhang et al. investigate the effects of current obfuscation and deobfuscation techniques regarding the performance of Anti-virus products and well-received detection approaches [48]. D. Wu et al. propose an approach for detecting order violations in Android, which could also be used for analysing some buggy Android malware [49]. Similarly, Y. Sui et al. present a event trace reduction tool for accurately and effectively reporting bugs of an APK [50]. Y. Tang et al. investigate application performance management (APM) libraries for their usage patterns. They find that some APM libraries still employ deprecated APIs thus failing to perform satisfactorily and misuse of APM libraries can lead to sensitive data leakage [51]. L. Wang et al. target a very important topic under the COVID-19 pandemic background, i.e., coronavirus-themed Android malware [52]. They build a coronavirus-themed dataset and study the attack vectors and mechanics of such malware. CHIME models an APK into an Activity Transition Graph (ATG) that is both context-sensitive and object-sensitive [53]. Such representation could also be valuable for extracting useful information from an APK for further detection or classification tasks. In addition, L. Li et al. conduct a systematic literature review that outlines the challenges and the methods regarding repackaged application detection in the research field [54]. They also collect a dataset of repackaging pairs, which is meaningful for further repackage detection researches. J. Gao et al. perform evolution studies on a market-scale application lineage regardingn the vulnerabilities. Some key findings are that most vulnerabilities can survive for at least 3 updates; some third-party libraries are the the root cause of vulnerabilities [55]. Y. Zhao et al. conduct experiments regarding the effect of sample duplication in the machine-learning dataset for machine-learning Android malware detection [56]. Their finding is that duplication in the public datasets has limited influence on supervised malware classification models.

2.3 Android malware detection

Other literature is more focused on detection (i.e. a binary decision system that tells whether a given sample is malicious or not), clustering (i.e. a decision system that can aggregate similar samples into a group without any label from prior knowledge) or classification (i.e. a decision system that can aggregate similar samples into a group with labels from initial knowledge).

We can partition the related work of Android malware detection in two classes: (i) run-time monitoring of the invoked events, (ii) static analysis of the code to detect known patterns of

misbehaviors.

2.3.1 Static Approaches

In 2013, an approach named AndroSimilar was proposed by P. Faruki et al. [57]. It attains the accuracy of 60% and follows the foot-print mechanism of known malware. Further it is used to identify the unknown malware. AndroSimilar classifies an application as malware or benign based on variable length signatures which are compared with signatures present in its database. DroidAnalytics follows the approach of signature-based detection which extracts and analyses the application at three levels [58]. It uses the principle of application signature along with API calls to identify malware applications. It detects 2494 malware samples from 102 families of malware database. Having said that, a higher than acceptable delay of false positive occurred in the experiment results. Stowaway was designed to test whether the Android applications were over-privileged or not [59]. It was applied to a set of 940 applications and found that about 33% Android applications were over-privileged. R. Sato et al. proposed a lightweight malware detection technique which analyses the Androidmanifiest.xml file [60]. It gains the accuracy of 90% by comparing the extracted information from the manifest file with keyword lists and also computes the malignancy score to judge the sample as malware or not. However, it fails to analyse entire code. Y. Tang et al. investigated the vulnerabilities of links within APKs from both attack and defense perspectives [61]. C. Y. Huang et al. detected 81% of the malware samples by applying ML algorithms which follows the principle of labelling [62], but it requires a second pass to determine whether the detected are malware or not. PUMA achieved the accuracy of 80% by extracting the permissions from the Android applications and used the machine learning approach to detect malicious applications [63], which lacks dynamic analysis. D. Arp et al. proposed DREBIN which is a lightweight method to identify malware applications by using the principles of joint vector space without using dynamic analysis [64]. DERBIN achieves the accuracy of 94% with a few false alarms and uses the machine learning approach to detect malware applications. Androguard disassembles and decompiles the applications and then calculates normalised compression distance pairwise [65], which contributes to reverse engineering but turns out to be time-consuming. H. Kang et al. adds developer's information as another feature in clustering malware groups [66], which is time-saving. Its performance for detection and classification is 98% and 90% accurate respectively. But its dataset is not enough. D. Octeau et al. employs the principle of connection between apps [67], which provides a method for link detection and gains 636 million links over 30 minutes. The only problem is that its probabilistic model seems to have some flaws. W. Tang et al. proposed a security distance model which is based on the idea that if the application requires more than

one permission, then it poses a challenge to the security of Android devices [68]. KIRIN is a lightweight tool which works on the principle of the certificate provided to the application and it used at the time of installation [69]. An application is said to be malware only when it is unable to pass all of its security tests. DroidMat is based on extracting information from the Android Manifest.xml file [70]. To improve the effectiveness of the classifier, K-mean classifier is used along with K-nearest neighbours algorithm. J. Hou et al. utilise the API calls, permissions and manifest components as basic features for building mixed feature sets to be trained with layer-by-layer Boosting model [71], which outperforms Random Forest and AdaBoost with an accuracy of 92% and a precision of 93%. E. R. Wognsen et al. formed the first ever formalized version of Dalvik Bytecode including java reflective features [72]. This approach is used to identify malware by using the data flow analysis. W. Zhou et al. proposed DroidMOSS which is a system that can measure the similarity of applications [73]. Fuzzy hashing is used to detect changes made in the application by repackaging. This tool is limited to a small set of malware database.

2.3.2 Dynamic Approaches and Hybrid Approaches

I. Burguera et al. proposed CrowDroid that uses dynamic analysis of Android applications behavior to detect malware [74]. They used unsupervised machine learning algorithms to detect malware from Android application and results were saved at the server, which achieves accuracy between 85% and 100% depending on malware. The shortcoming rests on its instability when dealing with increasing system call. A. Shabtai et al. proposed Andromaly which uses the principles of machine learning algorithms to monitor devices and differentiate between benign and malware applications [75]. Its accuracy lies between 80% and 90%.Unfortunately, it can lead to battery drainage problem. M. Zhao et al. proposed AntiMalDroid which monitors the behaviour of applications to detect malware and benign applications [76]. AntiMalDroid compares the signature of the application with the signature present in its database of benign and malware applications, but it is not cost-efficient. W. Enck et al. proposed TaintDroid which does real-time analysis [77]. It provides virtualized execution environment which is leveraged to deliver a real-time analysis. It tracks numerous sources of sensitive data and recognises the data leakage. L. K. Yan and H. Yin proposed DroidScope which works on three levels of an Android device, i.e., hardware, operating system and Dalvik virtual machine which enables in facilitating custom analysis and detect privilege based attacks [78], but it fails to cover sufficient code. A. Narayanan et al. proposed Context-aware Adaptive and Scalable Android malware detector which is capable of detecting all types of malicious behaviour applications [79], but it is adaptive to evolving malware. STREAM uses emulation based technique to detect privilege

based attack [80]. DroidAPIMiner extracts features at API level and evaluate different variables with the generated dataset, which overcomes the shortcomings of permission-based mechanisms and get a 99% accuracy [81]. S. Sheen et al. also looked at permission-based and API-based features [82], but some of them may get lost at run-time. R. Vinayakumar et al. took advantage of network parameters for all the features extracted, and attained 93.9% and 97.5% for dynamic and static analysis respectively [83]. The drawback is that it cannot be used to detect unknown malware. StormDroid combines popular features (i.e. Permission and sensitive API) with novel features (i.e. API sequence and dynamic behaviors) and then deploys and streamlinglizes the project [84]. The result shows that StormDroid achieves a good accuracy of 94%.

2.4 Android Malware Clustering and Classification

Previous works in the research field employed machine learning techniques to classify PC and Android malware. On PC platform, J. Z. Kolter and M. A. Maloof extracted n-gram features from dalvik bytecode to classify malware [85]. J. Kinable and O. Kostakis proposed a classification model by converting malware samples into FCGs (Function Call Graph) [86]. Compared with traditional malware that targets PCs, Android malware are often generated by poisoning legitimate apps with malicious payloads and they usually invoke sensitive API calls to perform malicious behaviors. Android is the major target of mobile malware. G. Suarez-Tangil et al. proposed Dendroid [87], a tool that automatically classifies malware into families based on the code structures. However, code layout could be easily obfuscated by bytecode-level transformation. C. Yang et al. proposed DroidMiner, which proposes a two-level graph model based on behavioral information and picks out the sensitive paths that is more frequently reached by malware [29]. Presented by M. Hurier et al. , EUPHONY is a lightweight tool that intends to infer the family name of a given single malware according to the lexicons of the provided sample [88]. AVCLASS , built by M. Sebastian et al. in 2016, relies on predefined labels or a ground-truth list as an indispensable input so as to output the results [89]. The major methodology of AVCLASS is plurality voting instead of majority voting as it is difficult for Anti-virus engines to reach an over 50% of agreement. Y. Li et al. introduced their malware clustering approach in 2017 [90]. Their method focuses on the similarity among the core parts of malicious code inside each sample application. To extract those payloads, they manage to exclude the popular legitimate third-party library code and attain a 90% of precision.

3 Deep Analysis of Recent Android Trojans

Although some literatures (like [9, 64, 91, 92, 47]) have analysed the technical details of some Android malware thoroughly, the datasets collected by them are somehow obsolete. For instance, even the most recent research [47] utilises the dataset during 2016-2017; The dataset used by [92] ranges from 2010 to 2016. Therefore, A newer dataset of Android trojans is a pressing need.

3.1 Data Collection and Extraction

To collect newer samples of Android trojans, a simple script for crawling some public security repositories has been written, especially the platform of Koodous [93] and Apkdetect [1]. To illustrate, these platforms have already output labels or classification results given a sample.

Since Koodous is a collaborative platform for Android malware analysts, most labels are produced by analysts through commenting with hashtag, as shown in Figure 7. Besides, Koodous provides a REST API that can be used to fetch the search results more efficiently. The workflow of extracting trojans with corresponding family name using Koodous API is shown in Figure 9. First, their API is used for requesting all the samples that are tagged with **trojan** and then the corresponding MD5 file hashes as well as other labels if any are recorded. If a sample is merely labeled with a single label **trojan**, such sample is discarded. Afterwards, those generic labels like **Detected** and **Malware** are filtered out. Only the most meaningful label remain. The meaningful label here refer to the label that is not generic and is the most suitable to be regarded as the sample's family name. The family name needs to be most unique among all of its labels. If the number of the remaining labels is larger than 1, a manual inspection is introduced to pick the most meaningful label that represent the family of a sample. Note that if there is no meaningful label for a sample, such sample is also discarded.

Take the samples in Figure 8 as an example. The sample spider.cheese.sketch would generate the label **BlackRock** as its family name, since the other three labels are all generic: The **Detected** tag is produced based on the results given by the vendors of the platform; the **Malware** tag is also another generic label that is too general; the label **Trojan** is one type of malware but still not specific enough to be the family name of the sample. The remaining label **BlackRock** is considered as the final label of the sample. The sample com.xlmfhz.cs would not generate any label, since all the three labels are generic. Besides the first two labels that have mentioned, the label **Dropper** refer to a type of malware that helps land the real trojan to the device. The sample fatigue.hazard.horror has the same tags with spider.cheese.sketch, thus generate the same label. Although alone.network.curve and behind.limit.track possess one

more label, **banker**, they also generate the same label **BlackRock**, as the label **banker** refer to a general type of malware that target the victim's bank card information. Compared with the label **BlackRock**, it is less specific and unique in terms of the coverage.

Figure 7: The screenshot that the hashtags added by an analyst's comment

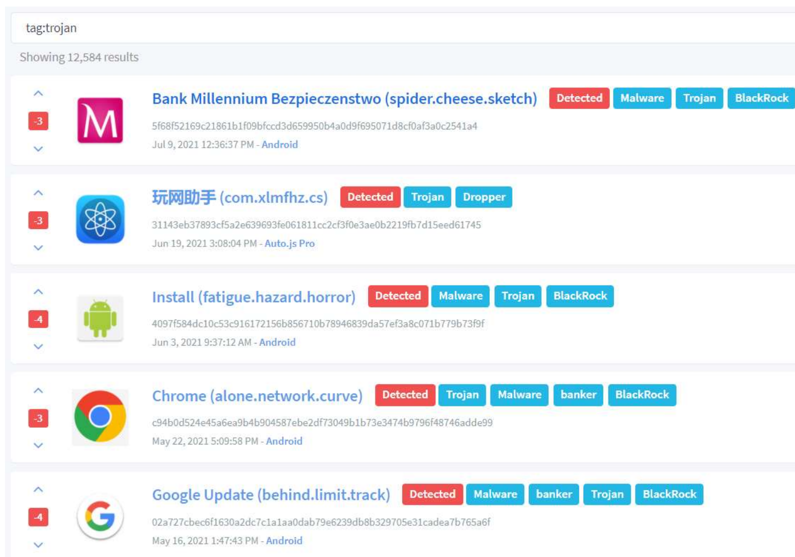


Figure 8: The screenshot of the first five samples when searching with "tag: trojan" on Koodous platform

Apkdetect is similar with Koodous. The differences are that Apkdetect does not provide any convenient API for queries and that the classification name produced by Apkdetect is based on the matching of the configurations of different malware, which are observed and submitted by security researchers. Due to the fact that Apkdetect does not possess a large volume of samples, a manual collection method is adopted, i.e. before loading the webpage, a chrome browser where the webpage is loaded is used. Specially, the network recording function is activated, which is able to capture and preserve all requests and responses in logs. Upon typing in the family name that is to be collected, the returned responses mainly include two parts, i.e. one with brief information including uploaded file name, file type, its MD5 hash and the most probable

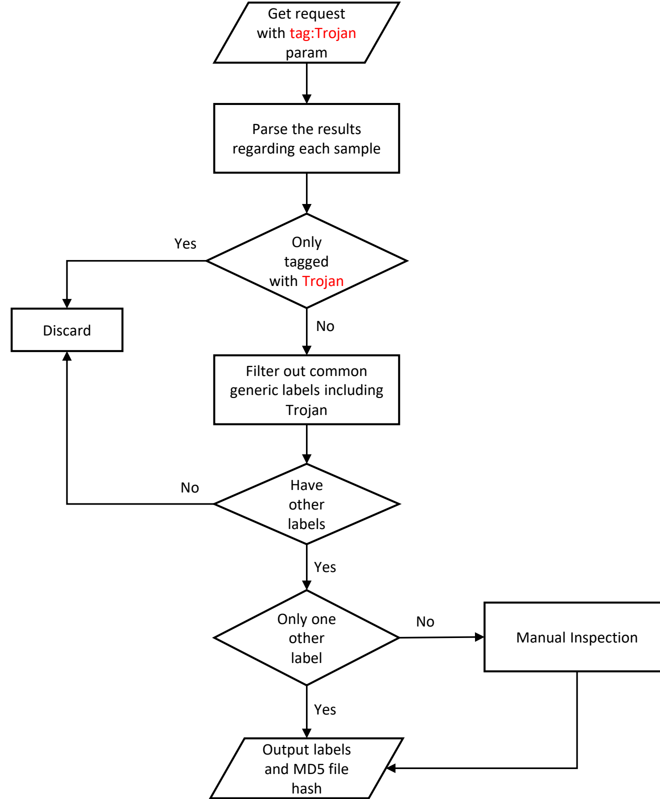


Figure 9: The workflow of generating family names of Android trojans through utilizing the Koodous platform

name produced (Figure 10); the other with more details, i.e. the configuration matched and the control & command server extracted from the sample (Figure 11). The second part is parsed to collect the MD5 hash, family name and matched configuration.

Specifically, Figure 12 illustrates the workflow of using Apkdetect for collecting the configuration information of the samples collected. First, the label of each MD5 hash collected from Koodous is used to search for samples that are identified to belong to the same family, and the configuration-MD5 pair regarding each sample is collected under such family. Afterwards, the MD5 hashes collected from Apkdetect are compared with those from Koodous. Once there is a duplicate MD5 hash, the MD5-label pair of the MD5 hash from the Koodous collection would be discarded, whereas the configuration-MD5 pair from Apkdetect would be collected under its family name. Due to the fact that Koodous does not generate any configuration information of a given sample, after all duplicates are discarded, each sample of the Koodous collection would be uploaded to Apkdetect to collect their configuration information. Since Apkdetect would also generate the probable name of an uploaded sample, if the probable name given by Apkdetect differs from the family name produced by Koodous, such sample (MD5-label and

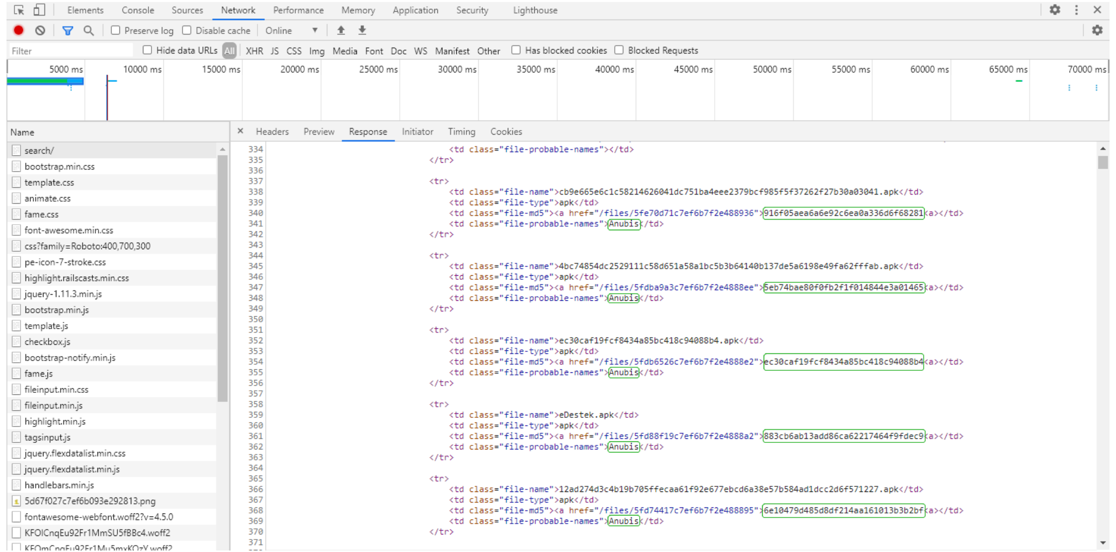


Figure 10: The captured brief information that matches the family name (e.g. Anubis) input in the search box in Apkdetect

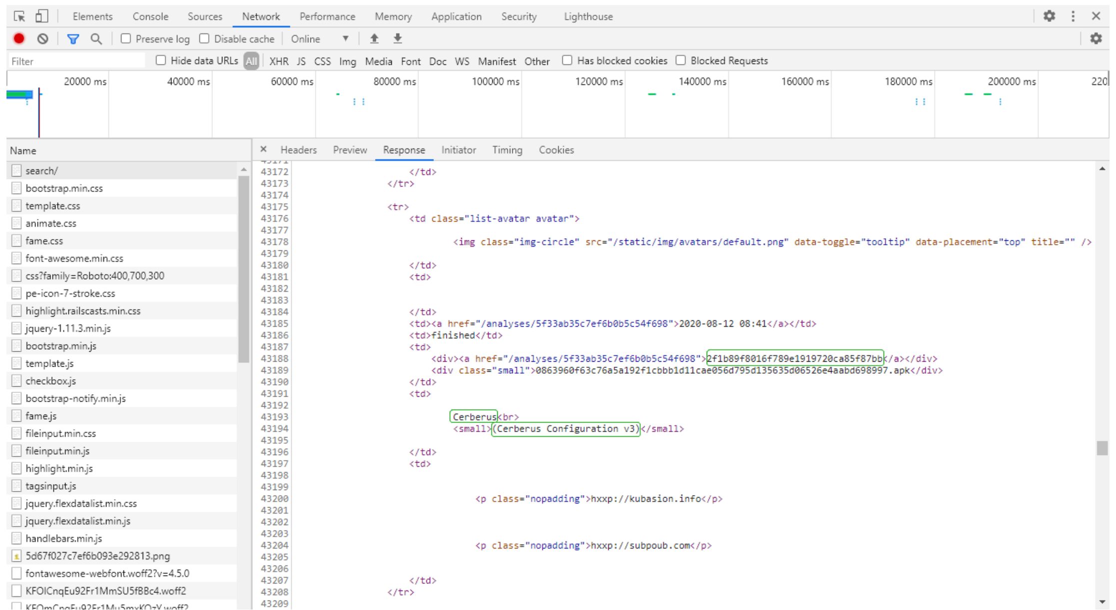


Figure 11: The captured detailed information that either matches or is somehow relevant to the family name (e.g. Anubis) input in the search box in Apkdetect

configuration-MD5 pairs) would be discarded from both collections. In the end, the statistics of the data collected are illustrated in Table 2 with the corresponding timeline based on online blog posts from security companies.

Table 2: The timeline of 20 (taking sub-families into account) Android trojans in the collection

Trojan	Number of Samples	Discovered Month
Flexnet	53	2017-07 [94]
RedAlert	521	2017-09 [95]
Bankbot Anubis	722	2017-11 [96]
Catelites	97	2017-12 [97]
Gustuff	25	2018-04 [98]
Hydra v1	84	2018-07 [99]
BianLian	7	2018-10 [100]
Rotexy	5	2018-11 [101]
Cerberus v1	254	2019-06 [102]
Ginp v1	1	2019-06 [103]
Hydra v2	8	2019-06
Brata	3	2019-08 [104]
Ginp v2	1	2019-08
Ginp v3	37	2019-11
Hydra v3	18	2020-02
Cerberus v2	3	2020-04
BlackRock	15	2020-05 [105]
Cerberus v3	476	2020-08
Hydra v4	27	2020-08
Hydra v5	23	2020-10
Total	2380	

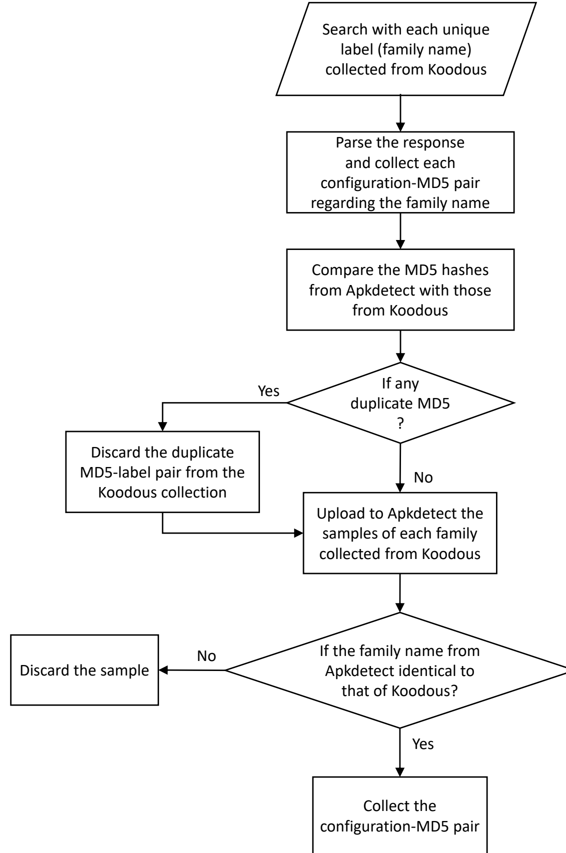


Figure 12: The workflow of using Apkdetect for generating configuration information of the samples

3.2 Attributes of Recent Android Trojans

Permissions. To dissect the mechanisms of how these trojans successfully attack Android devices, it is prerequisite to understand how the functionalities of an APK are able to be conducted. Android operating system has wrapped some java codes into functions or classes available for being called. These convenient functions or classes are dubbed Android API (Application Programming Interface) methods or classes, respectively. These API methods or classes are masked by corresponding permissions. An APK would be given access to certain module if and only if the corresponding permission is granted. Therefore, the permissions encoded in the Android-Manifest file are a critical indicator of what actions an APK is intended to perform. Thus, the permissions declared in AndroidManifest files of each collected sample are extracted by using some reverse-engineering tools like Apktool [106]. Afterwards, The permissions that come from the same (sub)family would be converted into an intersection set. The reason of extracting the largest shared set of these permissions is that some samples could be injected with noisy

permissions that are not used and are intended to counter analysis, the core permissions are those that have to be requested. At last, all the dangerous and/or sensitive permissions of one (sub)family are picked out and form a new set **X**.

According to Google's official documentation on Android development [107], each permission is characterized by a protection level which aids developers to understand the potential risks as well as the level of privilege implied in the permission. Here the permissions with dangerous or signatureOrSystem protection level are considered sensitive permissions. However, according to [108] and [109], the protection levels of some permissions are not always consistent, and even if the protection level of a permission is labeled as dangerous by Google, it is still possible to be classified as an indicator of a relatively benign feature (e.g. READ_CONTACTS which is among the top ten permission features in three classifiers in [108]). Nevertheless, in order to ensure that the studied trojan is fully characterized, permissions that have been and are labeled with dangerous or signatureOrSystem protection level are considered valuable for understanding the attacks of trojans. Table 3 groups the sensitive permissions of **X** into ten categories concerning their functionalities. Note that although the permission of RECEIVE_BOOT_COMPLETED has always been and is labeled with a protection level of normal [109, 110], it is actually more maliciously intended [108], which appears in the top ten malicious permission features of all four classifiers. Thus, it is assumed that the risks implied in such permission are somehow underrated. So this permission is also included as a sensitive permission.

With the ten categories of permissions, each trojan can be profiled by whether or not each category of permissions are requested. Table 4 displays the permission-based profiling of the 20 recent Android trojans. As the table illustrates, all of the 20 trojan families are designed to exfiltrate device information as well as personal records and/or receive commands from their masters, i.e. the remote servers; Each of these trojan families endeavours to grab more advanced privileges of the device so as to modify some configurations or neutralize the device for further attacks to harvest more revenues (e.g. device locking for ransom and/or aggressive adware planting); All of them are curious about what is happening or what has happened on the infected device, they abuse the permissions that masks some system applications or functions to steal the personal data. Besides, most of them want to keep monitoring the device so they abuse the RECEIVE_BOOT_COMPLETED to be notified whenever the infected device has finished booting. Also, the modules of messages and phone calls are of interest to most trojans since they could either steal the history data of these modules for selling these more detailed private information or more aggressively conduct SMS and/or call forwarding which is able to impede victims from arousing suspicion.

Table 3: The (abbreviated) dangerous and sensitive permissions requested by recent Android trojans

Abbreviation	Permissions	Abbreviation	Permissions
SMS (SMS/MMS)	READ_SMS RECEIVE_SMS SEND_SMS WRITE_SMS RECEIVE_MMS BROADCAST_SMS BROADCAST_WAP_PUSH SEND_RESPOND_VIA_MESSAGE	PKG (Package)	BROADCAST_PACKAGE_REMOVED REQUEST_INSTALL_PACKAGES INSTALL_PACKAGES PACKAGE_USAGE_STATS
CALL (Phone)	READ_CALL_LOG CALL_PHONE READ_PHONE_STATE PROCESS_OUTGOING_CALLS MODIFY_PHONE_STATE	APP (Application Info and OEM/System Application Related)	GET_TASKS REORDER_TASKS INTERACT_ACROSS_USERS_FULL INSTALL_SHORTCUT RECORD_AUDIO CAPTURE_VIDEO_OUTPUT DISABLE_KEYGUARD WAKE_LOCK
CONT (Contacts)	READ_CONTACTS WRITE_CONTACTS	BOOT (Boot Completed)	RECEIVE_BOOT_COMPLETED
SYS (System/File)	SYSTEM_ALERT_WINDOW WRITE_EXTERNAL_STORAGE READ_EXTERNAL_STORAGE MOUNT_UNMOUNT_FILESYSTEMS MODIFY_AUDIO_SETTINGS BIND_DEVICE_ADMIN BIND_JOB_SERVICE BIND_ACCESSIBILITY_SERVICE WRITE_SYNC_SETTINGS	NET (Network)	NFC INTERNET CHANGE_NETWORK_STATE CHANGE_WIFI_STATE CHANGE_WIFI_MULTICAST_STATE
LOC (Location)	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	ACC (Accounts)	MANAGE_ACCOUNTS GET_ACCOUNTS AUTHENTICATE_ACCOUNTS

Table 4: An overview of recent Android trojan (sub)families in regard to the categories of permissions requested

Trojan	Permission Categories									
	SMS	PKG	CALL	APP	CONT	BOOT	SYS	NET	LOC	ACC
Bankbot Anubis	✓	✓	✓	✓	✓	✓	✓	✓	✓	
BianLian	✓	✓	✓	✓		✓	✓	✓		
BlackRock	✓		✓	✓	✓	✓	✓	✓		✓
Brata				✓			✓	✓		✓
Catelites	✓		✓	✓	✓	✓	✓	✓		✓
Cerberus v1	✓		✓	✓	✓	✓	✓	✓		
Cerberus v2	✓		✓	✓	✓	✓	✓	✓		✓
Cerberus v3	✓		✓	✓	✓	✓	✓	✓		✓
Flexnet	✓		✓	✓	✓	✓	✓	✓		
Ginp v1	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ginp v2	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Ginp v3	✓	✓	✓	✓	✓	✓	✓	✓	✓	
Gustuff	✓			✓	✓	✓	✓	✓	✓	✓
Hydra v1	✓		✓	✓		✓	✓	✓		
Hydra v2	✓		✓	✓		✓	✓	✓		
Hydra v3	✓		✓	✓		✓	✓	✓		
Hydra v4	✓	✓	✓	✓		✓	✓	✓		
Hydra v5	✓	✓	✓	✓		✓	✓	✓		
RedAlert	✓	✓	✓	✓	✓	✓	✓	✓		
Rotexy	✓		✓	✓	✓	✓	✓	✓		

Insights from Permissions: Android trojans heavily rely on the functionalities that are related to the system/file modules provided by the Android operating system. These functionalities are the building blocks of all the subsequent attacks. Besides, the network functionalities also play an important role to perform data exfiltration for an Android trojan. Therefore, from the user side, whenever an application request and prompt an abnormal permission for approval, the end user should always inspect the functionalities behind such permission but also think twice before granting the permission. If an application often consume more network data than other similar applications, the user should be careful of using such application and/or even consider uninstalling it.

Intent-filters. An APK can consist of four types of components (activitity, service, broadcast receiver, content provider), which are also entry points through which the system or the user can enter the application. Three out of four types, activities, services, and broadcast receivers, can be activated by intents, which can be considered as asynchronous messages that wrap corresponding system-wide events. Thus malwares fully exploit such mechanism to trigger specific components for launching their attacks. Therefore, the intents within intent filters of a sample can reflect what events are of interest for a sample as well as how the components can be triggered. Besides, according to Google's official documentation, some intents are actually not normal intent since they can only be delivered by the system. These intents are usually what malwares listen to.

The intents within intent filters can be extracted also by using Apktool. Afterwards, The intents that come from the same (sub)family would be converted into an intersection set. At last, all the (protected) intents of one (sub)family are picked out and form a new set $\mathbf{Y}$. Table 5 displays the grouped intents that are of interest to trojans for activation, and the starred intents are documented as protected intents by Google. As can be seen in Table 6, all trojan families are designed to be triggered upon some system events, which means that the device has already been compromised. 19 out of 20 trojan (sub)families listen to boot events to avoid being interrupted by booting activities, thus being persistent. Also, except Brata, all trojans are given the missions to listen to SMS events, which could help them steal any SMS and/or MMS. 17 out of 20 (sub) families have modules for listening to admin events, which is indicative of the techniques exploited by these trojans to either become a system application or disable the victim's current administrative privilege. 14 out of 20 could also act as downloaders. They are interested in events of package changes for inviting other malicious payloads into the infected device. Comparatively, Rotexy is the most sensitive trojan family of all, since nearly all the categories of intents listed

Table 5: The (abbreviated) intents that are of interest to malware authors to trigger trojans

Abbreviation	Intents	Abbreviation	Intents
DRM (Dream)	ACTION_DREAMING_STOPPED*	PLUG (Headset Plug)	ACTION_HEADSET_PLUG
MAIN (Main Activity)	ACTION_MAIN	GCM (Google Cloud Messaging)	ACTION_TASK_READY
SCR (Screen)	ACTION_SCREEN_OFF* ACTION_SCREEN_ON*	CALL (Phone Events)	ACTION_PHONE_STATE_CHANGED ACTION_NEW_OUTGOING_CALL*
BOOT (BOOT Completed)	ACTION_BOOT_COMPLETED* ACTION_LOCKED_BOOT_COMPLETED* QUICKBOOT_POWERON	C2DM (Cloud to Device Messaging)	RECEIVE REGISTRATION UNREGISTRATION
NET (Network)	CONNECTIVITY_CHANGE WIFI_STATE_CHANGED	PKG (Package)	ACTION_PACKAGE_ADDED* ACTION_PACKAGE_REMOVED* ACTION_EXTERNAL_APPLICATIONS_AVAILABLE*
ADMIN (Administration)	ACTION_DEVICE_ADMIN_ENABLED ACTION_DEVICE_ADMIN_DISABLED ACTION_DEVICE_ADMIN_DISABLE_REQUESTED	BATT (Power/Battery)	ACTION_POWER_CONNECTED* ACTION_POWER_DISCONNECTED* ACTION_BATTERY_LOW* ACTION_BATTERY_OKAY*
SYS (System Events)	ACTION_APP_ERROR ACTION_TIME_TICK* ACTION_USER_PRESENT* AccessibilityService	SMS (SMS/MMS)	ACTION_SEND ACTION_SENDTO SMS_RECEIVED SMS_DELIVER WAP_PUSH_DELIVER ACTION_RESPOND_VIA_MESSAGE

are what Rotexy listens to. It can be noted that Rotexy even listens to the plug events of the headset as well as the battery conditions. Although more intent actions to listen to could more easily cause suspicion and even get killed by some security applciations, the attacks prepared by the trojan authors are able to be performed more frequently. Moreover, it can be noticed that Gustuff utilizes GCM for potential information exfiltration and/or command delivery as a Control & Command server while Rotexy abuses C2DM as a channel for communications.

Insights from Intent-filters: Android trojans register a wide range of intent filters so that they can be triggered frequently to keep active. Therefore, from the user side, whenever an application is monitored to be always running in the background and to always appear at the top of the back stack, the end user should become vigilant, since they are very likely to be the symptoms that the device has been compromised.

Anti-analysis

To avoid being detected or noticed by either anti-virus applications or the user himself, malware writers usually employ some techniques to hinder the analysis or monitoring. Table 7 displays the anti-analysis techniques that are used by these trojans. In order to evade static

Table 6: An overview of recent Android trojan (sub)families in regard to the categories of intent actions used

Trojan	Categories of Intent Actions													
	DRM	PLUG	MAIN	GCM	SCR	CALL	BOOT	C2DM	NET	PKG	ADMIN	BATT	SYS	SMS
Bankbot Anubis	✓		✓		✓		✓		✓	✓			✓	✓
BianLian	✓		✓		✓		✓		✓	✓	✓		✓	✓
BlackRock			✓				✓				✓		✓	✓
Brata			✓										✓	
Catelites			✓				✓			✓	✓		✓	✓
Cerberus v1			✓				✓			✓	✓		✓	✓
Cerberus v2			✓				✓			✓	✓		✓	✓
Cerberus v3			✓				✓			✓	✓		✓	✓
Flexnet			✓			✓	✓				✓		✓	✓
Ginp v1			✓				✓		✓	✓	✓		✓	✓
Ginp v2			✓				✓		✓	✓	✓		✓	✓
Ginp v3			✓				✓		✓	✓	✓		✓	✓
Gustuff			✓	✓			✓		✓				✓	✓
Hydra v1	✓		✓		✓		✓		✓	✓	✓		✓	✓
Hydra v2	✓		✓		✓		✓		✓	✓	✓		✓	✓
Hydra v3	✓		✓		✓		✓		✓	✓	✓		✓	✓
Hydra v4	✓		✓		✓		✓		✓	✓	✓		✓	✓
Hydra v5	✓		✓		✓		✓		✓	✓	✓		✓	✓
RedAlert			✓				✓				✓		✓	✓
Rotexy	✓	✓	✓		✓	✓	✓	✓	✓		✓	✓	✓	✓

analysis, malware authors can take several countermeasures to increase the difficulties for analysts to find out the real malicious payload. Such measures include renaming, string encryption, dynamic payload and native payload loading. Renaming, or more specifically, variable, method and class names renaming is a technique that renames any meaningful variable, method and class name into randomly generated meaningless names. The advantage of this technique is that if some anti-virus engines employ a detection mechanism based on name matching, the malware can easily evade such detection. In addition, the renaming strategy also increases the time cost for human analysts to find its malicious parts. The disadvantage of it is that it is not capable of escaping the detection approaches that are based on APIs or other literals. String encryption is a technique that encrypts most or all the sensitive plaintext encoded in the source code with some public encryption algorithm like AES, or even implement their own algorithm of ciphering plantext. The advantage of this technique is that some detection engines would fail to identify the malware if its detection is based on string matching, and human analysts would also have to take time to figure out the real string. The disadvantage of it is that it is still not able to escape API-based detections, In order to escape the static scanning of API records in the reverse-engineered source code, malware authors employ a dynamic loading method by using classloader API class. They can encrypt the real malicious payload in some location (usually in the assets folder) and decrypt it at run-time. By doing so, the static API-matching tactics would fail. Similarly, malware authors can also choose to hide sensitive APIs and/or strings in native libraries, which would be invoked only at run-time to defeat mainstream static analysis that focuses on Dalvik bytecode.

Although the techniques above seem promising when dealing with static analysis, all of them would not manage to escape from dynamic analysis, since all the malicious intentions would be exposed when the malware is executed. In order to evade such detection scheme, malware authors employ techniques based on an observation that most dynamic analysis are conducted on emulators rather than real devices to avoid causing any unrecoverable exploit to the real system. Thus the device parameters like manufacturer, ID are indicative of whether the environment is a sandbox environment, since these parameters of most emulators are actually fixed and can not be modified easily. Besides, some network traffic capturing tools like wireshark [111] or fiddler [112] are able to monitor any communications between the endpoint and remote servers by introducing an intermediate proxy server. To avoid being identified by such tools, some trojan authors would try to encrypt the communication between the Command & Control server and the bot. By doing so, even if the network traffic would be obtained, if analysts can not figure out the encryption algorithm, they still can not understand what is happening. Moreover, to avoid being killed by security applications and to evade performing malicious activities in

Table 7: An overview of recent Android trojan (sub)families in regard to the anti-analysis techniques

Anti-analysis	Evade Static Analysis (ESA): Renaming (RN), String Encryption (SE), Dynamic Loading (DL), Native Payload (NP)							
	Evade Dynamic Analysis (EDA): Check Device Info (CDI), Encrypt Communication (EC), Check Installed App (CIA) & Check Running Processes (CRP), Verify Sensor Data (VSD)							
Trojan	ESA				EDA			
	RN	SE	DL	NP	CDI	EC	CIA and/or CRP	VSD
Bankbot Anubis v1					✓		✓	
Bankbot Anubis Obfuscated	✓	✓	✓		✓	✓	✓	✓
Bankbot Anubis v2	✓	✓	✓		✓	✓	✓	✓
Bankbot Anubis v2.5	✓	✓	✓		✓	✓	✓	✓
BianLian	✓	✓	✓		✓	✓	✓	
BlackRock	✓		✓		✓	✓	✓	
Brata		✓			✓		✓	
Catelites	✓		✓				✓	
Cerberus v1	✓	✓	✓			✓	✓	✓
Cerberus v2	✓	✓	✓			✓	✓	✓
Cerberus v3	✓	✓	✓			✓	✓	✓
Flexnet			✓		✓			
Ginp v1			✓				✓	
Ginp v2			✓				✓	
Ginp v3			✓		✓		✓	
Gustuff	✓		✓		✓		✓	
Hydra v1			✓		✓	✓	✓	
Hydra v2			✓		✓	✓	✓	
Hydra v3			✓		✓	✓	✓	
Hydra v4			✓		✓	✓	✓	
Hydra v5			✓		✓	✓	✓	
RedAlert	✓		✓		✓	✓	✓	
Rotexy	✓	✓	✓		✓	✓	✓	

an emulated environment, some trojans are able to extract and exfiltrate the list of installed applications as well as the currently running processes so that the Command & Control server would deliver different commands based on different infected environments. If the infected environment is an emulated environment, the C&C server would command the trojan to keep silent and behave normally. More interestingly, some trojans would try to acquire the sensor data to ensure that the infected device is not still all the time. If the infected device is still for a long time, then such device could be regarded as a potential emulated environment.

As shown in Table 7, all trojans except Flexnet extract the list of installed applications or running processes prior to sending to the C&C server. The popularity of such technique could be due to the fact that the installed applications as well as the current running processes are actually quite informative of both the compromised environment and the potential targets that are valuable to be overlayed by phishing pages. All trojans except two abuse the dynamic loading technique to hide their real payloads. However, no trojan employs packing or shelling technique which encrypts the payload into native code to evade static analysis. This could be due to an observation that some anti-virus products are sensitive to such packing or shelling technique which would even report a packed benign APK as malicious [113], since some of the products themselves would inject some unwanted modules to the submitted APKs, which would definitely trigger some anti-virus engines. Additionally, the products of reinforcementing APKs also would bring about obvious library names [114] that would be used to identify the products themselves. As the writer himself experienced, most of these packing services these days also detect the submitted samples so that the samples do not include any violent intentions. 14 out of 20 sub(families) have implementations to check the device's parameters, like os.Build.PRODUCT, os.Build.MANUFACTURER and os.Build.MODEL. Higher versions of Anubis employ a communication encryption and decryption technique which hard-codes the key within the source code, while Hydra and BlackRock borrow JSCH (Java Secure Channel) for safeguarding the traffic. RedAlert encodes the traffic in Base64 format [115], and Rotexy encrypts the traffic with AES algorithm [116].

Insights from Anti-analysis: Android trojans utilize various techniques to bypass anti-virus applications so that they are able to land on the device safely. These techniques ensure that only when the environment is ideal would the Android trojans perform their real functionalities. The traditional detection methods like static reverse engineering analysis and dynamic sandbox-based analysis are gradually becoming outdated. This could mean that hybrid analysis would become a trend for dissecting and detecting new Android trojans.

Persistence

Table 8: An overview of recent Android trojan (sub)families in regard to the persistence techniques

Trojan	Persistence						
	Delete Created Files	Delete New SMS	Hide Shortcut	Privilege Escalation	Monitor Uninstallation	AV Disabling	Screen Off/Lock
Bankbot Anubis v1		✓	✓	✓	✓	✓	✓
Bankbot Anubis Obfuscated	✓	✓	✓	✓	✓	✓	✓
Bankbot Anubis v2	✓	✓	✓	✓	✓	✓	✓
Bankbot Anubis v2.5	✓	✓	✓	✓	✓	✓	✓
BianLian	✓	✓	✓	✓	✓		✓
BlackRock	✓		✓	✓	✓		✓
Brata			✓	✓	✓		✓
Catelites	✓	✓	✓	✓		✓	✓
Cerberus v1			✓	✓	✓	✓	✓
Cerberus v2			✓	✓	✓	✓	✓
Cerberus v3			✓	✓	✓	✓	✓
Flexnet	✓		✓	✓	✓	✓	✓
Ginp v1			✓	✓	✓		✓
Ginp v2			✓	✓	✓		✓
Ginp v3	✓		✓	✓	✓	✓	✓
Gustuff	✓		✓	✓	✓	✓	✓
Hydra v1	✓	✓	✓	✓	✓	✓	✓
Hydra v2	✓	✓	✓	✓	✓	✓	✓
Hydra v3	✓	✓	✓	✓	✓	✓	✓
Hydra v4	✓		✓	✓	✓	✓	✓
Hydra v5	✓		✓	✓	✓	✓	✓
RedAlert	✓		✓	✓			✓
Rotexy	✓		✓	✓			✓

After successfully landing on the device, trojans need to maximise its possible living spans for launching attacks and harvesting revenunes. To achieve this, they can hide their traces (e.g. shortcut, SMS history, notification and/or alert window) after being executed to avoid uninstallation, more aggressively redirect the user to the main screen if the trojan monitors that the user tries to uninstall the trojan from the settings, disable existing anti-virus applications, and/or grab admin privilege. Table 8 displays the techniques abused by these trojans to keep persistent on the device. By calling PackageManager.setComponentEnabledSetting(componentName, COMPONENT_ENABLED_STATE_DISABLED, DONT_KILL_APP), all trojans can hide their icon from the screen. Also, all trojans pursue a higher level of privilege to avoid being killed. All trojans except Catelites, RedAlert and Rotexy abuse the accessibility privilege to monitor whether any uninstallation (like buttons with text **delete** or **remove**) action is happening, while Catelites, RedAlert and Rotexy registers themselves as new admistrators. All trojans implement screen control functions by using DevicePolicyManager.lockNow. 16 out of 20 trojan (sub)families are beware of any Anti-virus applications on the device. Anubis and Cerberus build an app.AlertDialog with uncancelable icon to coerce victims to manually deactivate Google Protect. Ginp v3 as well as Hydra also implement an auto-click module for disabling Google

Protect. Catelites, Flexnet and Gustuff encode 20+ keywords from the package names of some popular Anti-virus products and scan the device's installed applications and running processes to check whether there is any match with these key words. Only 9 of the (sub)families actually delete the new SMS created by using ContentResolver.delete(SMS_Uri, (String) null, (String[]) null). It can be seen that although three versions of Hydra implement SMS removal, such technique is not of interest to the other two versions.

Insights from Persistence: Android trojans are designed to become the Superuser of the target device. Only based on the power of the Superuser can Android trojans become the real devil. They hide what they create to blind the victims. Even if the victims realise their existence, to uninstall them or even to interact with the compromised device is a difficult task. Thus, from the user side, it is reasonable to always keep skeptical of anything unusual that occurs on the device. Once found, it is further recommended to inspect the device with the help of some famous third-party anti-virus applications.

Command and Control (C&C)

Command and Control server (C&C) refers to the server that has been possessed by cyber-criminals under certain domains and deployed with protocol parsing template as well as some critical resources which may have high risks of being detected by security countermeasures on target devices based on the attack flow expected. It can be regarded as a counsellor, which both collects valuable information and gives different commands judging by different condition parameters returned by the bot (i.e. trojan that has been planted successfully on some endpoint). Since the C&C servers of some samples collected have already been taken down, so some of the analysis can only be achieved by looking into the reverse-engineered source code.

Table 9 shows how different (sub)families communicate with their C&C servers. Anubis construct its own communication data with self-defined delimiters, like |*sockshost*=, |*user*=, |*pass*=. BianLian utilises the Firebase Messaging service for commanding the modules. Flexnet directly append the request parameters like IMEI, country, number and operator, after a specific PHP address. Other trojan (sub)families communicate with their C&C servers with JSON formatted messages. Interesting, only Rotexy has a backup plan in case of the takedown of its C&C. The role of C&C can also be fulfilled by sending specific SMS to the infected devices like "3458" to revoke device administrator privileges from the app and "393838" to change C&C address to the one in the SMS, which is a wise strategy.

Table 9: An overview of recent Android trojan (sub)families in regard to the communication methods with C&C

Trojan	C&C		
	Internet	SMS	Command Encoding
Bankbot Anubis v1	✓		Custom Protocol
Bankbot Anubis Obfuscated	✓		Custom Protocol
Bankbot Anubis v2	✓		Custom Protocol
Bankbot Anubis v2.5	✓		Custom Protocol
BianLian	✓		Custom Protocol
BlackRock	✓		JSON
Brata	✓		JSON
Catelites	✓		JSON
Cerberus v1	✓		JSON
Cerberus v2	✓		JSON
Cerberus v3	✓		JSON
Flexnet	✓		Custom Protocol
Ginp v1	✓		JSON
Ginp v2	✓		JSON
Ginp v3	✓		JSON
Gustuff	✓		JSON
Hydra v1	✓		JSON
Hydra v2	✓		JSON
Hydra v3	✓		JSON
Hydra v4	✓		JSON
Hydra v5	✓		JSON
RedAlert	✓		JSON
Rotexy	✓	✓	JSON

Insights from C&C: Android trojans is designed to communicate with their master servers (C&C) to exfiltrate the victims' personal data to their C&C as well as receive commands from their C&C. Communication with the remote server is an essential part if the trojan developers want to gain more revenue, since they can further launch customized scams according to different conditions of different victims. Although some trojans employ JSON protocol for communication, it is still futile to capture the network traffic for analysis, since the messages have already been encrypted from both the trojan side and the server side and only get decrypted locally, let alone custom protocols. However, from the user side, the packet rate (number of packets per second) of the device would be higher than usual if the device gets compromised by trojans, since the trojans are constantly uploading data and receiving data or even downloading files. Some network monitoring applications hence are of assistance in revealing abnormal applications on the Android device.

4 Analysis of the Bankbot Anubis family

According to G Data, a German cyber-security company [117], more than 94.2 million malicious apps in total have been detected by the end of this June, and more than 10 thousand new Android malware have been published per day [118]. Android OS is the second most attractive targeted OS for cyber-criminals. It will both incur potential risks to the protective mechanisms and pose severe financial threats to end-users. [7, 119]. Anubis Banking trojan, a type of financial malware, has broadly spread since 2018. By the end of 2018, the total number of detected banking trojan had exceeded 25 thousand as the proportion of banking trojan detected by Avast had quadrupled from the start of 2017 to September 2018 [120]. Motivated by the high threat level of banking trojans and by the lack of the publicity of knowledge in defense of its intrusive attack, we conducted detailed analysis of the attack mechanism of Anubis in four versions.

Although there are plenty of researches on Android malware, most of them focus on either malware detection [121] or malware classification (clustering) [122], some study the obfuscation techniques employed on Android malware [123], few have been interested in analysing the evolutionary patterns within a single family. This section aims to fill such gap by performing a thorough analysis of a specific Android malware family, which provides a new approach for understanding Android malware from low-level features to high-level behaviors.

In this section, we intend to provide a comprehensive insight on an obnoxious Android banking trojan named Anubis, which has been distributed mainly in Europe and Asia and

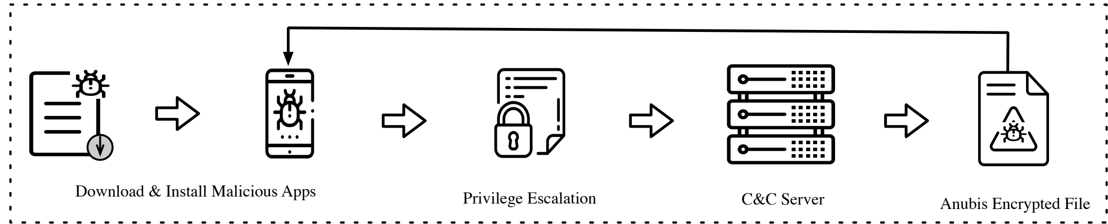


Figure 13: Anubis Attack Procedures

has shaped into extreme polymorphism [124], with over 17,000 samples captured on 2 open server panels for one time, according to Trend Micro, a multinational cyber security and defense company. The main contributions of this work can be summarized as follows:

- We describe the Anubis family from both run-time behaviors and implementation details.
- we study the overlay attack mechanism and how such mechanism is successful for tricking users
- we conduct different levels of analysis to find connections and variations among different versions of Anubis family

4.1 Anubis's Approach

The threat model of Anubis can be summarized as Figure 13, covering malware downloading and installing, run-time phishing-based privilege request, sensitive data exfiltration and further attacks based on the communications with the C&C server.

4.1.1 Phase 1: Mobile Devices Cyber Attack

Since the policy of Android open-source, all Android users not only gain access to billions of mobile-phone application free of charge and easy-install, but also suffer from the highest vulnerable attacks. Developers can easily publish Android applications in an App-marketplace, which is the rudimentary weakness: various applications which contain malicious codes can be published to the third-party market without any intensive code inspection or supervision. Although Google has deployed security mechanisms, like Bouncer[17] and Play Protect[125], which check and scan the applications that have been uploaded to its platform, malware authors still have ways to elude detection (e.g. program obfuscation or downloader a.k.a update attack[126, 127]).

In Figure 14, Anubis authors normally use the forgery methods to attach the "non-malfunction" downloaders to the popular applications in the App-markets then republish them back. Anubis downloader is a download toolbox, which doesn't contain any malicious functionality, for downloading the real payload of the Anubis. While most of safeguarding mechanisms of code

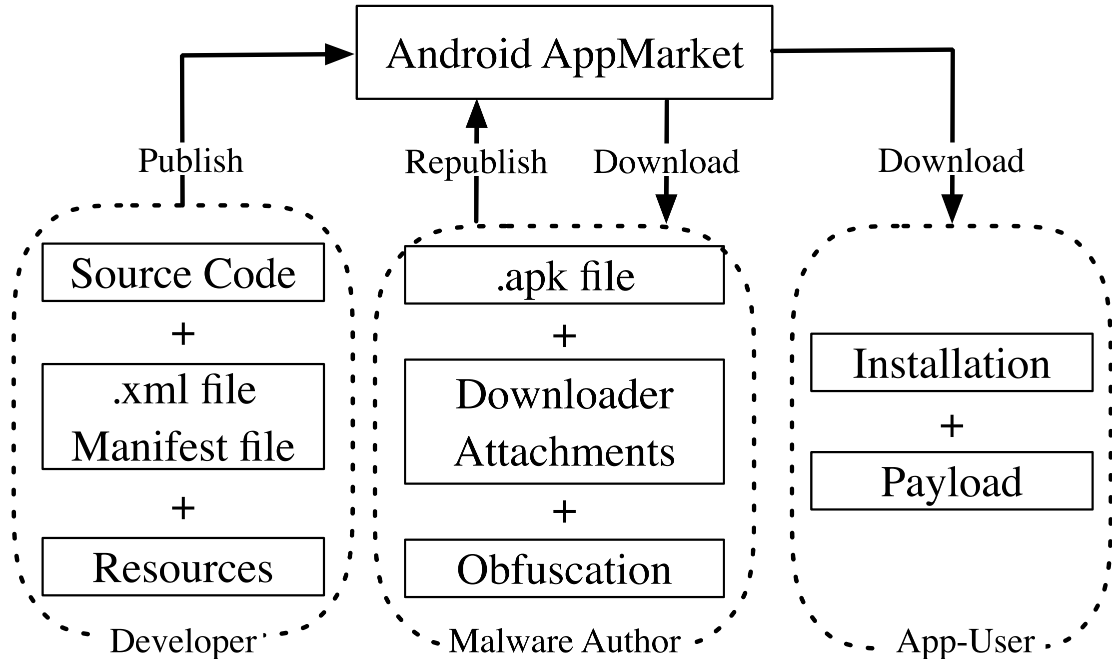


Figure 14: Infection Process

inspection and analysis are based on the 'developer-side': source code inspection, manifest file pre-execution test and resources file checkup. The official Android App-market can't afford a large scale of in-depth iteration of checking "in-apps" download in the code resources due to the high cost of resources and time. The real payload then manages to bypass the security check and is able to be downloaded into user's devices. Figure 15 gives the real-world code example that the attacker just manipulated the manifest-xml file in order to show updating notification to users while actually download the mal-functional application package from another server, which contains the real malicious payload of Anubis for higher privilege in system.

4.1.2 Phase 2: Privilege Escalation

In Figure 15, as users execute the downloader App, the downloader then fetches the full package of the malicious application containing the real payload of Anubis by rendering the *android.permission* block for INSTALL_REFERRER. Once the downloaded application is executed on the device, Anubis applies a devious method to render the *android.widget.Toast* block for requesting access to ACCESSIBILITY module in the phone. It is the core module used to ask users to grant ACCESSIBILITY access to the malicious application named "Google Service". However, the fact is that the accessibility module is a powerful functionality on the device, for example, Anubis can obtain window content, monitor changed events and even simulate

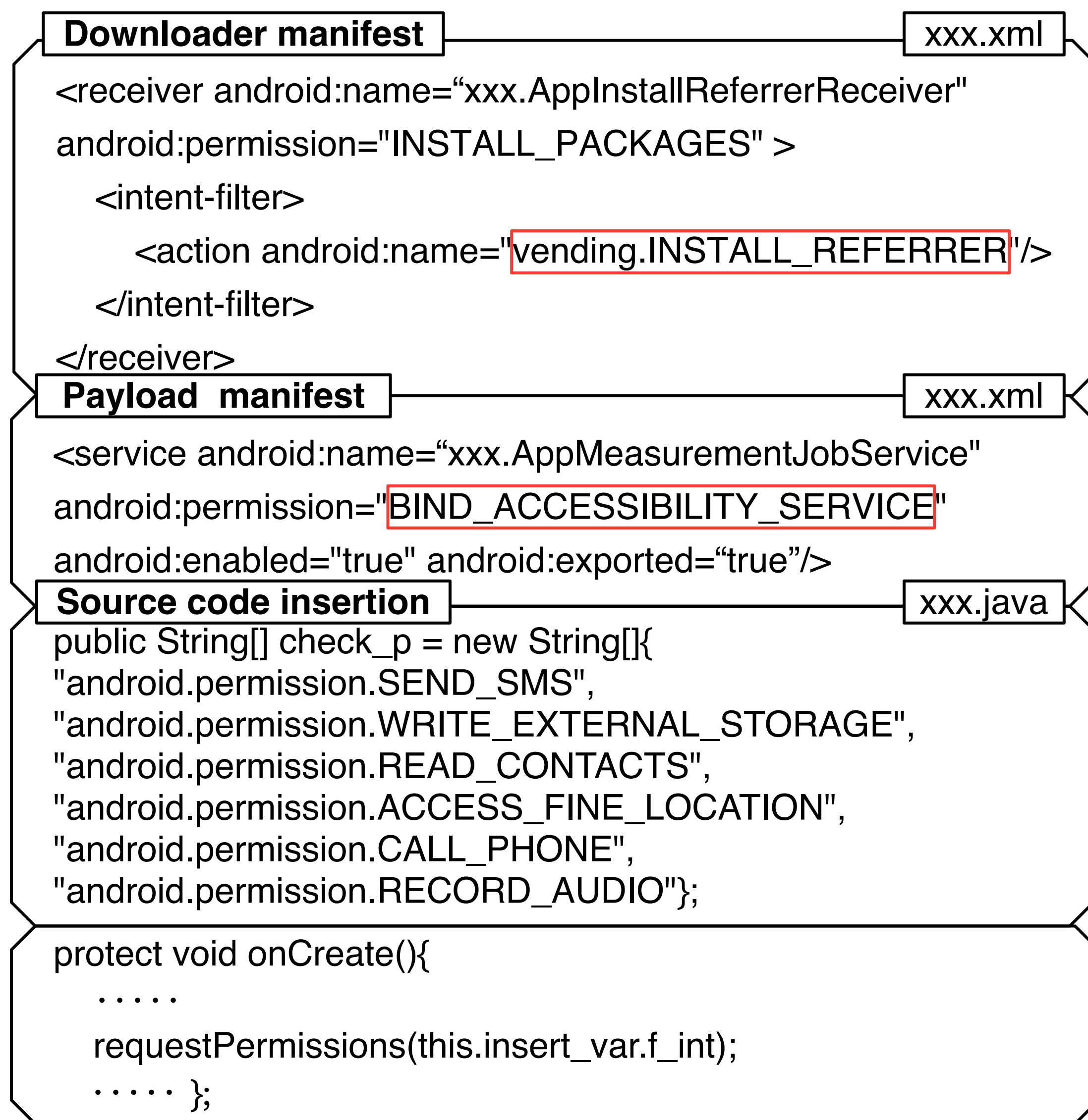


Figure 15: Code Injection in Forged App

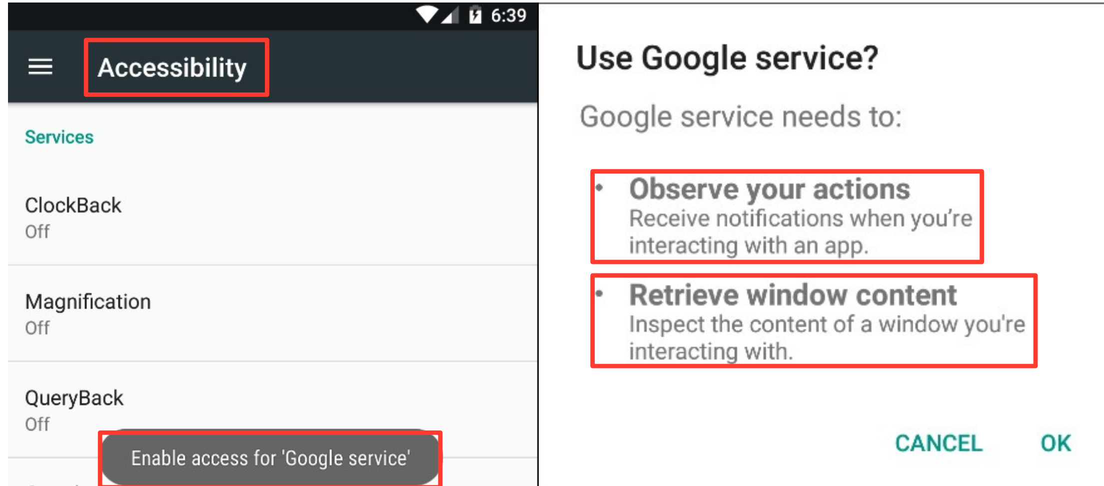


Figure 16: Interface of Anubis Accessibility Extraction

user's actions without interactions. With limited knowledge, unprofessional device users may become confused by the content shown in Figure 16 and just approve of the request. Then the device now is infected with the Anubis Trojan. The main targets of Anubis are banking and financial-related Apps (e.g. Stock holders trading applications, Amazon or eBay online payment stores.)

Figure 15 illustrates a fatal functionality in the source code of Anubis. The permission strings are utilized to check whether these permissions are granted or not after Anubis gains the accessibility access of the device. If one of these permissions is not granted, Anubis would request it again. Afterwards, the attackers have the 'devastated' privileges to devices which include but are not limited to:

- making calls and sending SMS.
- stealing storage content including contact list.
- capturing screenshots.
- toggling off and altering administration.
- receiving commands from attackers.
- collecting device and location information.
- Recording the audio

4.1.3 Phase 3: C&C Server

C&C servers serve as control and command centers that store the stolen data from victims, collect and send the commands from attackers. Most attackers implement cloud-based deployments, which is the core quarter for Anubis Trojan to steal data, spread malware, mock requests

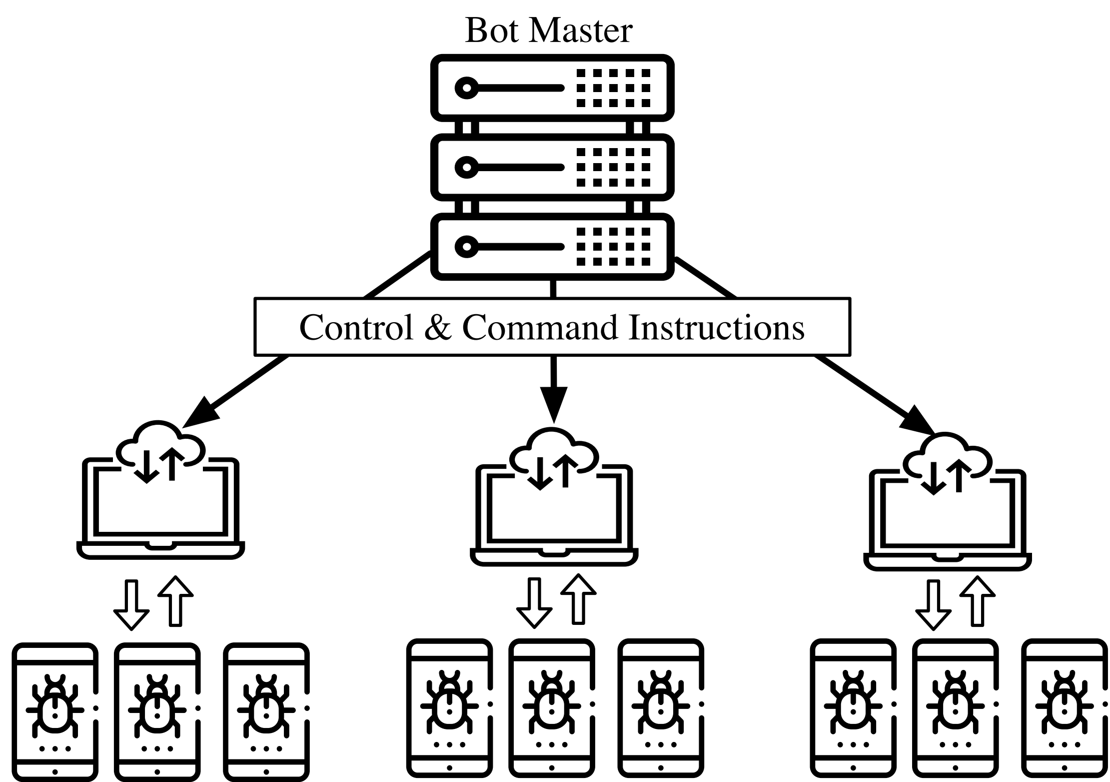


Figure 17: Anubis C&C Server Model

from the infected Android devices, containing the installed banking apps if any as well as whether any of these apps is running at the top of the process stack at the moment. Once certain condition is met, corresponding phishing webpage as well as some commands would be delivered to the infected devices, then the page would be popped up through webview API and overlay on top of the targeted app. Besides, Anubis would perform specific tasks based on the commands received. Most users might not notice that the overlay page layout and the real banking page layout are actually different. As soon as some careless users input their login credentials, the payload would invoke the keystroke module or take a screenshot stealthily and then upload the critical information to the Command and Control server. When malware authors collect the credentials, they would use them to log into the bank to perform a money transfer. However, most banks would be aware of such abnormal behavior thus challenging the operator for their authenticity by sending authentication code or directly calling the user. To deal with this, malware authors exploit SMS and call forwarding techniques to blind users and finally confirm the genuineness to the bank to make the transfer successful.

Furthermore, the C&C server can also instruct the compromised device through the RAT (Remote Access Tool) module carried by Anubis payload. The functionalitiess of these orders mainly include stealing local SMS and contacts info, sending SMS, requesting certain permissions, pushing notifications, call forwarding, sound recording, screen shooting, keylogging, file downloading and deleting as well as local data crypting and decrypting (after paying ransom).

4.1.4 Phase 4: Decrypting the encrypted

```
String dex_path = dec_arr(new byte[](42,15,44 ... 108,85,-7));
//image/files
```

Listing 1: A pseudo code snippet in an Anubis sample that decrypts a byte array into a path-like string

Most of the Anubis samples utilize a packer technique. To illustrate, after landing on the devices, the payload does not expose itself directly, but is mostly hidden in other resource directories like 'image', 'templates', 'assets' or even 'layout' with innocuous names (e.g. 'files','images', 'main.tpl', 'extend.bak' or 'about.xml'). The real and only class that is called is the 'Application' class, with methods that are responsible for functionalities like reading the real but encrypted classess.dex file in bytes, decrypting the bytes, outputing the decrypted bytes into certain path as a jar file, reflecting the run-time class(i.e. decrypted class), and deleting the output files(one .jar file and one .dex file) after loading the class with *java.lang.ClassLoader.loadClass* method. Wisely, the malware author encrypts all the used key methods' and classes' names as well as literals into byte arrays with RC4 routine taking a fixed byte array as the encryption key. An example of the obfuscated path literal for the real dex file can be seen in Listing 1. Therefore, to acquire the real payload, we can focus on either run-time file-removing API hooking or statically re-implement the decryption routine. Besides, the real C&C server address is usually fetched through reading and decrypting the base64-encoded[128] content from an intermediate web address. Interestingly, the commands received from the C&C server, the information sent to the server are encrypted with RC4[129] cipher, whose key is always hard-coded in the source code, which is actually not hard to find. What's more, a ransomware-like cryptor could encrypt all the local files with RC4 cipher using the same key. Actually, A more secure way for encryption and decryption could be using an asymmetric cipher.

4.2 Data and Analysis

Virustotal Intelligence [130] database is one of the largest commercial databases which collects most of the potentially malicious Android packages. However, due to it is high cost, it is not the best choice for our research. As an alternative, we collect our research data from other similar open-source mobile threat intelligence platforms with considerable APK (Android package) repository such as Koodous and Apklab.io [131]. Both Koodous and Apklab.io are the open-source platform which collected a large repository of APK samples for free downloading. However, Koodous is a collaborative platform which lacks systematic long-term maintenance.

As a result, some related samples are out of the range of their repository. Apklab.io is another mobile threat intelligence platform launched by Avast, a cybersecurity software company. It has collected big data of malicious applications and can satisfy the need of searching for all kinds of samples. However, unfortunately, it only allowed the invited users to download 10 samples per day. As a result, the combination of these two resources is a recommended way to accomplish sample collection. Specifically, on one hand, we use "tag: Anubis" on Koodous to find all the samples which have been labelled "Anubis" at least once by other analysts, then take advantage of their REST API to search for and download related samples. On the other hand, we search for and collect Anubis MD5, SHA1 or SHA2 (SHA256) hashes on cybersecurity-related blogs and websites, download the additional samples from Koodous and Apklab.io. We have collected samples captured by the platform from Jan.2018 to Jul.2019.

4.2.1 Anubis Roadmap

Malware becomes robust and multi-functional with the advanced versions. Although quite a number of repackaged and nearly clone contemporaries belong to the same family, they actually contribute little to the evolution of malware, since most repackaged malware seldom alters the implementation layout like APIs but mostly just change parameters (e.g server addresses, secret keys for cryption and decryption, etc.). Thus we investigate the evolution of Anubis through different perspectives.

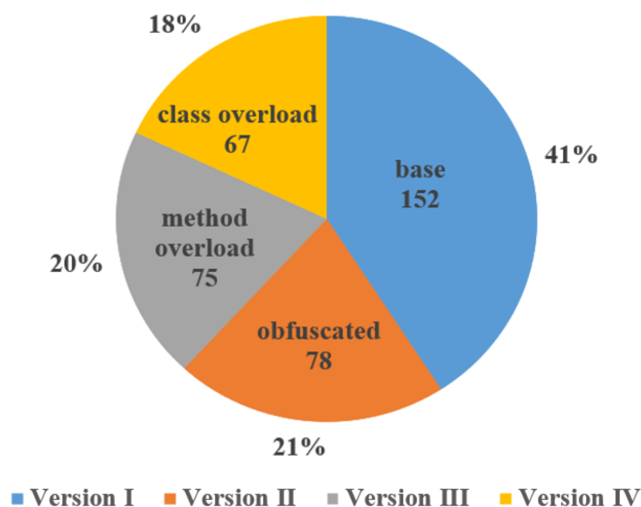


Figure 18: The breakdown of the Anubis samples regarding configurations and versions

To obtain the ground-truth labels for triaging different versions, we choose another more credible platform, Apkdetect, which specialises in Android malware analysis. Compared with those anti-virus engines that utilize a signature-based detection and labeling mechanism, Ap-

kdetect is able to generate the family name based on configurations extracted. As shown in Figure18, 41.3% of the samples collected belong to Version I which are identified with Bankbot Anubis or Anubis v1 configuration in Apkdetect, 21.2% belong to Version II with Anubis obfuscated Configuration, 19.6% belong to Version III with Anubis method overloaded configuration and the rest (17.9%) belong to Version IV with Anubis class overloaded Configuration. Version II is built upon the base version with method obfuscations, while Version III are inserted numerous trash codes to the added junk methods but little change is made to the classes, which is contradictory to Version IV, which is filled with junk classes and junk methods.

To obtain a more accurate birth date of one sample, we first download verbose analysis report from Virustotal [132] regarding each sample. Each report contains three kinds of useful information that is related to birth date, i.e. **validfrom** denoted in certificate, **compressed date time** and **first_seen** in Virustotal database. Since any one of the three could be biased to some extent, we exclude some invalid data like 1980-00-00, and then apply a majority vote for each report. For those reports where only two data remain, a mid-date of the two are used as the final birth date. The timeline of Anubis samples' emergence in regard to different versions are illustrated in Figure 19. Since the total number of each version differs with other versions, we just calculate the percentage of each time period regarding specific version, which depicts the trends of different versions more clearly. The graph suggests that the original version was developed relatively early, and remains existent for the whole 18 months, which might be due to

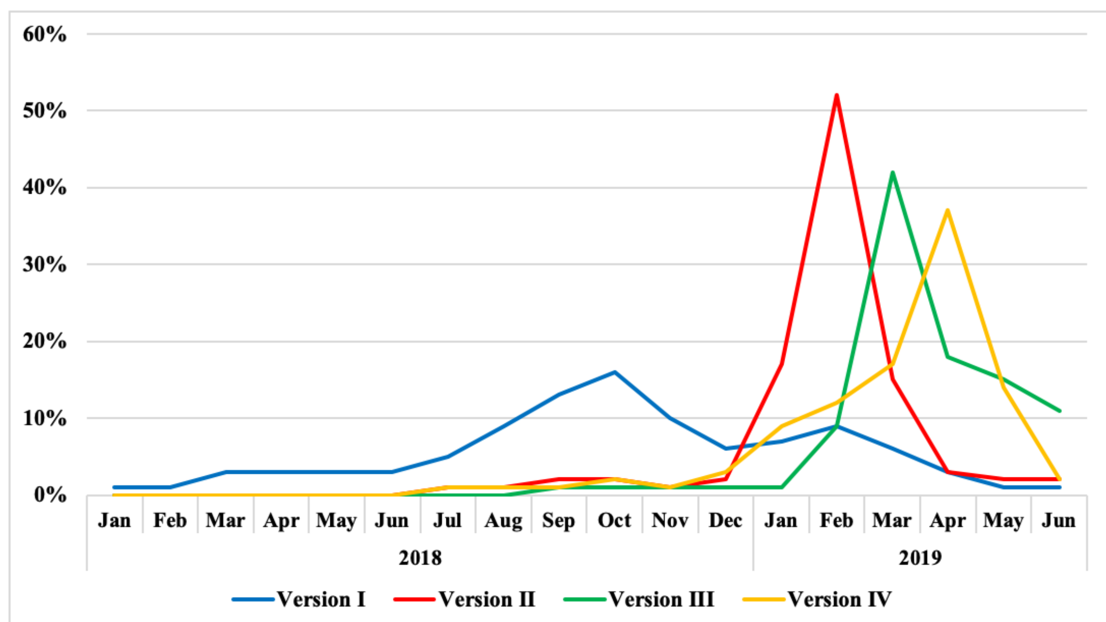


Figure 19: Emergence of different versions of Anubis from January 2018 to July 2019

the simplicity of the program for cloning and reproducing. Version II peaks on February 2019, 4 months after the peak month of Version I, whereas Version III and IV culminate consecutively on March 2019 and April 2019, respectively. To understand more details of its evolution, we perform several analysis and generate the findings as follows.

4.2.2 Development of Encryption

For instance, Anubis instruments the corresponding intermediate link address in one of the signature files (classes) for acquiring the real Control & Command server address. The first version of Anubis stores the URL string in plain text (Listing 2a), whereas Version II abuses RC4 and Base64 scheme to encrypt the string (Listing 2b), Version III exploits XOR(Exclusive OR with different integers in different classes) and Base64 scheme for encryption (Listing 2c), Version IV aggressively obfuscates the URL into byte array (Listing 2d), which significantly impedes static analysis especially static string-matching technique, since this version of Anubis not only adds thousands of junk classes but also inserts some unused byte arrays, useless loops and pointless calculations to conceal the actual referenced variables. Listing 2d is the code snippet after omitting those junk codes. Notably, both encryption processes for Version II and the later versions are implemented without using any encryption API (e.g. *android.util.Base64* or *Crypto.Cipher.ARC4*). Furthermore, some higher versions (Version II, III and IV) of Anubis encrypt a jpg-formatted image which is originally located in *res.drawable* directory into byte arrays in their source code. The resource_id of the image is called by *setImageResource(resource_id)* method for some of the Version I samples to trick user into deactivating Google Protect service on the device. Similar with other Version I samples, they also utilize android.widget.Toast class to pop up the phishing content.

Insights from Development of Encryption: As a Android trojan family, Anubis evolves to become more and more resilient to generic static reverse engineering analysis. Only by means of customizing the static analysis methods and/or introducing dynamic analysis can analysts extract the original meaningful strings.

4.2.3 AndroidManifest

AndroidManifest.xml functions as a prerequisite resolve table that declares information like requested modules(permission), the names of the classes that implement different components, the functionalities(intents) for each component, etc. As a result, we extract these significant parameters from all decompiled AndroidManifest.xml files and compare the parameters of the

```
String url_str = "https://twitter.com/Utriania5d86lni";
this.http = (HttpURLConnection) new URL(url_str).
            openConnection();
```

(a) Version I

```
String raw_str = "MmRkYmE ... jE0NjI=";
String url_str = dec_str_str(raw_str, "fduozenzsxh");
this.http = (HttpURLConnection) new URL(url_str).
            openConnection();
```

(b) Version II

```
String raw_str = "emZmYmE ... XdjZXc=";
String url_str = dec_str_int(raw_str, 18);
this.http = (HttpURLConnection) new URL(url_str).
            openConnection();
```

(c) Version III

```
byte[] bArr1 = new byte[]{(byte) 50, (byte) 58, ... (byte) 45, (byte
    ) 43};
byte[] bArr2 = new byte[]{(byte) 98, (byte) 78, (byte) 48};
String url_str = dec_byte_byte(bArr1, bArr2);
this.http = (HttpURLConnection) new URL(url_str).
            openConnection();
```

(d) Version IV

Listing 2: Pseudo code snippets that indicates the evolution of string obfuscation for intermediate link address

Table 10: Frequently used permissions by Anubis

android.permission.*	**Functionalities**	**Purpose of Usage**
ACCESS_FINE_LOCATION	Allows to determine as precise a location as possible from the available location providers	Functionality request
CALL_PHONE	Allows to initiate a phone call without going through the Dialer user interface for the user to confirm the call	Functionality request
READ_CONTACTS	Allows to read the user's contacts data	Functionality request
RECORD_AUDIO	Allows to record audio	Functionality request
SEND_SMS	Allows to send SMS messages	Functionality request
WRITE_EXTERNAL_STORAGE	Allows to write to external storage	Functionality request
READ_EXTERNAL_STORAGE	Allows to read from external storage	Functionality request
READ_PHONE_STATE	Allows read only access to phone state and a list of any PhoneAccounts registered on the device	Functionality request
READ_SMS	Allows to read SMS messages	Functionality request
RECEIVE_SMS	Allows to receive SMS messages	Functionality request
BROADCAST_SMS	Allows to broadcast an SMS receipt notification	Component enforcement
BROADCAST_WAP_PUSH	Allows to broadcast a WAP PUSH receipt notification	Component enforcement
SEND_RESPOND_VIA_MESSAGE	Allows to send a request to other applications to handle the respond-via-message action during incoming calls	Component enforcement
BIND_ACCESSIBILITY_SERVICE	Allows the application to run in the background and receive callbacks by the system	Component enforcement

Table 11: Shared intent filters by all versions of Anubis

Component	Generic action to be performed (android.*)
Activity	intent.action.SENDTO
	intent.action.SEND
Service	intent.action.RESPOND_VIA_MESSAGE
	accessibilityservice.AccessibilityService
Receiver	provider.Telephony_SMS_DELIVER
	intent.action.PACKAGE_ADDED
	intent.action.PACKAGE_REMOVED
	intent.action.QUICKBOOT_POWERON
	intent.action.BOOT_COMPLETED
	intent.action.DREAMING_STOPPED
	intent.action.SCREEN_ON
	intent.action.USER_PRESENT
	intent.action.EXTERNAL_APPLICATIONS _AVAILABLE
	provider.Telephony.SMS_RECEIVED
	net.wifi.WIFI_STATE_CHANGED
	net.conn.CONNECTIVITY_CHANGE (!)

same type to explore the differences among different versions of Anubis.

<**android.permission**> module acts as a mechanism to safeguard sensitive data on the device. Ten sensitive permissions in Table 10 are required to be granted by users during runtime, otherwise the related functionalities will not be provided. It is noticed that Version III and IV almost request largely identical permissions.In addition, the top 6 permissions are what Anubis values are checked again with *checkCallingOrSelfPermission* method as the Anubis running on. Furthermore, the last four permissions are to enforce the components (i.e. activities, services, broadcasts and/or content providers) which are related to the telecommunication with the remote C&C server controlled by the attackers. Therefore, these permissions function as the necessary infrastructure for launching further attacks.

<**intent-filter**> define how the corresponding components can be started by specifying the action, data and/or category information of intents to accept. Only those intents that match the declared intent filters can be delivered to the corresponding components. These specified parameters can relatively reflect the capabilities of the app therefore worth investigating. Table 11 shows that the evolution of such trojan does not witness any change in regard to the intent

filters instructed in AndroidManifest. Rather, Anubis register a broadcast receiver that accepts the android.provider.Telephony.SMS_DELIVER action to receive incoming SMS messages, another receiver for the android.provider.Telephony.WAP_PUSH_DELIVER action to receive MMS messages, an activity for the android.intent.action.SEND (or android.intent.action.SENDTO) action so as to send SMS/MMS messages and a service handling the RESPOND_VIA_MESSAGE action to shoot quick response messages to incoming callers. These declarations pave the way for Anubis to become another default option of messaging application. The main abusive entry point of Anubis is the receiver which declares 12 instances of action including system boot(both cold boot BOOT_COMPLETED and warm boot QUICKBOOT_POWERON), network condition change, installing and uninstalling of APKs, device (interaction) activation and more importantly SMS receiving.

Although both the action event provider.Telephony.SMS_RECEIVED and the action event provider.Telephony.SMS_DELIVER both indicate that a new SMS has been received by the device, the former is intended for all registered receivers to accept as a notification while the latter is only delivered to the system's default SMS application. There is an action event that has been declared twice in a single receiver for all the samples, net.conn.CONNECTIVITY_CHANGE, which largely suggests that most Anubis samples could be developed in an automated progress (inline with Finding 6 of [133]).

4.2.4 Hard-coded API classes and methods

API is short for Application Program Interface. Google provides a series of built-in classes and methods for Android developers. These methods and classes act as a convenient interface for developers to call, which significantly accelerates the developing process. These API calls are the core part that forms the functionalities of an application. From Version II to Version IV, a module that utilizes the accelerometer sensor module to collect and send the information about whether the device is still or not is added. Such module actually upgrades Anubis into a more furtive hunter. Also, another socket module is added for more secure network traffic from Version II to higher versions. The Method *getInputStream*, *getOutputStream* and *getLocalAddress* are from Class *java.net.Socket*, while *getAddress* originates from Class *java.net.InetAddress* and accept is the method from Class *java.net.ServerSocket*. Method *getLocalPort* is called by both Socket and ServerSocket class. Besides, the *registerReceiver* and *unregisterReceier* method is called for a new SMS spam module. Compared with Version III which to some extent hinders the static hashing detection technique of some anti-virus vendors by hiding a resource image, Version IV perfects its stealthiness and anti-sandbox technique. The method *isignoringbattery-optimization* is used for checking whether any battery optimization technique is being applied

Table 12: API classes abused by different versions of Anubis

Version I	82 API classes
Added	
Version II	android.hardware.Sensor, android.hardware.SensorEvent, android.hardware.SensorEventListener, android.hardware.SensorManager, java.net.Socket, java.net.ServerSocket, java.net.InetAddress, android.widget.Imageview
Added	
Version III	android.graphics.BitmapFactory
Added	
Version IV	—

by the device, which will alert users about the existence of the long-time loops in the program; The method *getExternalStorageState* is an indicator of whether the infected device could be an virtual device or honeypot; *onWindowFocusChanged* method is triggered to send a broadcast of the intent action.CLOSE_SYSTEM_DIALOGS which helps to set up a silent "environment" for the subsequent XSS webpage injection. Based on the number of the methods used, the change of API methods between any two versions is less than 1.9%.

The difference of API classes between different versions is shown in Table 12. Since there is no deleted API class, we list the added ones. The change of API classes between any two versions is less than 10%. Apart from investigating the categories of API methods and classes used, the frequence of different API methods and classes regarding different versions are also collected and compared. For each version, we first extract the frequence of all the API classes and methods used for each sample, and then average the data within the same version of Anubis.

As shown in Figure 20, Version IV aggressively abuses the *AccessibilityService*, *Activity*, *Service* and *ViewGroup* API classes, compared with other versions, which again indicates that the Anubis authors become more inclined to build a user-friendly interface and make more efforts to create and instantiate more components. Table 13 shows great distinctions between Version III (IV) and Version I (II). The gap between them regarding Method *toString* and *decode* are roughly equivalent. The drastic increase of the three methods is attributed to the aggressive encryption technique abused by Version III and IV. Almost all the exposed (un-

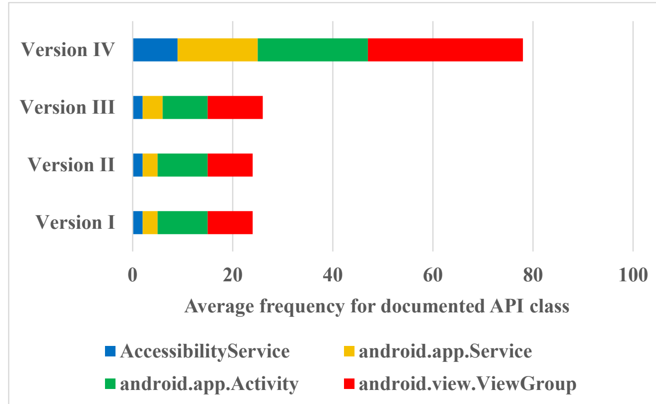


Figure 20: Top 4 frequent API classes used by four versions of Anubis

encrypted) strings are encrypted with base64, XOR or RC4 scheme. Those long strings are broken into short ones, and get concatenated using the method append and toString, then utilize *Base64.decode* for preliminary decryption.

Table 13: Top 3 documented API methods that are the most different in terms of their average frequency

Version No.	**append**	**toString**	**decode**
Version I	241	79	0
Version II	1636	648	0
Version III	4072	3044	2420
Version IV	4115	3191	2463

Due to the fact that some samples are actually not obfuscated, the names of the decompiled classes just indicate corresponding purposes. Thus we can take advantage of such sample as a representative for comparing modules implemented in different versions. As Figure 21 illustrates, Version II adds a port forwarding IntentService module and a ServicePedometer module which are invoked whenever Anubis receives the corresponding commands from the server. Additionally, Version III as well as Version IV add an SMS spam module for further spreading, which can also be triggered by the C&C server.

Insights from Hard-coded API classes and methods: As Anubis evolves, it becomes more and more obfuscated as well as accurate (follow the one-shot-one-kill principle, i.e., always be extremely cautious unless the infected environment meets some requirements). Although that more junk code (unused code) gets injected into the source code, and that

more complex obfuscation techniques are used on the source code could lead to higher overhead costs, the source code effectively gets well-protected from being reverse engineered. Only through combining the customized static analysis and test-device-based dynamic analysis can analysts unveil the true nature of Anubis.

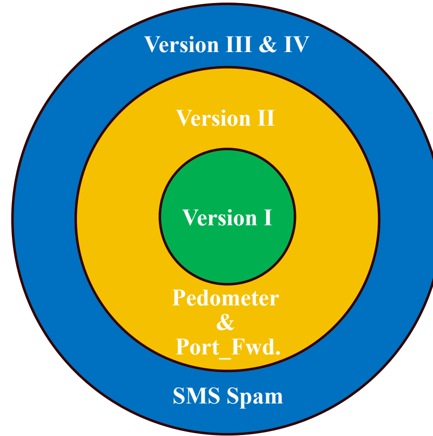


Figure 21: The name of the newly implemented module for new versions

4.2.5 Opcode Sequences

Smali [134] is an assembler for the binary Dalvik bytecode [135], which is the format that Android operating system cognises. Given a raw APK sample, we can use some reverse engineering tools like Apktool to obtain a close-to-source-code human-readable Smali code. After obtaining the corresponding Smali code, some opcodes are essential for modeling and representing the program. We only take those essential operation codes into consideration and discard the parameters. The formulated operation codes are shown in Table 14.

Table 14: Seven types of Dalvik Opcodes

Category	Opcode Constants	Examples
M	MOVE	MOVE_RESULT, MOVE_WIDE
R	RETURN	RETURN_VOID, RETURN_OBJECT
G	GOTO	GOTO, GOTO_16
I	IF	IF_EQ, IF_GT
T	GET	AGET_OBJECT, IGET_CHAR
P	PUT	APUT_BOOLEAN, IPUT_SHORTE
V	INVOKE	INVOKE_DIRECT, INVOKE_SUPER

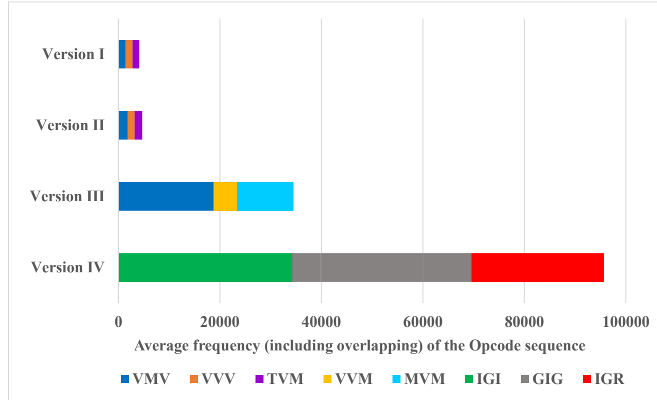


Figure 22: Top 3 frequent (including overlapping) documented Opcode sequences of four versions of Anubis

In order to separate the entire program into methods, we check the signal that indicates the start (i.e. **.method**) and end (i.e. **.end method**) of a given method and add a delimiter when the end of a method is met. We first decompile all the samples and unpack those packed classes if existent, and then aggregate all the collected classes prior to calculating different frequencies for each sample under a certain version, then average them and repeat such step for other versions. To investigate whether the average frequencies of the 3-gram Opcode sequences differ among different versions, Figure 22 illustrates the top 3 frequent documented Opcode sequences in each version of Anubis, Version I and II have identical top 3 Opcode sequences, i.e. VMV, VVV and VTM, while VMV, MVM and VVM have more occurences for Version III. Since Version IV is full of useless conditional statements, jumping statements and method calls, IGI, GIG and GIR are the 3 top-ranking Opcode sequences. Therefore, the average frequency of Opcode sequences have changed significantly from Version II to Version IV, whereas little discrepancy regarding the frequencies of Opcode sequences can be seen between Version I and Version II.

Insights from Opcode Sequences: Due to the massive injection of junk code, a great number of meaningless control flows are added. For the purpose of sample representation, a byte-code-level operation code modelling hence becomes heavily biased. So in order to get rid of the noises and generate accurate representation, the flow of execution (call flow) has to be considered before modelling.

5 An Observation for Android Malware Detection

With various approaches that rely on machine learning model or neural network model building, the performance of identifying Android malware is becoming more promising. However, the efficiencies of those approaches seem to have been sacrificed. With some sophisticated models, even if the features extracted are light-weighted, there are still many resources consumed during training. On the contrary, the real-life scenario of a security analyst is that he/she needs to needs to label or decide a raw sample as fast as possible, since there are tens or even hundreds of raw samples that need taking care of. Considering such gap, the writer tries to merely use some lightweight features to accomplish such task without using machine learning or neural network techniques.

Table 15: The statistics of the APK samples in the dataset

Source	No. of samples	Cluster
VirusTotal	15000	Malicious
Googld Play	7500	Benign
Yingyongbao	7500	Benign

The dataset are composed of two clusters (malicious cluster and benign cluster). The samples in the malicious cluster are randomly collected from the Android malware repository from the year of 2020 shared by VirusTotal [132], while half of the samples in the benign cluster are the most trending applications crawled from the top free applications in the official application store Google Play [136], the other half are the top applications crawled from the download rankings in the Tencent Application Store [137]. The Tencent Application Store is the third-party Android application store with the most active users in China [138]. Table 15 displays the breakdown of the dataset used. The experimented approach is based on a simple principle, i.e. most benign APKs are not generated by SaaS (Software as a Service), which means that they are not churned out, while malicious Apks are just the opposite, malware developers need to massively produce malware on one hand, to maximise the chance of harvesting revenues from victims and on the other hand, to maximise the living rate of the samples against removals from App stores. Thus, the name of the malicious APK that displays to the user need to be attracting, however the package name of the APK can not always be relative to the App name. Similarly, the certificate information is very possible to be automatically generated thus also irrelative to the package name. Therefore, the writer writes a Python script to extract app_name (**AN** for short) from

Manifest or res/strings, package name (**PN** for short) from Manifest, and owner and issuer info ((**CI** for short)) from the RSA file from META-INF folder by using Androguard and keytool.

Figure 23 displays the workflow of identifying a given APK as suspicious or not. First, the AN, PN and CI are extracted and parsed into tokens. Considering the fact that some ANs are not English, the names would be first normalized, which tries to convert unstandard English letters (e.g. â) into English letter. Afterwards, some generic tokens that frequently appear but actually do not hold much information in PN like app, com, de, are discarded. Besides, If some token is a word for certain digit (e.g. two for 2), a new synthetic item is created and added. For example, if the AN of an APK is **Life of Two**, then a synthetic item **Life of 2** is created and added as another AN. Besides, considering some APKs that have long APP names and cert issuer/owner names, acronyms of AN, CI and abnormal tokens, together with filtered PN tokens, AN tokens and CI tokens are lowercased and added into set **R**, **P**, **A** and **C**, respectively. Here, abnormal tokens refer to the tokens that contains more than one nonconsecutive uppercased letters. These nonconsecutive uppercased letter would be picked out, concatenated and be regarded as an acronym token in the acronym token set **R**. For example, if a token of CI of an APK is **BookMyRoom**, this token would be regarded as an abnormal token and be parsed into a new acronym token **BMR** and added into set **R**. Besides, this token is also parsed again into **Book**, **My** and Room and added into set **C**. And the AN **Life of Two** would be acronymized into token **LoT** and **Lo2**. And these two tokens would be added into the **R** set. At the last step in Phase 1, all the tokens in set **P** would be compared with each token in set **A**, **C** and **R**. If a match appears and the two matched tokens are from **P** and **R**, then such APK can be considered as unsuscpicious. Here a match is defined as one token is contained in and equals to another token. For example, Token **mad** and Token madboy is considered a match, since String **mad** is contained in **madboy**. If a match does not come from the comparison of **P** and **R**, then the length of the matched token is taken into consideration. If the length of matched token is more than 2 letters, then such APK can also be regarded as unsuspicious. If none of the conditions mentioned are met, then the analysis would come to the second phase.

The second phase first checks whether set **A** contains any token that is not made with ascii characters. If so, such token is uploaded and translated to English with Google translate, since Google translate is able to automatically detect the language. Then both the phonetic spelling of the token and the translation result would be extracted, tokenized and lowercased into set **T**. Then each token in **P** would be caompared with each token in **T** to find whether there is any match and the length of the matched token is more than 2 letters, it would be considered a valid match thus the APK being considered as unsuspicious. If neither of the conditions fails to be met, then such APK would also be considered as suspicious. Finally, the CIC dataset is

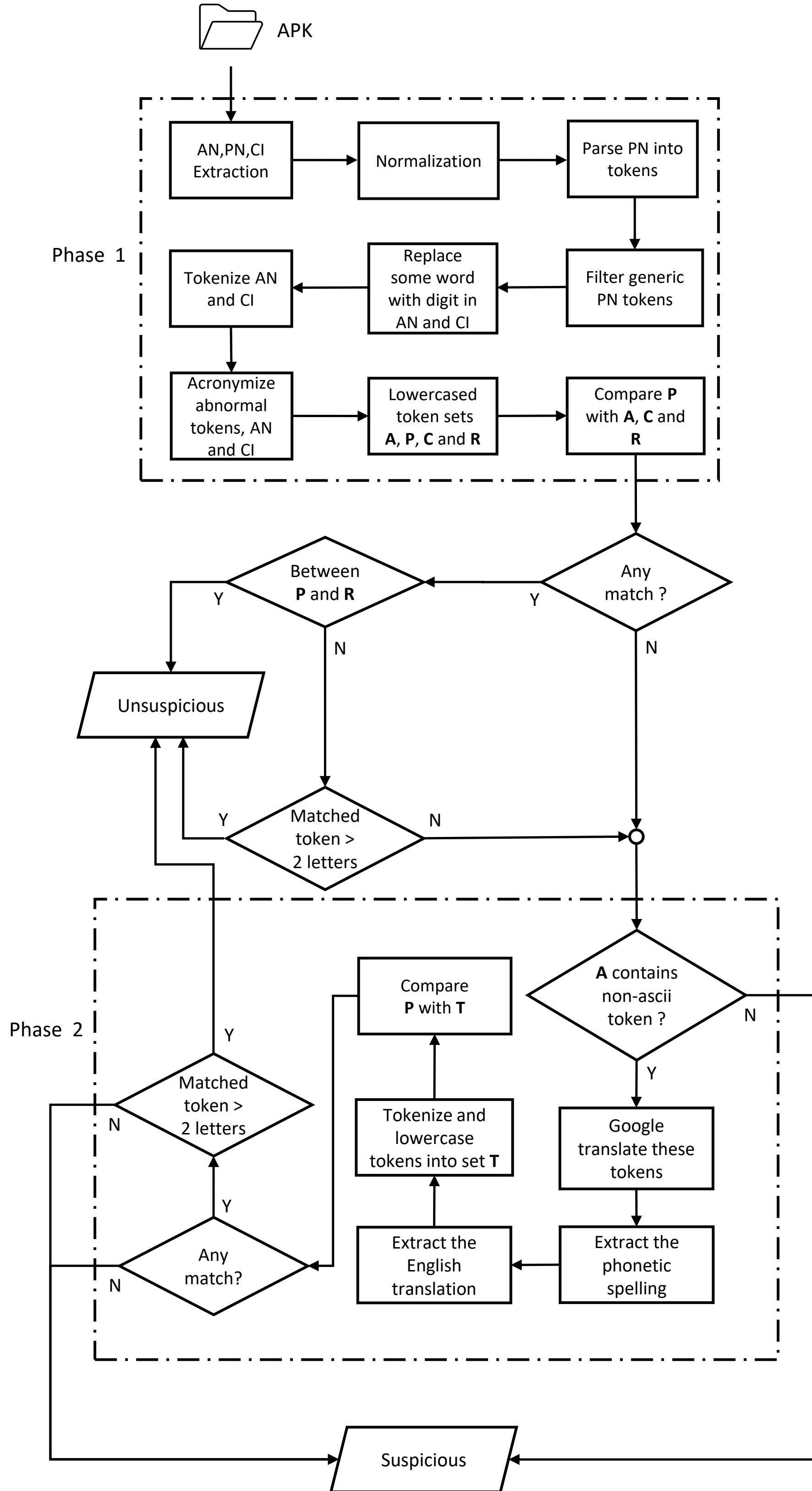


Figure 23: The workflow of identifying suspicious and unsuspicious APKs

Table 16: The confusion matrix of the results

	Actual Positive	Actual Negative
Predicted Positive	14930	738
Predicted Negative	70	14262

used to verify whether such method is satisfactory or not. Table 16 is a confusion matrix that is used to evaluate the results of the approach. Here, positive is defined as malicious, and negative benign. The results are satisfactory based on the following evaluation metrics.:

- **TPR (True Positive Rate)**:
  TPR = TP / (TP + FN) = 14930 / (14930 + 70) = 99.5%
- **FPR (False Positive Rate)**:
  FPR = FP / (FP + TN) = 738 / (738 + 14262) = 4.9%
- **FNR (False Negative Rate)**:
  FNR = FN / (FN + TP) = 70 / (70 + 14930) = 0.5%
- **TNR (True Negative Rate)**:
  TNR = TN / (TN + FP) = 14262 / (14262 + 738) = 95.1%
- **ACC (Accuracy)**:
  ACC = (TP + TN) / (TP + TN + FP + FN) = (14930 + 14262) / (14930 + 14262 + 738 + 70) = 97.3%
- **Precision**:
  Precision = TP / (TP + FP) = 14930 / (14930 + 738) = 95.3%
- **F1 score**:
  F1 = 2TP / (2TP + FP + FN) = 2*14930 / (2*14930 + 738 + 70) = 97.4%
- **AUC-ROC score**: AUC-ROC score = 0.93
- **AUC-ROC curve**: Figure 24

However, what could act as a threat to such approach can be inferred from some wrongly classified APKs. For example, the details of a false positive APK are:

- **AN**: Gadget Guardian
- **PN**: com.lookout
- **CI**: James Burgess, Los Angeles, Flexilis, Mobile Security

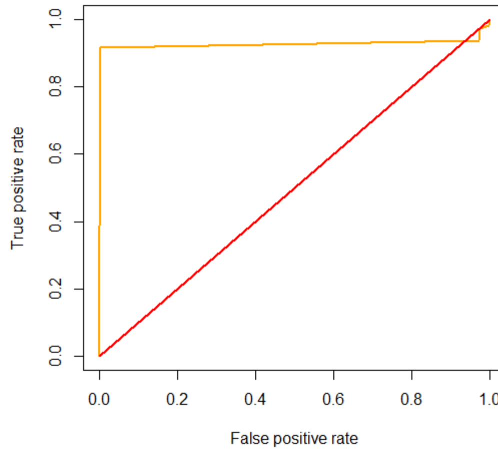


Figure 24: The AUC-ROC curve of the results

The PN actually has nothing to do with AN or CI just from the similarity of spelling, although the word lookout does have some semantic similarity with the word security, and guardian. Similarly, another false positive APK is misclassificed for the same reason. Its details are:

- **AN**: Alarmy
- **PN**: droom.sleepIfUCan
- **CI**: 2

The AN also seems irrelevant with PN. However, **sleepIfUCan** actually indicates that the APK would be an alarm, which is then related with AN. Moreover, Phase 2 of using the translation results from Google translate is not always a right direction. Take such false positive APK as an example:

- **AN**: 节目通
- **PN**: com.example.suishoukan
- **CI**: CN, Jack, Jack, Jack

By typing in the AN in Google translate, the phonetic spelling of it is **Jiémù tōng**, translation result is **Show pass**. There is no connection between such information with PN. The reason for this is that **suishoukan** in PN is actually another phonetic spelling for Mandarin phrase 随手看 . The real meaning of 节目通 is TV program master, while 随手看 means to watch TV programs at hand.

Therefore, restricted by the inaccurate translation of Google translate and interfered by some self-designed package names, this approach still needs more optimization that may include

adding some measurement of semantic similarity between PN and AN tokens, which would not be a simple task, especially considering different cultural and language backgrounds.

6 Conclusion and Future Work

This thesis collects and investigates the recent Android trojans, and profiles them both statically and dynamically in order to illustrate what techniques current Android malware are exploiting to launch attacks towards users. It should be noticed that most trojans can only function well with a live C&C server. If there is no C&C server alive, even though trojans are able to seize top privilege to perform malicious actions, malware authors are not able to harvest any profit from victims. After all, what motivates most trojan authors to publish and distribute trojans is the potential revenues. Planting aggressive adwares on the infected devices is one method of gaining a small amount of profits, but doing so would also have to take the risks of immediately arouse the victim's suspicions as well as annoy victims, As a consequence, advanced countermeasures would be taken to remove the trojan. By contrast, if trojan authors focus on preparing a well-designed scheme to make victims believe the integrity, then it would be highly possible that such strategy gain a proming profit. This, from another perspective explains why the trojans need to evade both static analysis and dynamic analysis as long as possible.

Besides, this thesis also looks deeper into Anubis, a popular Android trojan family that severely infects Android platforms to understand how it has evolved, This work, finished by the writer, has been accepted by and presented at the 7th International Conference on Dependable Systems and Their Applications. The Anubis family has evolved into different versions, where technical details vary while the core code blocks of the family are nearly identical. Their phishing, overlay and code obfuscation mechanisms have been described. Such information will be useful for past, present and future victims. At the same time, how new complex versions emerge based on the old versions has also been illustrated.

As for possible directions of future work, it should be valuable to investigate how to employ hybrid (both static and dynamic) analysis more efficiently to deal with different payload loading patterns (i.e. dynamic loading and native loading), since some APK reinforcement products like Qihoo 360, are able to pack the real payload with multiple layers of encryption, which would make purely static analysis nearly ineffective. On the other hand, dynamic analysis especially in the sandbox environment, some parameters reflecting the infrastructure need to be dynamic or at least not fixed, thus making it possible for trojans that check device-info to perform their real functionalities smoothly.

References

[1] apkdetect, https://app.apkdetect.com/, @pr3wtd, accessed: February 2020.

[2] "Mobile operating system market share worldwide, 2020," https://gs.statcounter.com/os-market-share/mobile/worldwide, 2020.

[3] "Number of available applications in the google play store from december 2009 to june 2020," https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, 2020.

[4] "Android security & privacy 2018 year in review," https://source.android.com/security/reports/Google_Android_Security_2018_Report_Final.pdf, 2019.

[5] E. Alepis and C. Patsakis, "Hey doc, is this normal?: exploring android permissions in the post marshmallow era," in *International Conference on Security, Privacy, and Applied Cryptography Engineering.* Springer, 2017, pp. 53–73.

[6] D. J. Tan, T.-W. Chua, and V. L. Thing, "Securing android: a survey, taxonomy, and challenges," *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 1–45, 2015.

[7] P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, M. S. Gaur, M. Conti, and M. Rajarajan, "Android security: a survey of issues, malware penetration, and defenses," *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[8] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy.* IEEE, 2014, pp. 409–423.

[9] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *2012 IEEE symposium on security and privacy.* IEEE, 2012, pp. 95–109.

[10] J. Gamba, M. Rashed, A. Razaghpanah, J. Tapiador, and N. Vallina-Rodriguez, "An analysis of pre-installed android software," *arXiv preprint arXiv:1905.02713*, 2019.

[11] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 226–237.

[12] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith, "Sok: Lessons learned from android security research for appified software platforms," in *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 2016, pp. 433–451.

[13] S.-T. Sun, A. Cuadros, and K. Beznosov, "Android rooting: Methods, detection, and evasion," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2015, pp. 3–14.

[14] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013, pp. 623–634.

[15] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *2012 19th Working Conference on Reverse Engineering*. IEEE, 2012, pp. 83–92.

[16] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones." in *NDSS*, vol. 14, 2012, p. 19.

[17] J. Oberheide and C. Miller, "Dissecting the android bouncer," *SummerCon2012, New York*, vol. 95, p. 110, 2012.

[18] P. Mateti, "Viruses, worms and trojans," 2002.

[19] T. M. Chen and J.-M. Robert, "The evolution of viruses and worms," *Statistical methods in computer security*, vol. 1, 2004.

[20] N. Weaver, V. Paxson, S. Staniford, and R. Cunningham, "A taxonomy of computer worms," in *Proceedings of the 2003 ACM workshop on Rapid malcode*, 2003, pp. 11–18.

[21] A.-D. Schmidt, H.-G. Schmidt, L. Batyuk, J. H. Clausen, S. A. Camtepe, S. Albayrak, and C. Yildizli, "Smartphone malware evolution revisited: Android next target?" in *2009 4th International conference on malicious and unwanted software (MALWARE)*. IEEE, 2009, pp. 1–7.

[22] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 3–14.

[23] M. Zheng, P. P. Lee, and J. C. Lui, "Adam: an automatic and extensible platform to stress test android anti-virus systems," in *International conference on detection of intrusions and malware, and vulnerability assessment*. Springer, 2012, pp. 82–101.

[24] S. Peng, S. Yu, and A. Yang, "Smartphone malware and its propagation modeling: A survey," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 925–941, 2013.

[25] H. Le Thanh, "Analysis of malware families on android mobiles: detection characteristics recognizable by ordinary phone users and how to fix it," *Journal of Information Security*, vol. 2013, 2013.

[26] K. J. Abela, D. K. Angeles, J. D. Alas, R. J. Tolentino, and M. A. Gomez, "An automated malware detection system for android using behavior-based analysis amda," *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, vol. 2, no. 2, pp. 1–11, 2013.

[27] X. Xia, C. Qian, and B. Liu, "Android security overview: A systematic survey," in *2016 2nd IEEE International Conference on Computer and Communications (ICCC)*. IEEE, 2016, pp. 2805–2809.

[28] M. Karresand, "A proposed taxonomy of software weapons," 2002.

[29] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *European symposium on research in computer security*. Springer, 2014, pp. 163–182.

[30] V. C. Roman Unuchek, "Mobile malware evolution 2015," https://securelist.com/mobile-malware-evolution-2015/73839/, 2016.

[31] R. Unuchek, "Mobile malware evolution 2016," https://securelist.com/mobile-malware-evolution-2016/77681/, 2017.

[32] R. Unuchek, "Mobile malware evolution 2017," https://securelist.com/mobile-malware-review-2017/84139/, 2018.

[33] V. Chebyshev, "Mobile malware evolution 2018," https://securelist.com/mobile-malware-evolution-2018/89689/, 2019.

[34] V. Chebyshev, "Mobile malware evolution 2019," https://securelist.com/mobile-malware-evolution-2019/96280/, 2020.

[35] S. Komatineni and D. MacLean, "Understanding android resources," in *Pro Android 4*. Springer, 2012, pp. 51–78.

[36] Malwarebytes, "2017 state of malware report," https://kitedistribution.co.uk/wp-content/uploads/2017/03/StateofMalware_Report_ønal_PT.pdf.

[37] H. Pieterse and M. S. Olivier, "Android botnets on the rise: Trends and characteristics," in *2012 information security for South Africa*. IEEE, 2012, pp. 1–5.

[38] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: detecting malicious apps in official and alternative android markets.” in *NDSS*, vol. 25, no. 4, 2012, pp. 50–52.

[39] D. Maslennikov, “Carberp-in-the-mobile, 2012.”

[40] A. Cherepanov and R. Lipovsky, “Hesperbot—a new, advanced banking trojan in the wild,” 2013.

[41] A. Atzeni, F. Diaz, F. Lopez, A. Marcelli, A. Sanchez, and G. Squillero, “The rise of android banking trojans,” *IEEE Potentials*, vol. 39, no. 3, pp. 13–18, 2020.

[42] R. Yu, “Ginmaster: a case study in android malware,” in *Virus bulletin conference*, 2013, pp. 92–104.

[43] A. Martin, J. Hernandez-Castro, and D. Camacho, “An in-depth study of the jisut family of android ransomware,” *IEEE Access*, vol. 6, pp. 57 205–57 218, 2018.

[44] A. Coletta, V. Van Der Veen, and F. Maggi, “Droydseuss: A mobile banking trojan tracker (short paper),” in *International Conference on Financial Cryptography and Data Security*. Springer, 2016, pp. 250–259.

[45] Y. Hu, H. Wang, Y. Zhou, Y. Guo, L. Li, B. Luo, and F. Xu, “Dating with scambots: Understanding the ecosystem of fraudulent dating applications,” *IEEE Transactions on Dependable and Secure Computing*, 2019.

[46] N. I. Aminuddin and Z. Abdullah, “Android trojan detection based on dynamic analysis,” *Advances in Computing and Intelligent System*, vol. 1, no. 1, 2019.

[47] C. Bai, Q. Han, G. Mezzour, F. Pierazzi, and V. Subrahmanian, “Dbank: Predictive behavioral analysis of recent android banking trojans,” *IEEE Transactions on Dependable and Secure Computing*, 2019.

[48] Y. Zhang, G. Xiao, Z. Zheng, T. Zhu, I. Tsang, and Y. Sui, “An empirical study of code deobfuscations on detecting obfuscated android piggybacked apps,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 914–926.

[49] D. Wu, J. Liu, Y. Sui, S. Chen, and J. Xue, “Precise static happens-before analysis for detecting UAF order violations in android,” in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 276–287.

[50] Y. Sui, Y. Zhang, W. Zheng, M. Zhang, and J. Xue, "Event trace reduction for effective bug replay of android apps via differential GUI state analysis," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 1095–1099.

[51] Y. Tang, H. Wang, X. Zhan, X. Luo, Y. Zhou, H. Zhou, Q. Yan, Y. Sui, and J. W. Keung, "A systematical study on application performance management libraries for apps," *IEEE Transactions on Software Engineering*, 2021.

[52] L. Wang, R. He, H. Wang, P. Xia, Y. Li, L. Wu, Y. Zhou, X. Luo, Y. Sui, Y. Guo *et al.*, "Beyond the virus: a first look at coronavirus-themed android malware," *Empirical Software Engineering*, vol. 26, no. 4, pp. 1–38, 2021.

[53] Y. Zhang, Y. Sui, and J. Xue, "Launch-mode-aware context-sensitive activity transition analysis," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 598–608.

[54] L. Li, T. F. Bissyandé, and J. Klein, "Rebooting research on detecting repackaged android apps: Literature review and benchmark," *IEEE Transactions on Software Engineering*, vol. 47, no. 4, pp. 676–693, 2019.

[55] J. Gao, L. Li, P. Kong, T. F. Bissyandé, and J. Klein, "Understanding the evolution of android app vulnerabilities," *IEEE Transactions on Reliability*, 2019.

[56] Y. Zhao, L. Li, H. Wang, H. Cai, T. F. Bissyandé, J. Klein, and J. Grundy, "On the impact of sample duplication in machine-learning-based android malware detection," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 3, pp. 1–38, 2021.

[57] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "Androsimilar: robust statistical feature signature for android malware detection," in *Proceedings of the 6th International Conference on Security of Information and Networks*, 2013, pp. 152–159.

[58] M. Zheng, M. Sun, and J. C. Lui, "Droid analytics: a signature based analytic system to collect, extract, analyze and associate android malware," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*. IEEE, 2013, pp. 163–171.

[59] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 627–638.

[60] R. Sato, D. Chiba, and S. Goto, "Detecting android malware by analyzing manifest files," *Proceedings of the Asia-Pacific Advanced Network*, vol. 36, no. 23-31, p. 17, 2013.

[61] Y. Tang, Y. Sui, H. Wang, X. Luo, H. Zhou, and Z. Xu, "All your app links are belong to us: understanding the threats of instant apps based attacks," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 914–926.

[62] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, "Performance evaluation on permission-based detection for android malware," in *Advances in Intelligent Systems and Applications-Volume 2*. Springer, 2013, pp. 111–120.

[63] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "Puma: Permission usage to detect malware in android," in *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*. Springer, 2013, pp. 289–298.

[64] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, K. Rieck, and C. Siemens, "Drebin: Effective and explainable detection of android malware in your pocket." in *Ndss*, vol. 14, 2014, pp. 23–26.

[65] A. Desnos *et al.*, "Androguard-reverse engineering, malware and goodware analysis of android applications," *URL code. google. com/p/androguard*, vol. 153, 2013.

[66] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and classifying android malware using static analysis along with creator information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, 2015.

[67] D. Octeau, S. Jha, M. Dering, P. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale android inter-component analysis," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 469–484.

[68] W. Tang, G. Jin, J. He, and X. Jiang, "Extending android security enforcement with a security distance model," in *2011 International Conference on Internet Technology and Applications*. IEEE, 2011, pp. 1–4.

[69] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[70] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, “Droidmat: Android malware detection through manifest and api calls tracing,” in *2012 Seventh Asia Joint Conference on Information Security*. IEEE, 2012, pp. 62–69.

[71] J. Hou, M. Xue, and H. Qian, “Unleash the power for tensor: A hybrid malware detection system using ensemble classifiers,” in *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*. IEEE, 2017, pp. 1130–1137.

[72] E. R. Wognsen, H. S. Karlsen, M. C. Olesen, and R. R. Hansen, “Formalisation and analysis of dalvik bytecode,” *Science of Computer Programming*, vol. 92, pp. 25–55, 2014.

[73] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, “Detecting repackaged smartphone applications in third-party android marketplaces,” in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012, pp. 317–326.

[74] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, “Crowdroid: behavior-based malware detection system for android,” in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, 2011, pp. 15–26.

[75] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, ““andromaly”: a behavioral malware detection framework for android devices,” *Journal of Intelligent Information Systems*, vol. 38, no. 1, pp. 161–190, 2012.

[76] M. Zhao, F. Ge, T. Zhang, and Z. Yuan, “Antimaldroid: An efficient svm-based malware detection framework for android,” in *International Conference on Information Computing and Applications*. Springer, 2011, pp. 158–166.

[77] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones,” *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 1–29, 2014.

[78] L. K. Yan and H. Yin, “Droidscope: Seamlessly reconstructing the {OS} and dalvik semantic views for dynamic android malware analysis,” in *Presented as part of the 21st {USENIX} Security Symposium ({USENIX} Security 12)*, 2012, pp. 569–584.

[79] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, “Context-aware, adaptive, and scalable android malware detection through online learning,” *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 157–175, 2017.

[80] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," in *2013 9th international wireless communications and mobile computing conference (IWCMC)*. IEEE, 2013, pp. 1666–1671.

[81] Y. Aafer, W. Du, and H. Yin, "Droidapiminer: Mining api-level features for robust malware detection in android," in *International conference on security and privacy in communication systems*. Springer, 2013, pp. 86–103.

[82] S. Sheen, R. Anitha, and V. Natarajan, "Android based malware detection using a multifeature collaborative decision fusion approach," *Neurocomputing*, vol. 151, pp. 905–912, 2015.

[83] R. Vinayakumar, K. Soman, P. Poornachandran, and S. Sachin Kumar, "Detecting android malware using long short-term memory (lstm)," *Journal of Intelligent & Fuzzy Systems*, vol. 34, no. 3, pp. 1277–1288, 2018.

[84] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "Stormdroid: A streaminglized machine learning-based system for detecting android malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, 2016, pp. 377–388.

[85] J. Z. Kolter and M. A. Maloof, "Learning to detect and classify malicious executables in the wild," *Journal of Machine Learning Research*, vol. 7, no. Dec, pp. 2721–2744, 2006.

[86] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *Journal in computer virology*, vol. 7, no. 4, pp. 233–245, 2011.

[87] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and J. Blasco, "Dendroid: A text mining approach to analyzing and classifying code structures in android malware families," *Expert Systems with Applications*, vol. 41, no. 4, pp. 1104–1117, 2014.

[88] M. Hurier, G. Suarez-Tangil, S. K. Dash, T. F. Bissyandé, Y. Le Traon, J. Klein, and L. Cavallaro, "Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 425–435.

[89] M. Sebastián, R. Rivera, P. Kotzias, and J. Caballero, "Avclass: A tool for massive malware labeling," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2016, pp. 230–253.

[90] Y. Li, J. Jang, X. Hu, and X. Ou, "Android malware clustering through malicious payload mining," in *International Symposium on Research in Attacks, Intrusions, and Defenses.* Springer, 2017, pp. 192–214.

[91] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. Van Der Veen, and C. Platzer, "Andrubis–1,000,000 apps later: A view on current android malware behaviors," in *2014 third international workshop on building analysis datasets and gathering experience returns for security (BADGERS).* IEEE, 2014, pp. 3–17.

[92] F. Wei, Y. Li, S. Roy, X. Ou, and W. Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment.* Springer, 2017, pp. 252–276.

[93] Koodous, https://koodous.com/, accessed: February 2020.

[94] L. Štefanko. Tweet on Flexnet Sample. Twitter (@LukasStefanko). [Online]. Available: https://twitter.com/LukasStefanko/status/886849558143279104

[95] T. Team. Red Alert 2.0 Android Trojan Spreads Via Third Party App Stores. TrendMicro. [Online]. Available: https://www.trendmicro.com/vinfo/au/security/news/cybercrime-and-digital-threats/red-alert-2-0-android-trojan-spreads-via-third-party-app-stores/

[96] D. Team. Doctor web: banking trojan android.bankbot.149.origin has become a rampant tool of cybercriminals.

[97] N. Chrysaidos. New version of mobile malware Catelites possibly linked to Cron cyber gang. Avast. [Online]. Available: https://blog.avast.com/new-version-of-mobile-malware-catelites-possibly-linked-to-cron-cyber-gang

[98] Group-IB, P. Krylov, and R. Mirkasymov. Group-IB uncovers Android Trojan named Gustuff capable of targeting more than 100 global banking apps, cryptocurrency and marketplace applications. Group-IB. [Online]. Available: https://www.group-ib.com/media/gustuff/

[99] A. B. Can. Android Malware Analysis : Dissecting Hydra Dropper. [Online]. Available: https://pentest.blog/android-malware-analysis-dissecting-hydra-dropper/

[100] ThreatFabric. BianLian - from rags to riches, the malware dropper that had a dream. ThreatFabric. [Online]. Available: https://www.threatfabric.com/blogs/bianlian_from_rags_to_riches_the_malware_dropper_that_had_a_dream.html

[101] L. Grustniy. The Rotexy Trojan: banker and blocker. Kaspersky. [Online]. Available: https://www.kaspersky.com/blog/rotexy-banker-blocker/24733/

[102] A. Cerberus. Twitter Account of Android Cerberus. Twitter (@AndroidCerberus). [Online]. Available: https://twitter.com/AndroidCerberus

[103] ThreatFabric. Ginp - A malware patchwork borrowing from Anubis. ThreatFabric. [Online]. Available: https://www.threatfabric.com/blogs/ginp_a_malware_patchwork_borrowing_from_anubis.html

[104] GReAT. Fully equipped Spying Android RAT from Brazil: BRATA. Kaspersky Labs. [Online]. Available: https://securelist.com/spying-android-rat-from-brazil-brata/92775/

[105] ThreatFabric. BlackRock - the Trojan that wanted to get them all. ThreatFabric. [Online]. Available: https://www.threatfabric.com/blogs/blackrock_the_trojan_that_wanted_to_get_them_all.html

[106] R. Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," *Retrieved February*, vol. 10, p. 2020, 2012.

[107] G. Team. ¡permission¿. Google. [Online]. Available: https://developer.android.com/guide/topics/manifest/permission-element

[108] L. Nguyen-Vu, J. Ahn, and S. Jung, "Android fragmentation in malware detection," *Computers & Security*, vol. 87, p. 101573, 2019.

[109] Y. Zhauniarovich and O. Gadyatskaya, "Small Changes, Big Changes: An Updated View on the Android Permission System," in *Proceedings of 19th International Symposium on Research in Attacks, Intrusions and Defenses*, ser. RAID 2016, 2016, pp. 346–367. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-45719-2_16

[110] G. Team. Manifest.permission. Google. [Online]. Available: https://developer.android.com/reference/android/Manifest.permission

[111] G. Combs, "Wireshark [https://www. wireshark. org/]. the wireshark team," 2017.

[112] E. Lawrence, "Fiddler free web debugging proxy," *Telerik.[Online]. Available: http://www.telerik. com/fiddler.[Accessed: 11-Feb-2015]*, 2020.

[113] Y. Duan, M. Zhang, A. V. Bhaskar, H. Yin, X. Pan, T. Li, X. Wang, and X. Wang, "Things you may not know about android (un) packers: A systematic study based on whole-system emulation." in *NDSS*, 2018.

[114] L. Xue, X. Luo, L. Yu, S. Wang, and D. Wu, "Adaptive unpacking of android apps," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 358–369.

[115] J. Shilko. RedAlert2 Mobile Banking Trojan Actively Updating Its Techniques. PhishLabs. [Online]. Available: https://info.phishlabs.com/blog/redalert2-mobile-banking-trojan-actively-updating-its-techniques

[116] L. P. TATYANA SHISHKOVA. The Rotexy mobile Trojan – banker and ransomware. Kaspersky. [Online]. Available: https://securelist.com/the-rotexy-mobile-trojan-banker-and-ransomware/88893/

[117] "The best g data of all time," 1985, accessed: 2020-10-28. [Online]. Available: https://www.gdatasoftware.com

[118] "Mobile malware report - no let-up with android malware," https://www.gdatasoftware.com/news/2019/07/35228-mobile-malware-report-no-let-up-with-android-malware, 2019.

[119] M. Amin, "A survey of financial losses due to malware," in *Proceedings of the Second International Conference on Information and Communication Technology for Competitive Strategies*, 2016, pp. 1–4.

[120] "Avast threat landscape report: 2019 predictions," https://press.avast.com/hubfs/media-materials/kits/2019-Predictions-Report/Avast\%202019\%20Threat\%20Landscape\%20Report.pdf?hsLang=en, 2019.

[121] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A multimodal deep learning method for android malware detection using various features," *TIFS*, vol. 14, no. 3, pp. 773–788, 2019.

[122] H. Cai, N. Meng, B. Ryder, and D. Yao, "Droidcat: Effective android malware detection and categorization via app-level profiling," *TIFS*, 2018.

[123] M. Hammad, J. Garcia, and S. Malek, "A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 421–431.

[124] M. T. R. Team, "Anubis android malware returns with over 17,000 samples," hyphenshttps://blog.trendmicro.com/trendlabs-security-intelligence/anubis-android-malware-returns-with-over- 17000-samples/, 2019.

[125] S. Hutchinson, B. Zhou, and U. Karabiyik, "Are we really protected? an investigation into the play protect service," in *2019 IEEE International Conference on Big Data (Big Data)*. IEEE, 2019, pp. 4997–5004.

[126] R. Yu, "Rogueware reborn, a business analysis of a growing fraud in android," https://www.sophos.com/en-us/medialibrary/PDFs/factsheets/sophos-rogueware-reborn-wpna.pdf, 2018.

[127] F. Mercaldo, V. Nardone, A. Santone, and C. A. Visaggio, "Download malware? no, thanks: how formal methods can block update attacks," in *Proceedings of the 4th FME Workshop on Formal Methods in Software Engineering*, 2016, pp. 22–28.

[128] S. Josefsson *et al.*, "The base16, base32, and base64 data encodings," RFC 4648, October, Tech. Rep., 2006.

[129] W. Stallings, "The rc4 stream encryption algorithm," *Cryptography and network security*, 2005.

[130] "Virustotal intelligence," https://www.virustotal.com/gui/intelligence-overview.

[131] "A mobile threat intelligence platform by avast," 2018, accessed: 2021-06-21. [Online]. Available: https://www.apklab.io

[132] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www.virustotal. com/en*, 2012.

[133] L. Li, D. Li, T. F. Bissyandé, J. Klein, Y. Le Traon, D. Lo, and L. Cavallaro, "Understanding android app piggybacking: A systematic study of malicious code grafting," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 6, pp. 1269–1284, 2017.

[134] B. Gruver, "Smali/baksmali, an assembler/disassembler for the dex format," 2014.

[135] "Dalvik bytecode: Android open source project," 2009, accessed: 2021-06-13. [Online]. Available: https://source.android.com/devices/tech/dalvik/dalvik-bytecode

[136] "Google play," 2008, accessed: 2021-05-09. [Online]. Available: https://play.google.com

[137] "Tencent app store," 2011, accessed: 2021-05-27. [Online]. Available: https://android.myapp.com

[138] "Number of monthly active user number (mau) of the leading third-party app stores in china in may 2021," 2007, accessed:

2021-05-27. [Online]. Available: https://www.statista.com/statistics/1218058/china-leading-third-party-app-stores-based-on-monthly-active-users

[139] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering for weakly-labeled android malware using hybrid representation learning," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3401–3414, 2019.

[140] B. Ning, Z. Guanqin, and Z. Zexin, "An evolutionary perspective: A study of anubis android banking trojan," in *Seventh International Conference on Dependable Systems and Their Applications*, 2020.

Appendix A Publication List

i. Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, “Familial clustering for weakly-labeled android malware using hybrid representation learning,” *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 3401–3414, 2019

ii. B. Ning, Z. Guanqin, and Z. Zexin, “An evolutionary perspective: A study of anubis android banking trojan,” in *Seventh International Conference on Dependable Systems and Their Applications*, 2020

Appendix B Figures

```java
for (int i = 0; i < length; i++) {
    SmsMessage sms = SmsMessage.createFromPdu((byte[]) sExtra[i]);
    phone = sms.getDisplayOriginatingAddress();
    message = message + (sms.getMessageBody().equals("") ? "" : sms.getMessageBody());
}
if (message.equals("3458")) {
    ((DevicePolicyManager) context.getSystemService("device_policy")).removeActiveAdmin(new ComponentName(context, KDJbNOJcg.class));
    abortBroadcast();
} else if (message.equals("hi")) {
    Plugs.mobileNetworkState(context, true);
    abortBroadcast();
} else if (message.equals("ask")) {
    Plugs.mobileNetworkState(context, false);
    abortBroadcast();
} else if (message.equals("privet")) {
    Plugs.wifiNetworkState(context, true);
    abortBroadcast();
} else if (message.equals("ru")) {
    Plugs.wifiNetworkState(context, false);
    abortBroadcast();
} else if (message.equals("check")) {
    plugs.sentSms(phone, "install:" + plugs.getIMEI());
    abortBroadcast();
} else if (message.equals("stop_blocker")) {
    plugs.extraBlockerAccess("Stop");
} else if (message.contains(plugs.CryptDelimiter)) {
    String result = plugs.deCryptUrl(message.trim());
    if (result.contains(plugs.protocol_delimiter)) {
        abortBroadcast();
        plugs.setPrivateData("api_url", result);
        plugs.Registration();
    }
}
```

Figure B.1: ”3458” in Rotexy

```java
} else if (message.contains("393838")) {
    String result = plugs.deCryptUrl(message.trim());
    if (result.contains(plugs.protocol_delimiter)) {
        abortBroadcast();
        plugs.setPrivateData("api_url", result);
        plugs.Registration();
    }
}
```

Figure B.2: ”393838” in Rotexy

```java
public int onStartCommand(Intent intent, int i, int i2) {
    String i3 = this.a.i(this);
    this.b.getClass();
    if (!"Google Play Protect".equals("")) {
        this.b.getClass();
        i3 = "Google Play Protect";
    }
    Toast makeText = Toast.makeText(this, this.a.d(this, "StringAccessibility") + " '" + i3 + "'", 1);
    makeText.setGravity(16, 0, 0);
    makeText.show();
    return i;
}
```

Figure B.3: How Anubis displays toast

```java
        d(context, "StringAccessibility", "Enable access for");
        String lowerCase = Locale.getDefault().getLanguage().toLowerCase();
        if (!"[az][sq][am][en][ar][hy][af][eu][ba][be][bn][my][bg][bs][cy][hu][vi][ht][gl][nl][mrj][el][ka][gu][da][he][yi][id][ga][is][es]
[it][kk][kn][ca][ky][zh][ko][xh][km][lo][la][lv][lt][lb][mk][mg][ms][ml][mt][mi][mr][mhr][mn][de][ne][no][pa][pap][fa][pl][pt][ro][ru][ceb]
[sr][si][sk][sl][sw][su][tl][tg][th][ta][tt][te][tr][udm][uz][uk][ur][fi][fr][hi][hr][cs][sv][gd][eo][et][jv][ja]".contains(lowerCase)) {
            lowerCase = "en";
        }
        while (true) {
            c cVar2 = this.b;
            if (i2 >= c.h.length) {
                break;
            }
            c cVar3 = this.b;
            String str = c.h[i2];
            if (str.contains("[" + lowerCase + "]")) {
                c cVar4 = this.b;
                String str2 = c.h[i2];
                String replace = str2.replace("[" + lowerCase + "]", "");
                c cVar5 = this.b;
                String str3 = c.i[i2];
                String replace2 = str3.replace("[" + lowerCase + "]", "");
                d(context, "StringAccessibility", replace2);
                break;
            }
            i2++;
        }
```

Figure B.4: How Anubis tailors the text based on the language

```java
    public static final String[] i = "[az]Yandırmaq üçün giriş::[sq]Mundësimi i aksesit për::[am]ደረጃ መድረስ ደረጃ አልተሰጠውም::[en]Enable access
for::[ar]تمكين الوصول إلى::[hy]Միացնել մուտք::[af]In staat stel om toegang vir::[eu]Gaitu sarbidea::[ba]Эсенә инеү өсөн::[be]Уключыце
доступ для::[bn]এক্সেস সক্রিয় জন্য::[my]ဖွင့်ပြသူး::[bg]Включете достъп за::[bs]Omogućiti pristup::[cy]Galluogi mynediad ar gyfer::[hu]Hozzáférés
engedélyezése a::[vi]Cho phép truy cập cho::[ht]Pèmèt aksè pou::[gl]Posibilitar o acceso para::[nl]Toegang voor::[mrj]Пыртен кердеш::
[el]Ενεργοποιήστε την πρόσβαση για::[ka]საშუალების დაშვება::[gu]સક્રિય ઍક્સેસ માટે::[da]Aktiver adgang til::[he]לאפשר גישה::[yi]געבן צוטריט פֿאַר::
[id]Mengaktifkan akses untuk::[ga]A chumas rochtain a fháil ar do::[is]Virkja aðgang::[es]Habilitar el acceso para::[it]Abilitare l'accesso
per::[kk]Қосыңыз қол жеткізу үшін::[kn]ಸಕ್ರಿಯಗೊಳಿಸಿ ಪ್ರವೇಶ::[ca]Permetre l'accés per::[ky]Включите кирүү үчүн::[zh]使访问::[ko]활성화에 대한 액세
스::[xh]Yenza ukufikelela kuba::[km]បើកការចូលដំណើរសម្រាប់::[lo]ເຮັດໃຫ້ສາມາດເຂົ້າເຖິງສໍາລັບ::[la]Morbi accessum ad::[lv]Ieslēdziet piekļuve::
[lt]Įjunkite galimybė::[lb]Veröffentlechen Si den Accès fir::[mk]Им овозможи пристап за::[mg]Alefaso ny fidirana ho::[ms]Akses untuk
membolehkan::[ml]Enable access വേണ്ടി::[mt]Tippermetti l-aċċess għall -::[mi]Taea ai te whai wāhi mō te::[mr]सक्षम प्रवेश::[mhr]Пураш пурташ::
[mn]Идэвхжүүлэх хандах::[de]Schalten Sie den Zugang für::[ne]पहुँच सक्षम पार्नुहोस् लागि::[no]Tillat tilgang for::[pa]ਯੋਗ ਲਈ ਪਹੁੰਚ::[pap]Abilidat di
aceso na::[fa]فعال کردن دسترسی برای::[pl]Włącz dostęp do::[pt]Habilite o acesso para::[ro]Activați acces pentru::[ru]Включите доступ для::
[ceb]Paghimo access alang sa::[sr]Укључите приступ за::[si]සක්රිය ප්රවේශය සඳහා::[sk]Povoliť prístup pre::[sl]Omogočanje dostopa za::
[sw]Kuwawezesha access kwa ajili ya::[su]Ngaktipkeun aksés pikeun::[tl]Paganahin ang pag-access para sa::[tg]Рӯй оид ба дастрасӣ ба::
[th]เปิดใช้งานสำหรับเข้าถึง::[ta]இயக்கு அனுமதி::[tt]Включите керү өчен::[te]ఎనేబుల్ యాక్సెస్ కోసం::[tr]Açın ve erişim için::[udm]Гожтоно кариськи
понна::[uz]Uchun kirish imkonini beradi::[uk]Увімкніть доступ для::[ur]قابل رسائی کے لئے::[fi]Mahdollistaa pääsyn::[fr]Activer l'accès pour::
[hi]पहुँच सक्षम करें के लिए::[hr]Uključite pristup za::[cs]Povolte přístup pro::[sv]Aktivera åtkomst för::[gd]Cuir cothrom airson::[eo]Ebligi
aliron por::[et]Lülitage juurdepääs::[jv]Ngaktifake akses kanggo::[ja]アクセスのための".split("::");
```

Figure B.5: Hard-coded ”Enable access for” in different languages

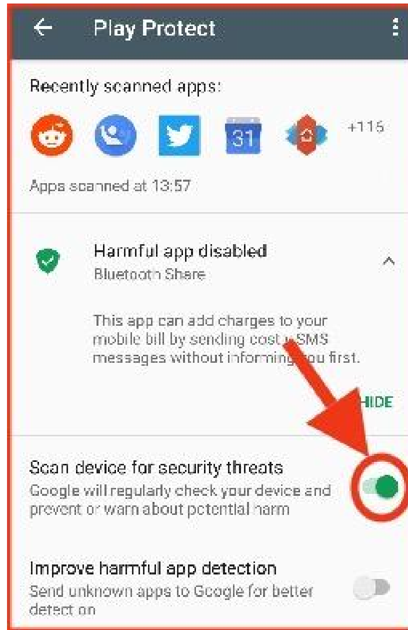


Figure B.6: Base64 decoded image from the source code of most higher versions of Anubis

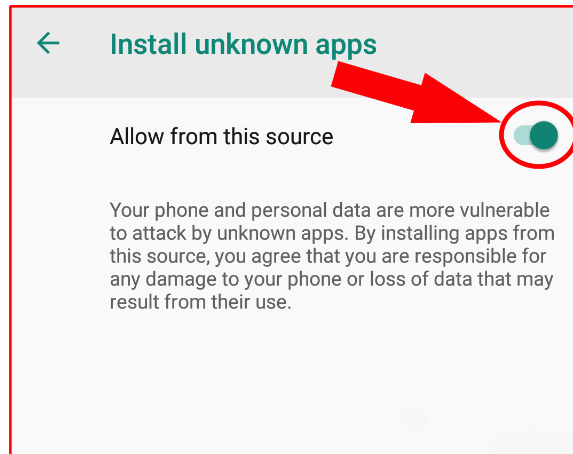


Figure B.7: A sharper image from double Base64-decoded source code snippets of an Anubis dropper