Deep Ground Truth Analysis of Current Android Malware

Fengguo Wei[1(✉)], Yuping Li[1], Sankardas Roy[2], Xinming Ou[1], and Wu Zhou[3]

[1] University of South Florida, Tampa, FL, USA
fwei@mail.usf.edu
[2] Bowling Green State University, Bowling Green, OH, USA
[3] Didi Labs, Mountain View, CA, USA

Abstract. To build effective malware analysis techniques and to evaluate new detection tools, up-to-date datasets reflecting the current Android malware landscape are essential. For such datasets to be maximally useful, they need to contain reliable and complete information on malware's behaviors and techniques used in the malicious activities. Such a dataset shall also provide a comprehensive coverage of a large number of types of malware. The Android Malware Genome created circa 2011 has been the only well-labeled and widely studied dataset the research community had easy access to (As of 12/21/2015 the Genome authors have stopped supporting the dataset sharing due to resource limitation). But not only is it outdated and no longer represents the current Android malware landscape, it also does not provide as detailed information on malware's behaviors as needed for research. Thus it is urgent to create a high-quality dataset for Android malware. While existing information sources such as VirusTotal are useful, to obtain the accurate and detailed information for malware behaviors, deep manual analysis is indispensable. In this work we present our approach to preparing a large Android malware dataset for the research community. We leverage existing anti-virus scan results and automation techniques in categorizing our large dataset (containing 24,650 malware app samples) into 135 varieties (based on malware behavioral semantics) which belong to 71 malware families. For each variety, we select three samples as representatives, for a total of 405 malware samples, to conduct in-depth manual analysis. Based on the manual analysis result we generate detailed descriptions of each malware variety's behaviors and include them in our dataset. We also report our observations on the current landscape of Android malware as depicted in the dataset. Furthermore, we present detailed documentation of the process used in creating the dataset, including the guidelines for the manual analysis. We make our Android malware dataset available to the research community.

1 Introduction

The Android platform continues to dominate the smartphone market with more than 80% share according to the study by International Data Corporation [8]

M. Polychronakis and M. Meier (Eds.): DIMVA 2017, LNCS 10327, pp. 252–276, 2017.
DOI: 10.1007/978-3-319-60876-1_12

and Gartner [29]. Over the last five years, the Android world has been changing dramatically with more features added, and more sensitive operations (*e.g.*, banking and wallet) becoming popular on smartphones. Along with the Android platform's popularity, the Android malware has been growing as well, with more complex logic and anti-analysis techniques.

As expected, research groups across academia and industry put enormous effort to design novel methods to detect Android malware. However, the above effort is adversely affected by the lack of clear understanding of the latest Android malware landscape. A reliable ground truth dataset is essential for building effective malware analysis techniques and verifying the validity of new detection methods. For understanding the nefarious techniques used in the state-of-the-art malware apps, **detailed behavior profiles for each malware variety must be provided in such a dataset**. While creating such a dataset is a must-do ground work, this task is extremely difficult. In particular, **to provide the rich information for malware behaviors, manual analysis is indispensable.** However it is not feasible to manually analyze all Android malware at our hands (we have 24,650 from various sources). Thus the first step is to categorize the samples into semantically equivalent groups; then we only need to study a few samples from each group.

One can use AV scanning service like VirusTotal [7] to group malware samples into families; however, the family labels returned are often inconsistent [16,25]. Moreover, we observe that malware samples within one family may actually contain different varieties with different behaviors. Thus we cannot simply rely upon the grouping provided by AV products, even after being refined by tools like AVclass [25]. Even if grouping has been done perfectly, the amount of work of manually analyzing representative apps from each malware variety is still daunting. Advanced obfuscation methods are widely adopted in recent Android malware apps, further complicating the manual analysis process.

Due to the above reasons, there has not been any effort on creating such a rich Android malware dataset, except for the Android Malware Genome [34] project. The Android Malware Genome dataset is no longer available to researchers due to resource limitations.

It provided a malware dataset containing 1260 malware samples categorized in 49 families, discovered in 2010 and 2011. We have collected a more recent Android malware dataset from several sources (VirusShare, Google Play and third party security companies). The malware in this collection were discovered between 2010 and 2016. We made comparative study of the Genome dataset with our malware samples of 2011 and later, and found that the majority of the threats in those newer samples are not captured by the Genome samples. As a result, we not only need a more up-to-date malware dataset for Android, we also need one with much richer semantic information than what the Genome dataset provided.

The main contributions of this work are as follows

1. We present a systematic method of analyzing large volumes of Android malware samples with high confidence, which helps us prepare a large ground

truth Android malware dataset with rich profile information. This method addresses the scalability challenge by leveraging a two-step grouping technique followed by a systematic and deep manual analysis.
2. We present a detailed guideline for performing the manual analysis so other researchers can replicate the process on other Android malware samples in their possession. Our manual analysis provides profiles for each variety of the Android malware regarding their behaviors. This provides insights into the landscape of the current Android malware.
3. We prepare a comprehensive dataset which contains 24,650 labeled Android malware samples that are classified in 135 varieties within 71 families, whose discovery dates range from 2010 to 2016. We publish detailed reports including behavior information for each malware variety at our Android malware website http://amd.arguslab.org/. We are sharing the whole dataset with the research community.

The rest of the paper is organized as follows. Section 2 discusses the process of preparing the dataset. Section 3 discusses in details the behaviors and techniques of malware in our dataset, and Sect. 4 discusses our analysis and observation of the malware evolution trends. We discuss related research in Sect. 5, and conclude in Sect. 6.

2 Methodology

We collect Android malware apps from multiple sources, analyze the samples, and report their detailed behaviors. Figure 1 illustrates the pipeline of the methodology, which consists of a two-step grouping process followed by a manual procedure: (a) Group malware samples with the same family name, (b) Categorize each family into semantically different varieties using a customized clustering analysis, (c) Conduct a systematic and deep manual analysis for each variety of malware samples to obtain the accurate and detailed behavior information for the malware.

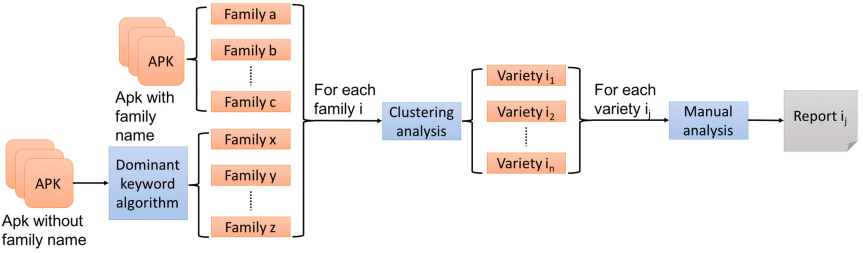


Fig. 1. Methodology pipeline: After malware families are identified, each family is categorized into semantically different varieties. For each variety we generate a malware behavior report, which is available at our Android malware website.

2.1 Identifying Malware Families

After raw malware samples are collected, it is an industry common practice to assign a family name to each app and group malware into families. The family name typically indicates the origin of the malware samples, such as in terms of the malware writer, malicious campaign, individual characteristics, *etc.*

We collect sample apps from multiple sources, including VirusShare, Google Play[1], and third party security companies. Most of the malware do not have an assigned malware family name. For such "unassigned" apps, the first step is to identify the family name.

Challenge. Existing state-of-the-art malware scanning service such as VirusTotal often provides multiple labels when it lists the scan result for an app using different anti-virus tools. However, due to inconsistent naming schemes from different anti-virus vendors [20,21], how to reliably identify a family name for a malware sample is a challenge.

Solution. We collected 1,464,590 unassigned app samples, and applied the following two steps:

(a) For each app x we get scan results of 55 antivirus products from VirusTotal (each result is either a candidate label or not-a-malware). If at least 50% of anti-virus products used in the VirusTotal recognize app x as a malware, we mark x as malware and move to the second step to obtain the family name. After this step, out of the collected apps, 1,216,885 are not labeled as malware by any AV product; about 195,185 are labeled as malware by some AV but did not reach the 50% threshold. We have 52,520 apps left.

(b) We obtain the family name of app x using a "dominant keyword algorithm" as follows. First, take the scanning results of app x from VirusTotal as label candidates. Second, normalize all the label candidates into individual English keywords, and meanwhile remove generic English keywords if any, *e.g.*, Trojan, Android, A, B, *etc.* There are a few hundred English keywords extracted and we identify the generic terms manually. Finally, we use the *dominant keyword* among the remaining labels as the family name. A keyword is dominant when: (a) the count of the keyword is greater than 50% of the anti-virus products used in the VirusTotal result; (b) the count of the most popular keyword is equal or more than twice of any other keyword, *i.e.*, there are no ambiguous labels that are highly popular at the same time. If for an app no dominant family name is found, we filter out the app from our dataset.

This process is very similar to *AVclass* [25], although we developed the approach independently without the knowledge of the AVclass work. An example is illustrated in Fig. 2, in which (1) We show VirusTotal result for an app (to save space we show only 10 anti-virus products' candidate labels for this app) (2) We

[1] Some malware can get pass Google's vetting system and end up in Google Play.

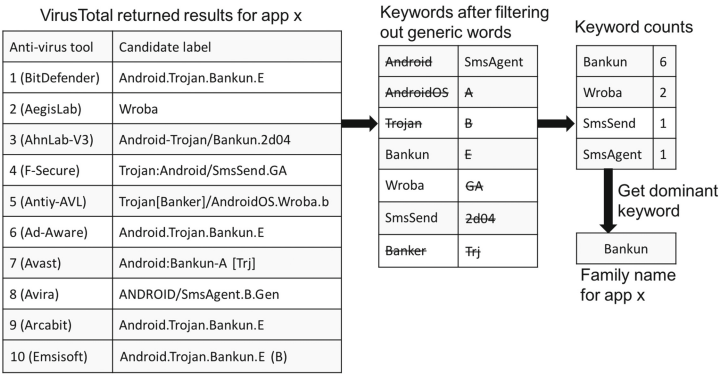


Fig. 2. Dominant keyword algorithm: identifying the malware family name of app x from VirusTotal scan results of app x. Not all AV tools are listed here to save space

extract the keywords from each of the result, and get a list of keywords such as *Android*, *AndroidOS*, *Bankum*, *Wroba*, *etc.* We filter out the generic keywords such as *Android*, *AndroidOS*, and *Trojan*. (3) We count the remaining keywords, and get *Bankun* as the dominant keyword, which is thus considered the family name. In particular, *Bankun* appeared 6 times, which is greater than 50% of the total results ($6 > 10 \times 50\%$), and more than twice of the count of the second dominant keyword *Wroba* ($6 > 2 \times 2$).

Out of the 52,520 apps obtained from step (1), we have 24,650 samples left after step (2). The rest are filtered out due to inconsistent family labels.

Discussion. Our goal is to provide a reliable ground truth dataset that presents insights into the up-to-date landscape of Android malware. The more anti-virus companies agree with the labeling for a malware sample, the more popular such family is and thus it is a more important representative to serve our purpose. We leave as future work to analyze those apps that as of now have no dominant family names.

2.2 Identifying Malware Behavior Groups – Varieties

It will not be feasible to perform deep manual analysis on each of the sample apps due to the large number of samples. How to reduce the amount of labor while maintaining the reliability of the result is a big challenge. While one may think that samples under the same family name should have similar behaviors, the reality is that the family name of a malware typically does not carry much semantic information. Anti-virus scanners name a malware with different and often inconsistent conventions [16]. Sometimes, a scanner names a malware after the malware writer Id; Other times the assigned family name is to highlight the main activities of the app (*e.g.*, ***FakePlayer***) or main goal of the app (*e.g.*, ***BankBot***), and so on. A malware app can achieve a goal through different

schemes. Thus the samples of a malware family can be very different in terms of their behaviors. Hence, we have to categorize the family members into semantically different groups which we call *varieties*. During our study, we observed that many families have more than one varieties.

This motivates us to apply a clustering analysis for a malware family to categorize the samples into different varieties. For a given family malware apps, we use a Android malware clustering analysis tool [18] to further categorize the labeled malicious apps into multiple varieties (Fig. 1). Each variety of apps reported by the clustering algorithm contains a unique version of malicious payload. Then, we only need to study a few representatives of each variety (not all apps therein) in the later manual analysis phase. This makes the whole manual analysis process scale. Details of the clustering algorithm can be found in our technical report [18].

2.3 Manual Analysis

We manually analyze each variety of malware samples. If a variety contains more than three samples, we randomly select three of them for manual analysis. Otherwise, we analyze all samples in the variety. Through a systematic study of the samples, we generate a detailed report on the malware variety's behavior.

Challenges

(a) Manually analyzing a malware sample warrants a systematic strategy; without a strategy it is nearly impossible to understand the comprehensive picture of a malware's behaviors.
(b) Anti-analysis/obfuscation techniques are commonly used in Android apps as well as in malware payloads, which has an adverse impact both on static analysis tools and to the analyst who wants to understand the semantics of the given app.
(c) The malware app itself may not always contain the full information. Many components could be fetched from a remote server while the malware runs on the infected device, and those servers may have already been taken down after the malware app was identified. Thus it may be impossible for us to obtain those missing parts for analysis.

Assistance Tools. When manually analyzing malware apps, we leverage available tools and frameworks wherever they are relevant and helpful. A static analysis tool with capability of collecting apk information and performing reachability analysis can help the analyzer quickly prioritize the analysis process. An appropriate tool can help obtain the trace to critical APIs. For instance, when analyzing renamed obfuscated apps, we cannot easily guess the semantics of the classes and methods. In that case, we should locate the critical API calls (*e.g.*, openConnection, sendTextMessage) and perform reachability analysis to understand from which component this API gets invoked, and track the call chain to get

a more clear picture of what the app is doing. To serve this purpose, we leverage Amandroid [28] which is a publicly available[2] comprehensive static analysis framework for analyzing Android apps.

In addition, an IDE-like editor that provides functionality of class hierarchy resolution, def-use chain building, method invocation tracing in the decompiled IR (intermediate representation) is also to the human analysts to understand the code flow. We built such an analysis tool for this purpose[3].

An Android app development environment is also important for manual analysis. An analyst may need to "re-implement" certain parts of the malware to test the real functionality, or to get the runtime value of certain variables. For instance, many malware apps encrypt the string constant and the malicious payloads to avoid detection. When analyzing such a malware, we first identify the decryption routine, extract and load it in a separate app, and then provide the encrypted content to get the plaintext information.

The Overall Strategy of Manual Analysis. With the help from the aforementioned assistance tools, we performed manual analysis of 405 Android malware samples representing 135 varieties. Here we present a systematic way of how to manually analyze Android malware, which serves as a guideline for other people who want to reproduce our analysis results, or to analyze other Android malware apps.

Identifying Malicious Components. An Android app is organized as a collection of components. To understand the behavior of a given malware sample we have to identify which components belong to the malware payload, or whether the whole app is a standalone malware. As the clustering analysis (CA) tool [18] we use is imperfect, the payload it outputs for each variety could be the full payload or a partial payload. For the latter case, we need more effort to identify the full payload. We get help from the following observations: (1) Since a component is the basic functional block for an Android app, we can expect that the full component is likely to belong to the payload, if a few of the component's methods or reachable methods appear in the CA-extracted payload; (2) In most cases, the package name is a good indicator; if some of a package's classes appear in the CA-extracted malicious payload, then the whole package is very likely to belong to the payload. Malware writers could also instrument the benign part of the repackaged malware to initialize the payload, so we should also search for any use of payload package names inside the benign components. This will enrich our understanding of the activation strategy for this malware.

Prioritizing Component Analysis. We should not start the analysis from a random component as that will not put the analysis in a meaningful context. After obtaining the malicious components using the CA tool, we follow a triaging scheme, and analyze the components in the following sequence:

[2] Tool website: http://pag.arguslab.org/argus-saf.
[3] Tool website: http://pag.arguslab.org/argus-cit.

(a) Event handlers: Event handlers mostly serve as the entry points in an Android app. More specifically, the main Activity and the BroadcastReceiver receiving "android.intent.action.BOOT_COMPLETED" event (BootReceiver) is the initializer, which can be used to start the core component (*e.g.*, monitor service) of the malware, so it should be analyzed first. Other event handlers are mainly related to monitoring user information and the environment of infected device. They also can be considered as entry points of certain malware. Take as example a BroadcastReceiver which receives "android.provider.Telephony.SMS_RECEIVED". This component is used to listen to any new incoming SMS message for this device. When we analyze such a component, we should check how it handles the message, whether it performs some operation related to the device inbox, if it matches the incoming message phone number with some list (*e.g.*, bank phone numbers, vendor phone numbers, *etc.*), and aborts the SMS using abortBroadcast() method call.

(b) The services that are started by initializers normally contain the main logic of the malware (monitor service); thus they need to be analyzed as soon as possible. It is the core component for most malware, which the malware will try to keep running as long as possible. It is common to see that many entry point components or scheduled tasks will start such service. The monitor service normally is used to fetch and reply to commands from a command and control server. It is also common to schedule some TimerTask or BroadcastReceiver to constantly check the internet connectivity, whether an anti-virus product is running, whether itself is still alive, and so on.

(c) All remaining components. The purpose of those components vary. The guideline is to start from such a component and trace all the reachable code to understand: (i) what role the component plays, (ii) which other components this component communicates with, (iii) which BroadcastReceiver this component registers, (iv) whether this component starts some thread or AsyncTask and what is the purpose.

Building the Behavior Report. After we analyze all the relevant components, we generate a report that includes an inter-component graph where a node represents a component (present in the malicious payload) or a worker thread loaded by such a component, and an edge represents the communication/interaction between two nodes. The graph also illustrates behavior description for each node and edge, such as the activation method, communication message, C&C commands, *etc.* This gives us a comprehensive picture of the malware on top of which we can understand its richer behavior, *e.g.*, what is the monetizing method, how it maintains the persistence, its main goal, and so on. The behavior report including the inter-component graph for each malware variety is available at the Android malware website.

Handling Anti-analysis/Obfuscation

(a) Renaming: Class name, method name and field name are important hints for understanding the malware's purpose. Renaming them to meaningless words makes manual analysis difficult. We can get help from static analysis tools to perform a reachability analysis to see all the reachable methods from a given component. This can help us locate the interesting APIs (as the system API names cannot be renamed). We follow the call trace to understand how an API gets invoked and how the calling parameters are prepared.
(b) String encryption: Oftentimes, we understand the malware behavior based on the strings used in the code, like URL, C&C command, class names, phone number, *etc.* If those strings are encrypted, it is very difficult to understand the semantics of those actions. To address this issue, we analyze the malware code to figure out the decryption routine and key. We then re-implement it in a separate app to decrypt the strings.
(c) Dynamic loading: Malware may hide its functionality in a separate apk/dex file and load it dynamically at runtime. Even worse, apk/dex file may be encrypted. To handle such cases, we first retrieve the decryption routine to decrypt the apk/dex file. For either case we decompile the code to study it as a regular app, which adds to our understanding of the malware.
(d) Native payload: Most Android static analysis tools do not handle native code. Thus malware writers like to put some core function or data in the native payload. For us to understand how the native payload works, we use standard binary reverse-engineering tools including IDA [5] and hexdump [4].

Handling Missing Contents. Sometimes, we may not be able to obtain the full payload of the malware, but we still have ways to maximize our understanding. The basic idea is to understand how the malware leverages the missing content. For instance, if we observe that the malware downloads an apk file, we could see whether this malware sends an installation request for this apk or it uses *DexClassLoader* to load some new classes. In the first case, we could check the description of the installation request (which will show up on the screen to the device user) to understand the purpose of such action. For example, this description may say "Crucial update found for xxx." Then we know it is misleading the user to install a malware. In the second case, we know this malware is dynamically loading some code; we should expect to see multiple java reflection calls to such code, and from those reflection calls we could infer what role it plays.

Discussion. One may wonder what is the benefit of our study given the fact that after a malware family is discovered, anti-virus companies usually publish a report/bulletin on a sample app from that family. In fact, for each family under our study (71 in total), we did find such reports on the web. However, such reports usually only highlight the security breaches and main activities of the

malware family and do not describe the malware behaviors in details. This is not sufficient for malware research. In addition, those reports do not provide the varieties for each family and the different malware behaviors from those varieties.

3 Android Malware Profiling

We present an overview of our Android malware dataset, and discuss the detailed profiles for the samples along two main dimensions: behaviors and monetization methods. The detailed information of each malware variety can be found at our Android malware website.

3.1 Malware Dataset Overview

Table 1 provides an overview of the malware families in our dataset. For each family we show the time it was first discovered. The malware type roughly indicates the main purpose of the family. The table shows the number of samples, and the number of varieties in each family. The dataset consists of 24,650 malware samples categorized in 135 varieties within 71 families.

Table 1. Dataset overview.

Family	Type	Samples	Variety	Detection
Lnk	Trojan	5	1	07/2010
FakePlayer	Trojan-SMS	21	2	08/2010
DroidKungFu	Backdoor	546	6	05/2011
GoldDream	Backdoor	53	2	07/2011
GingerMaster	Backdoor	128	7	08/2011
Boxer	Trojan-SMS	44	1	09/2011
Zitmo	Trojan-Banker	24	2	10/2011
SpyBubble	Trojan-SMS	10	1	11/2011
Fjcon	Backdoor	16	1	11/2011
Steek	Trojan-Clicker	12	1	01/2012
FakeTimer	Trojan	12	2	01/2012
Opfake	Trojan-SMS	10	2	01/2012
FakeAngry	Backdoor	10	2	02/2012
FakeInst	Trojan-SMS	2172	5	05/2012
FakeDoc	Trojan	21	1	05/2012
MobileTX	Trojan	17	1	05/2012
Nandrobox	Trojan	76	2	07/2012
Mmarketpay	Trojan	14	1	07/2012
UpdtKiller	Trojan	24	1	07/2012
Vidro	Trojan-SMS	23	1	08/2012
SmsZombie	Trojan-Spy	9	1	08/2012
Lotoor	HackerTool	333	15	09/2012
Penetho	HackerTool	18	1	10/2012
Ksapp	Trojan	36	1	01/2013
Winge	Trojan-Clicker	19	1	01/2013
Mtk	Trojan	67	3	02/2013
Kyview	Adware	175	1	04/2013
SmsKey	Trojan-SMS	165	2	04/2013
Obad	Backdoor	9	1	06/2013
Vmvol	Trojan-Spy	13	1	06/2013
AndroRAT	Backdoor	46	1	07/2013
Stealer	Trojan-SMS	25	1	07/2013
Boqx	Trojan-Dropper	215	2	07/2013
Bankun	Trojan-Banker	70	4	07/2013
Mseg	Trojan	235	1	08/2013
FakeUpdates	Trojan	5	1	08/2013
Minimob	Adware	203	1	09/2013
Tesbo	Trojan-SMS	5	1	09/2013
Gumen	Trojan-SMS	145	1	10/2013
Svpeng	Trojan-Banker	13	1	11/2013
Spambot	Backdoor	15	1	12/2013
Utchi	Adware	12	1	02/2014
Airpush	Adware	7843	1	03/2014
FakeAV	Trojan	5	1	04/2014
Koler	Ransom	69	2	05/2014
SimpleLocker	Ransom	173	4	06/2014
Cova	Trojan-SMS	17	2	06/2014
Jisut	Ransom	560	1	06/2014
Univert	Backdoor	10	1	07/2014
Aples	Ransom	21	1	07/2014
Finspy	Trojan-Spy	9	1	08/2014
Erop	Trojan-SMS	46	1	08/2014
Andup	Adware	45	1	11/2014
Ramnit	Trojan-Dropper	8	1	11/2014
Kuguo	Adware	1199	1	02/2015
Youmi	Adware	1301	1	02/2015
Dowgin	Adware	3385	1	02/2015
Fobus	Backdoor	4	1	03/2015
BankBot	Trojan-Banker	740	8	03/2015
Roop	Ransom	48	1	05/2015
Ogel	Trojan-SMS	6	1	06/2015
Mecor	Trojan-Spy	1820	1	07/2015
Ztorg	Trojan-Dropper	20	1	08/2015
Gorpo	Trojan-Dropper	37	1	08/2015
Leech	Trojan-SMS	128	3	09/2015
Fusob	Ransom	1277	2	10/2015
Kemoge	Trojan-Dropper	15	1	10/2015
SlemBunk	Trojan-Banker	174	4	12/2015
Triada	Backdoor	210	1	03/2016
RuMMS	Trojan-SMS	402	4	04/2016
VikingHorde	Trojan-Dropper	7	1	05/2016
Total: 71		24650	135	

3.2 Malware Behaviors

Table 2 illustrates the behaviors of malware families[4] we analyzed. Due to space constraint we only present part of the analysis result. The more detailed information can be found at our Android malware website. Behavior tags in one category may not be mutually-exclusive — some apps may present multiple behaviors in a category.

Composition. There are three ways an Android malware is composed: a **standalone** app where the malware was written from scratch, a **repackaged** app where the malware was repackaged within a legitimate app, and **library** where the malicious components exist in the library code of an otherwise legitimate app. For the library case, this is the common way adware gets on the user's device. The difference between this method and repackaging is that the malicious payload here may get tagged on the app by the app developer (who may not be aware of the malicious activity inside the library) as opposed to being repackaged by a malware writer.

In our dataset, we observe that 63% of malware varieties and 35% of malware apps (shorthanded 63%/35% thereafter) are standalone, 30%/7% of malware are repackaged, and in 7%/58% of malware the malicious payload is installed as a library of the "legitimate" app. This means repackaging is no longer the dominant method for composing Android malware. The reason could be that malware writers nowadays put more effort in Android, and have started to design more comprehensive and sophisticated malware from scratch. For instance, ***FakeAV*** is a fake anti-virus family; its behavior looks exactly the same as a typical anti-virus application and its appearance looks very professional. ***Bankun*** masquerades as the legitimate Korean bank app – in fact it looks exactly the same as the legitimate one.

Even in decline repackaging is still frequently used in distributing malware. We define two types of repackaging: *isolated* repackaging and *integrated* repackaging.

Isolated repackaging means the malware payload is packaged into a legitimate application x but not in any way connected with x's original functionality. It declares its own event handler as the activation component, and does all the malicious tasks on its own without affecting x's functionality.

Integrated repackaging is the more advanced way where the malware author modifies the workflow of (or injects code into) the host app, and lets the payload run together with the host app. This makes the malware more stealthy, and more likely to be activated. For instance, ***VikingHorde*** [22] replaces the app's launcher Activity with its own launcher; the launcher will activate its monitoring service and then start the host app's launcher component.

[4] Table 2 aggregates the behaviors over all malware varieties in a family. The more specific per-variety breakdown can be found at our Android malware website.

Table 2. Malware behaviors.

Legend					
Composition	Standalone (ST)	Repackaging (RPKG): Isolated (O), Integrated (T)	Library (LIB)	**Installation**	Drop (DR)	Drive-by Download (DD)
Activation	Event (EV)	By Host App (BHA)	Scheduling (SC)	**Info Stealing**	Device Info (DI)	Personal Info (PI)
Persistence	Stealthy (TH):	Block (BL), Clean (CL), Hide Icon (HI), Rootkit (RK)				
	Prevent destroy (PD): Hide Admin (HA), Kill AV (KA), Lock Device (LD), Monitor Destroy Action (MDA), Reinstall (RI), System App (SYS)					
Privilege	Request Admin (RA)	Root Exploit (RE)				
C&C	Internet (IN)	Command Encoding (CE): JSON (J), Java Script (JS), XML (X), Custom Protocol (P)				
Anti-analysis	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)					

Family	Composition			Installation		Activation			Info Stealing		Persistence		Privilege		C&C			Anti-analysis				
	ST	RPKG	LIB	DR	DD	EV	BHA	SC	DI	PI	TH	PD	RA	RE	IN	SMS	CE	RN	SE	DL	NP	EDA
Airpush			✓				✓	✓	✓	✓					✓		J	✓				
AndroRAT	✓			✓		✓		✓	✓	✓					✓		P					
Andup			✓	✓		✓	✓		✓									✓	✓			
Aples	✓					✓		✓	✓			LD	✓		✓		P					
BankBot	✓			✓	✓	✓		✓	✓	✓	BL&HI	MDA	✓		✓	✓	J&P	✓	✓	✓		CDI
Bankun	✓			✓	✓	✓		✓	✓	✓	BL&HI		✓		✓		J&X					CDI
Boqx		O		✓		✓															✓	
Boxer	✓				✓	✓												✓	✓			
Cova	✓			✓		✓			✓	✓	BL				✓		JS	✓				
Dowgin			✓	✓		✓	✓		✓						✓		J	✓	✓	✓		EC
DroidKungFu	✓	O&T		✓		✓	✓	✓	✓	✓		KA		✓	✓		J&P	✓	✓		✓	
Erop	✓					✓			✓													
FakeAV	✓					✓		✓			BL											
FakeAngry		O		✓		✓			✓						✓		P	✓	✓			
FakeDoc	✓					✓		✓	✓		BL							✓				
FakeInst	✓			✓		✓			✓		BL				✓		J&P	✓	✓			
FakePlayer	✓					✓					BL							✓				
FakeTimer	✓					✓		✓	✓	✓												
FakeUpdates		T		✓		✓	✓	✓							✓		X	✓	✓			
Finspy	✓					✓			✓	✓	HI					✓	P	✓				EC
Fjcon		O		✓		✓			✓		BL				✓		X					
Fobus	✓			✓		✓			✓		BL&HI		✓		✓	✓	X		✓	✓		EC
Fusob	✓					✓		✓	✓			LD&MDA	✓		✓		J	✓	✓	✓		
GingerMaster		O&T		✓		✓	✓	✓	✓	✓				✓	✓		P	✓	✓			
GoldDream	✓	T		✓		✓			✓	✓					✓		P					
Gorpo		O		✓					✓					✓	✓		J	✓	✓			EC
Gumen		T				✓		✓	✓	✓	BL				✓		X				✓	DI
Jisut	✓					✓						LD										
Kemoge		O		✓		✓			✓					✓	✓		P					
Koler	✓					✓			✓	✓		LD&MDA	✓		✓		JS&P	✓				CDI
Ksapp		T		✓		✓	✓	✓	✓	✓					✓		P					EC
Kuguo			✓	✓		✓	✓		✓	✓					✓		P	✓				
Kyview				✓	✓		✓		✓	✓					✓		J	✓	✓			
Leech		T		✓		✓	✓	✓	✓	✓	BL	MDA		✓	✓		J	✓	✓	✓		EC
Lnk		T					✓							✓								
Lotoor	✓			✓		✓								✓				✓		✓	✓	
Mecor	✓					✓			✓	✓					✓		JS					
Minimob					✓	✓	✓	✓	✓	✓					✓		J	✓				
Mmarketpay		O		✓		✓		✓	✓	✓	BL				✓		P					
MobileTX	✓					✓			✓	✓												
Mseg		O	✓			✓		✓	✓	✓	BL				✓		P	✓				
Mtk		O&T		✓		✓	✓	✓	✓						✓		P	✓	✓	✓		
Nandrobox		T		✓	✓	✓	✓		✓		BL				✓		J					
Obad	✓			✓	✓	✓		✓	✓	✓	BL&HI	HA	✓		✓	✓	J	✓	✓			CDI&EC
Ogel	✓					✓		✓	✓	✓	BL				✓		P	✓			✓	
Opfake	✓				✓	✓		✓	✓	✓	BL&CL&HI				✓		P		✓			
Penetho	✓					✓																
Ramnit		T					✓															
Roop	✓					✓			✓		HI	LD	✓		✓		JS	✓				
RuMMS	✓			✓	✓	✓		✓	✓	✓	BL&HI		✓		✓		J	✓	✓	✓		
SimpleLocker	✓			✓		✓		✓	✓		HI	LD	✓		✓	✓	J&P	✓				
SlemBunk	✓			✓	✓	✓		✓	✓	✓	BL&HI		✓		✓	✓	J		✓	✓	✓	
SmsKey		T				✓	✓											✓				
SmsZombie	✓			✓		✓		✓	✓	✓	BL&CL		✓			✓	X					
Spambot	✓					✓			✓	✓	BL		✓									
SpyBubble	✓			✓		✓		✓	✓	✓	BL&CL				✓		X					
Stealer	✓			✓		✓		✓	✓	✓	BL	MDA			✓		JS	✓				
Steek	✓					✓																
Svpeng	✓			✓	✓	✓			✓	✓	BL	LD			✓	✓	P					CDI
Tesbo		O				✓			✓		BL&CL				✓		X	✓	✓			
Triada	✓			✓		✓		✓	✓		CL&RK	SYS			✓		P	✓	✓	✓		CDI&CIA
Univert	✓					✓		✓	✓	✓	BL				✓		J					
UpdtKiller		T				✓	✓	✓	✓		BL	KA&MDA	✓		✓		X	✓			✓	
Utchi				✓	✓	✓	✓		✓	✓								✓				
Vidro	✓					✓		✓	✓	✓	BL				✓		J					
VikingHorde		T		✓		✓		✓	✓	✓		RI	✓		✓		J				✓	
Vmvol	✓			✓		✓			✓	✓	BL&CL				✓		J					
Winge		O		✓		✓			✓	✓					✓		X	✓				
Youmi			✓	✓			✓		✓	✓					✓		P	✓				
Zitmo	✓					✓		✓	✓	✓	BL&HI					✓	P	✓				
Ztorg		T		✓			✓		✓					✓	✓		J	✓	✓	✓		EC
Total families:	41	24	9	40	9	64	20	34	39	58	34	15	15	8	50	9	53	39	22	11	8	14
Total varieties:	85	40	9	76	15	120	23	58	92	61	53	27	30	32	83	12	86	64	35	13	19	15
Total apps:	8567	1833	14231	9231	1218	14980	14687	12341	21333	15035	2839	3549	2823	1061	22108	367	22145	18143	7211	5072	972	4051
Malware	ST	RPKG	LIB	DR	DD	EV	BHA	SC	DI	PI	TH	PD	RA	RE	IN	SMS	CE	RN	SE	DL	NP	EDA
	Composition			Installation		Activation			Info Stealing		Persistence		Privilege		C&C			Anti-analysis				

Installation. Besides being installed by users, there are a couple other ways Android malware get on a victim's device.

Drop: There are more than 56%/37% of malware that try to download and install applications on the victim's device; the downloaded application could be the malware's real payload, upgraded version, or other risky applications. There are different ways malware use to install applications on victim's device. ***Vmvol***

will show a dialog with critical update message to trick the victim user to install the payload as an update.

Drive-by Download: We adopt the following definition for Drive-by download [3]: (1) Downloads which a person authorized but without understanding the consequences; and (2) Any download that happens without a person's knowledge. As one example, ***SlemBunk*** [31,33] gets on the device when the user visits some porn website. The website will show a prompt that asks the user to upgrade the Flash Player; if the user chooses to upgrade, it will actually download the Slembunk malware. Another example: ***Bankun*** will collect victim's contacts, and send a message saying "*We will send you a mobile birthday invitations* http://vik6.pw" to each of the contacts. As people normally trust what they get from their friends, the friend will likely click on the link, and the malware will be downloaded.

Activation. Android Malware Genome only reported event-based activation methods. Our analysis found two more options: **by-host-app** and **scheduling**.

The by-host-app option is closely related to the integrated repackaging method, where the attacker instruments code into the host app to activate the malware together with the host app. This is the typical way for activating adware, which we will discuss in Sect. 3.3.

The scheduling option is also frequently used to start their monitoring or data collection in a periodic manner. Typically, the malware registers a Timer task thread, or uses Android's AlarmManager with PendingIntent. When certain time goes by, the malware's monitor service is activated to get new commands from the C&C server. One extreme use of scheduling is in ransomware. Some ransomware apps schedule a periodic task using a very short interval, making the victim device non-responding. We discuss this more in Sect. 3.3.

Information Stealing. In our dataset, more than 68%/87% malware collect users' device information, such as international mobile station equipment identity (IMEI), international mobile subscriber identity (IMSI), kernel version, phone manufacturer, network operator, *etc.* We observe that information items such as IMEI and IMSI are unique for each device and thus could be used as an identifier to register the compromised device with the C&C server. Other device information items, such as the OS version, the baseband version, the OS language, and installed applications give the C&C server some idea of the target device's specification, based on which the C&C server can decide the strategy for using the compromised device.

Persistence. In our dataset, 48%/22% malware use at least one persistence technique, which shows that persistence is one of the important attributes the malware writers consider in the app design. The longer the malware can stay in the victim's device, the more revenue they can produce for the adversary. Persistence can be achieved over multiple dimensions, including:

(a) Making malware's presence stealthy. We observed multiple stealthy methods malware use to hide evidence of malicious activity: (a) Blocking the appearance of items such as audio, call, notification, or SMS, (b) Cleaning items such as call log and SMS history – important for the malware since the automatically added messages or phone records may alert the victim user that something wrong may have happened, (c) Hiding the malware's launcher icon despite the malware's background service running, (d) Hooking system APIs to mask its existence.
(b) Preventing itself from being destroyed by the system, anti-virus product, or the user via techniques such as hiding itself from appearing in the device administrator list, killing AV process, locking device, *etc.*

Privilege Escalation. Obtaining admin privilege can make the malware much harder to remove, and can allow the malware to perform privileged operations such as changing lock-screen pin code, locking device, wiping device data, *etc.* More and more malware these days try to acquire admin-privilege. ***Obad*** leverages admin-privilege to make it disappear. Another notable malware family is ***Fobus***. Once ***Fobus*** gets admin-privilege, it will listen to the *DEVICE_ADMIN_DISABLE_REQUESTED* event. If the user tries to disable admin-privilege for this malware, it will lock the screen before the user can click the confirm button. Even if the user is fast enough to click the confirm button, it will display a message saying that if the user continues, the malware will do a factory reset of the device resulting in all the user's data being lost. Users usually know that granting admin-privilege is risky. Nevertheless, malware apps always try to convince the victim that they are security related services (*e.g.*, ***Updtkiller***), or they can make the device more efficient (*e.g.*, ***Fobus***). If the victim does not grant admin-privilege, many malware apps (*e.g.*, ***SmsZombie***) will aggressively ask for it, which annoys the victim and makes the device unusable.

Lotoor is a generic name for a collection of hacking tools that exploit vulnerabilities to root a device and perform privileged actions by leveraging the root privilege. Our dataset has 15 varieties of different hacking tools under the name ***Lotoor***. Those tools either help user root their device, or perform actions needing root privilege. Most rooter malware contain one to three root exploits targeting 2.x Android devices. The mostly used root exploits are *Exploid*, *RageAgainstTheCage (RATC)*, and *GingerBreak*. However, ***Lotoor.FramaRoot*** changes the story – it is the most comprehensive hacking tool containing at least eleven exploits that target devices with all kinds of processors (*e.g.*, *Exynos*, *Qualcomm*, *Mediatek*, *etc.*) ranging from 2.x to 4.x. ***Lotoor.MasterKey*** does not use any root exploit, but leverages a MasterKey vulnerability to hide its payload in a system app and bypasses the Android cryptographic verifier to infect the victim's device up to 4.x.

At the end of 2014 malware with root exploits appeared again while they still targeted devices before 4.x. ***Leech***, ***Ztorg***, ***Gorpo*** work together [13] and form a kind of "malvertising botnet." They leverage root privilege to drop new malware on the "network" of infected devices. For instance, ***Triada*** is dropped

by this network. ***Triada*** has some interesting behaviors. It is a modular malware (with well-defined interfaces) with active use of root privilege. Once it is installed on the rooted device, it will try to exchange a configuration file with the C&C server, which contains the communication rate, the modules that need to be downloaded, *etc.* The modules include downloader, SMS trojan, and banking trojan. ***Triada*** is as sophisticated as traditional PC malware, which raises the alert that Android malware are evolving from the more primitive form to the next level.

Kaspersky reports [14] that Android devices running versions higher than 4.4.4 have much fewer exploitable vulnerabilities. This may explain why malware with root exploits are becoming less popular than reported in Android Malware Genome. However, there are still about 60% devices running old versions of Android that are vulnerable to rooting attack. Thus root exploit is still a major threat to Android devices.

Command and Control (C&C). 64%/90% have C&C servers. C&C increases the functionality and flexibility of the malware, helps it adapt to its running environment, continuously monitors the victim, and makes the best strategy to generate revenue. A C&C module generally contains a message builder and a command handler. Android malware have a variety of ways to transmit the collected items to the server. For example, ***SmsZombie*** builds a formatted text message and sends it to the server via SMS; one version of ***FakeAngry*** builds a URL like *http://l.anzhuo7.com:8097/getxml.do?flagid=-500&mediaver=7&channel=202_109&imei=xxx&...* which contains information items as the parameters; a newer version of ***FakeAngry*** puts the data into the HTTP POST request entity; ***SpyBubble*** stores all the messages into an XML file; ***RuMMS*** [32] encodes data into JSON format.

Upon receiving a command from the C&C server, the malware will perform certain actions according to the command. We observe that there are at least the following ways by which a command handler is designed.

(a) Commands can be in a standard formatted such as XML or JSON. The decoding routine reads contents from the command and perform the tasks accordingly.
(b) Android *Webview* allows the developer to specify a Javascript to Java bridge interface [1]; when the server sends javascript code back to the *Webview*, it will automatically be mapped to the corresponding Java method to perform the task.
(c) Many malware varieties use plain text or a self-defined protocol for the command format. A custom protocol is not necessarily less sophisticated than the other types. One of the most notable is ***Ksapp***, which uses a self-designed language MDK as the command. In the malware payload, it contains a full interpreter of MDK including a lexer, a parser, and an MDK to Java type mapper. Whenever the malware receives a new command file, it will first parse it, generate a function table, and start executing from a predefined entry point function "start." In the execution, the MDK will

map MDK types to Java types, and for the invocation, MDK will issue the invocation in JVM via reflection. When analyzing this kind of malware, analysts cannot see any functionality in the malware payload, but the malware can perform whatever actions allowed by permissions specified in the AndroidManifest.

Anti-analysis Techniques. We observe that 63%/79% of malware use at least one anti-analysis technique.

Renaming is one of the most adopted obfuscation techniques. It translates the original meaningful package, class, method, field, and parameter names into some meaningless or unreadable form. This makes the manual analysis much harder. However, it does not impact static analysis tools, and API calls cannot be renamed.

String Encryption is also widely found in malware. Strings in the code like server URL, JSON/XML key values, intent action, component name strings, or reflection strings can help anti-virus product or analysts identify the malware. Malware can use string encryption to change the constant strings to ciphertext, which increases the difficulty of understanding the malware behavior. Normally, malware uses the following ways or their combination to encrypt the string: byte permutation, one-time pad, base64 encoding, DES/AES, *etc.* To manually inspect those malware, we had to reimplement the decryption/decoding routine and map the ciphertext back to the plaintext form to understand their behavior.

```
..ComponentName v0_3 = this.getComponentName();¶
..Object[] v2_2 = new Object[3];¶
..v2_2[2] = 1;¶
..v2_2[1] = 2;¶
..v2_2[0] = v0_3;¶
..Context v1_1 = this.getApplicationContext();¶
..Object v1_2 = Class.forName(IljIllj.IliijIII(IljIllj.IliijIII[104],¶
....IljIllj.IliijIII[33] + 1, IljIllj.IliijIII[26]))¶
....getMethod(IljIllj.IliijIII(IljIllj.IliijIII[20],¶
....IljIllj.IliijIII[20] | 14, 183), null)¶
....invoke(v1_1, null);¶
..Class<?> v0_2 = Class.forName(IljIllj.IliijIII(IljIllj.IliijIII[107],¶
....IljIllj.IliijIII[107] | 24, 79));¶
..int v3 = IljIllj.IliijIII[138];¶
..int v4 = IljIllj.IliijIII[127] - 1;¶
..v0_2.getMethod(IljIllj.IliijIII(v3, v4, v4 | 69),¶
....Class.forName(IljIllj.IliijIII(IljIllj.IliijIII[7],¶
....IljIllj.IliijIII[33] + 1, 136)), Integer.TYPE, Integer.TYPE)¶
....invoke(v1_2, v2_2);¶
```

```
..ComponentName v0_3 = this.getComponentName();¶
..Context v1_1 = this.getApplicationContext();¶
..PackageManager v1_2 = v1_1.getPackageManager();¶
..v1_2.setComponentEnabledSetting(v0_3,¶
....PackageManager.COMPONENT_ENABLED_STATE_DISABLED,¶
....PackageManager.DONT_KILL_APP);¶
```

Fig. 3. Obad Code Snippet. The obfuscated code is on the top; the de-obfuscated version is at the bottom.

One notable family that extensively adopts renaming and string encryption techniques is ***Obad***. Figure 3 shows the code snippets in ***Obad*** and the corresponding translation. We can see that, the obfuscation renames all classes, methods, fields to human unreadable forms (*e.g.*, *IljIllj*, *IliijIIl*, *etc.*). Furthermore, all invoke statements in the unobfuscated bytecode are translated to java reflection in the obfuscated version, and the name strings of such reflection are further encrypted and stored in a byte array list "IliijIIl." The decrypting method "*IljIllj.IliijIIl*" takes the byte array from "IliijIIl" and decrypts it and makes the reflection call. This clearly shows that the obfuscation can make both manual analysis and static analysis extremely difficult.

Dynamic Loading dex file becomes more popular nowadays. Normally it contains a dropper payload, which is lightweight and looks benign. But this dropper payload will then load the real payload from its assets or resource folder (*e.g.*, ***RuMMS***), or download the real payload from internet (*e.g.*, ***SlemBunk***). To further complicate the analysis, the real payload can even be encrypted (*e.g.*, ***Fobus***).

Native Payload: Most of the static analysis tools focus on Dalvik bytecode. So the native library seems to be a good place to hide malware behavior. In our analysis, we observed that native payloads are becoming more popular. Malware apps not only hide functionalities, but also hide sensitive strings, like server URL, premium numbers in the native code.

Evade Dynamic Analysis: The basic idea of evading dynamic analysis is to detect the malware's current running environment. For example, when ***BankBot*** [26] gets activated, it will check whether IMEI, MODEL, FINGERPRINT, MANUFACTURE, BRAND and DEVICE are of certain value. If the running environment satisfies the condition, it will act benignly and stop itself. ***Triada*** will check if IMEI matches some pattern, and check whether "com.qihoo.androidsandbox" is installed. To thwart dynamic analysis that monitors the communication channel (*e.g.*, Internet, SMS.) of the malware, many malware encrypt communication with their C&C servers.

Many of these anti-analysis techniques involve encryption; thus how to obtain the key is important to the analyst. In most of the cases, the key is just hardcoded in the application code. Some malware put the key in the manifest, a resource XML file, or in the native payload. We also observed a few smart ways to hide or generate the keys. ***Fobus*** reads the JVM stack trace and uses the class and method name of the fourth entry in the stack to construct the key. ***Obad*** obtains its key by requesting a webpage from Facebook, and reads certain location from that webpage to generate the key.

3.3 Monetization

We observe that many malware attempt to make money from the victims as Table 3 illustrate.

Premium Service Subscription. Subscribing to a premium service is one of the main ways cybercriminals use to make money. In general, subscribing to

Table 3. Monetization techniques.

Family	Premium Service Subscription	Bank	Ransom	Aggressive Advertising
Airpush	Dynamic			✓
AndroRAT	Dynamic			
Andup				✓
Aples			✓	
BankBot		✓		
Bankun		✓		
Boqx				
Boxer	Static			
Cova	Dynamic&Static			
Dowgin				✓
DroidKungFu				
Erop	Static			
FakeAV			✓	
FakeAngry				
FakeDoc	Static			
FakeInst	Dynamic&Static			
FakePlayer	Static			
FakeTimer				
FakeUpdates				
Finspy				
Fjcon	Dynamic			
Fobus	Dynamic			
Fusob			✓	
GingerMaster				
GoldDream	Dynamic			
Gorpo				✓
Gumen	Dynamic			
Jisut			✓	
Kemoge				
Koler			✓	
Ksapp				
Kuguo				✓
Kyview				✓
Leech				✓
Lnk				
Lotoor				
Mecor				
Minimob	Dynamic			✓
Mmarketpay	Dynamic			
MobileTX	Static			
Mseg	Dynamic			
Mtk				
Nandrobox	Static			
Obad	Dynamic			
Ogel				
Opfake	Dynamic&Static	✓		
Penetho				
Ramnit				
Roop			✓	
RuMMS	Dynamic	✓		
SimpleLocker			✓	
SlemBunk		✓		
SmsKey	Static			
SmsZombie		✓		
Spambot	Static			
SpyBubble				
Stealer	Dynamic			
Steek				
Svpeng	Dynamic	✓	✓	
Tesbo				
Triada	Dynamic	✓		
Univert	Dynamic			
UpdtKiller	Dynamic			
Utchi				✓
Vidro	Dynamic			
VikingHorde				✓
Vmvol	Dynamic			
Winge	Dynamic			
Youmi				✓
Zitmo		✓		
Ztorg				✓
Total families:	30	9	8	12
Total varieties:	41	27	13	13
Total apps:	11839	1652	2166	14336

a premium service requires the malware app to send a request to the service provider. The premium service sends back a confirmation message, which has to be entered back to finish the subscription process. A comprehensive premium-service-subscription module includes a premium service requester and an incoming message handler. The service requester makes phone calls, sends SMS or network request to the premium service. After that, the malware waits for the services to reply with confirmation message. The incoming message handler intercepts the confirmation message and parses it. It then applies a handler logic based on different subscription routines, and cleans any evidence that might alert the victim user.

In our analysis, we found the following ways to obtain the premium numbers and handler logic: hard coded into the bytecode, hidden in the resource XML files or native library, encrypted, and dynamically configured from C&C server.

Banking Trojan. Online payment and mobile wallet are becoming more popular nowadays. Cybercriminals are also putting much effort to increase their revenue by designing banking trojans. In 2013, banking trojan ***Bankun*** came into picture. Once activated this trojan will check the compromised device for installed Korean banking applications, and try to replace them with fake ones.

Newer versions of banking trojans are capable of overlaying the on-screen display of a legitimate banking app with a phishing window. ***Slembunk*** falls into this category. When this malware is activated, it will schedule a

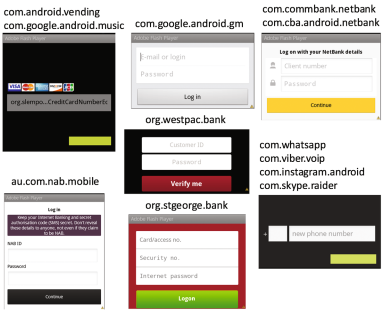


Fig. 4. Slembunk Phishing Windows

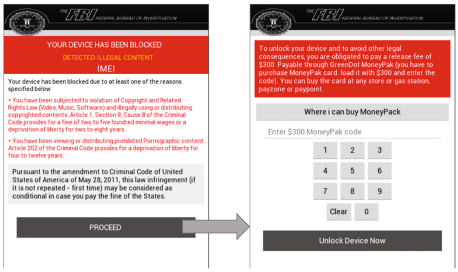


Fig. 5. Ransom Windows by Aples

java.lang.Runnable every 4 seconds to monitor the current running applications by looking at the Activity at the top of the Activity stack. If the current running Activity belongs to certain banking application, it will overlay a phishing window on top of the screen. Figure 4 shows what the phishing window looks like for different banking applications. As an example, if the current application is *com.android.vending* the left top window will be popped, and so on. ***Slembunk*** not only overlays phishing windows, it is also capable of forwarding phone calls and SMS from bank numbers, and applying the response logic. To effectively conceal the arrival of text messages or phone calls from banks, it will mute the device's audio system. Later versions of ***Slembunk*** even apply most sophisticated string encryption and dynamic loading obfuscation techniques (Sect. 3.2). Recently, IBM and FireEye report [9,17] that the source code of ***SlemBunk*** was leaked, which could result in the emergence of more variants.

Ransom. Ransomware locks the victim device by making it non-responsive or encrypting its data, and then coerces the victim to pay for the restoration.

Device Locking Techniques

Svpeng is both a banking trojan and ransomware. If its C&C server sends a command "forceLock," it will lock the infected device by using SYSTEM_ALERT_WINDOW permission and *WindowManager LayoutParams* with certain flags (*e.g.*, FLAG_SCREEN, FLAG_LAYOUT_IN_SCREEN, FLAG_WATCH_OUTSIDE_TOUCH, *etc.*) to achieve an unremovable full screen floating window.

Aples first appeared in 2014 – when activated, it will schedule a *Runnable* in every 0.1 second to load the threatening window with flag FLAG_ACTIVITY_NEW_TASK which looks like Fig. 5. Clicking on "PROCEED" at the first window will lead to the second window that asks the victim user to fill in a $300 MoneyPark code to unlock. Another malware family ***SimpleLocker*** has applied similar techniques, at the same time also encrypting all the data in the compromised device's external storage using AES with a hard-coded key.

Jisut once activated will launch a ransom window, and override *onKeyDown* method of *Activity* to redirect key press event (*e.g.*, return key, volume key, menu key, *etc.*) to some meaningless action to achieve the lock screen purpose.

Device UnLocking Techniques. After the victim has paid the money, the cybercriminal will tell the victim how to unlock the device or unlock it remotely. The most common way is to type in the pin. The pin in one variety of ***SimpleLocker*** is generated by obtaining a serial number at beginning (which is a random number), then uses some calculation logic (in one sample, the logic is *key = (serial_number - 2016)*2 + 2016*). The second way is using remote control. For instance, ***Koler*** uses network command to clear a lock tag at the malware's shared preference. The third way is by installing an unlock app. One variety of ***Jisut*** constantly checks whether an app with package "*tk.jianmo.study*" is installed or not; if yes, it will release the lock.

Aggressive Advertising. Mobile advertising is the main revenue source for app developers as well as malware writers. Advertising in malware is usually more aggressive, and this kind of apps are called adware.

Potentially Unwanted Application (PUA). A PUA adware performs tasks such as monitoring victim's personal data, showing unwanted advertisement content, annoying victim user with aggressive advertisement push, showing and tempting the victim to download and install potential harmful applications. ***Dowgin*** is one adware app. It will be activated once the device connectivity changes, user comes into presence, or a new application is installed or deleted. Once activated, it will display unwanted advertisements in the system's notification bar. If the victim clicks on this notification, it will show an application wall which attracts the victim to install new applications. At the same time, it will send device information and the list of installed apps to a remote C&C server using JSON, and receive commands for showing a new advertisement, uploading client info *etc.* Many other adwares have similar behaviors, *e.g.*, ***Airpush***, ***Kuguo***, ***Youmi***.

Malware Dropper. ***Gorpo***, ***Kemoge*** [30], ***Leech*** and ***Ztorg*** are some examples. Their task is to gain the root privilege on the infected device as discussed in Sect. 3.2, and then silently drop all the active malware apps that are available on the "malvertising campaign network" to the infected device. ***VikingHorde*** is running in two modes: rooted and not-rooted. If the device is not rooted, it performs in a regular fashion: uploading victim's data, fetch command from C&C to execute, *etc.* If the device is rooted, it will install some additional components, which are capable of constantly and silently downloading new malware onto the device. We include this as part of aggressive advertising even though their main purpose is spreading malware.

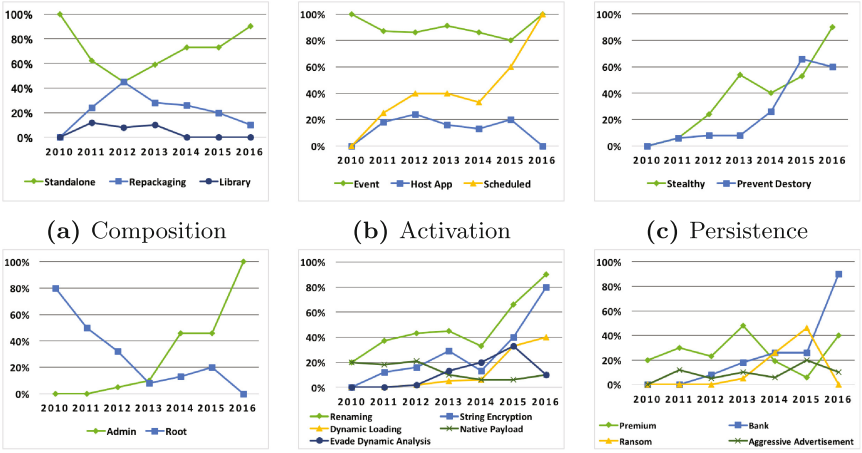
4 Evolution

We have performed a longitudinal study of our malware dataset with an attempt to discover the trend of malware behaviors and techniques used over the years from 2010 to 2016. For each type of behaviors and techniques, we observe the trend in terms of percentage of malware varieties manifesting a specific behavior/technique within a year. Figure 6 presents the results.

Figure 6a shows that the repackaging usage was growing until 2012, but later standalone malware became dominant. The reason could be that there are many effective anti-repackaging solutions made available during the last few years, which gives cybercriminals less incentive to use such techniques. On the other hand, the bad guys are putting more effort into designing comprehensive and sophisticated malware apps from scratch, and their malware design skill has matured.

Not surprising to see in Fig. 6b that listening to system events to activate malware's functional units is the main trick given the nature of Android system design. Scheduling a task to periodically start its functional unit is an alarmingly growing trend. By scheduling timer task or leveraging the *AlarmManager* the malware can constantly upload victim's information to or retrieve commands from the C&C server; in the ransomware apps, it is also one of the techniques to lock victim's device.

We observe that persistence has become a core feature of Android malware apps. Figure 6c shows that malware apps are evolving to be harder to notice by the victim, and harder to be destroyed by the system, anti-virus solutions, or users.



(a) Composition (b) Activation (c) Persistence

(d) Privilege (e) Anti-analysis (f) Monetizing

Fig. 6. Malware behavior trends

Root exploit is becoming less popular as we have discussed in Sect. 3.2, but obtaining device-admin-privilege seems to have become popular as seen in Fig. 6d.

The anti-analysis techniques are one of the key weapons of cybercriminals in the battle against security analysts. From Fig. 6e we can see that renaming and string encryption are the most growing techniques; dynamic loading and evading dynamic analysis are catching up while the practice of hiding behaviors in native payload is staying at the similar level.

Figure 6f shows that banking malware is becoming the main channel for cybercriminals to make money. Ransomware is a new threat that has started an uptick.

5 Related Work

The Android Malware Genome [34] was the first research project that has provided the community an Android malware dataset. This dataset has been the only well-labeled one and has been widely studied and used by the research community. Unfortunately, it has not been updated after its creation time around 2011. We comparatively studied this dataset with the new malware samples we have, and found that the Genome dataset does not include many of the new threats, which motivated us to carry out this work. Our dataset also provides much more detailed information on Android malware behaviors than that in Genome. Moreover, we provide detailed documentation of the process used in creating the dataset, including the guidelines for the manual analysis, to help other researchers do the same.

Recently, the *AndroZoo* [10] dataset has been published, which contains more than 3 million Android apps from Google Play, other smaller markets, and app repositories. *AndroZoo*'s goal is to create a comprehensive app collection for software engineering studies. Our goal is different and we focus on (only) malware apps to study their security related behaviors. Our dataset provides malware labels and detailed behavior information of the malware.

There are a few other repositories for Android malware apps which researchers can use, such as *Contagio Minidump* [2] and *VirusShare* [6]. However, they do not provide a comprehensive malware collection or comprehensive label and behavior information on the malware.

The *ANDRUBIS* [19] combines static and dynamic analysis to automatically extract feature and behaviors from Android apps, and studies the changes in the malware threat landscape and trends among "goodware," or benign apps, developers. However, as many behaviors are either unknown or can evade the automated analysis method, this work cannot give a comprehensive understanding of the malware landscape as we produced through the systematic and deep manual analysis.

AVclass [25] provides a method to extract malware family name by processing the AV labels obtained from VirusTotal. We adopted a similar approach for identifying malware family label. Our work is focused on deep manual analysis

of malware samples from different malware varieties, and reporting the detailed behavioral profiles for Android malware.

There has been quite some work on how to detect malicious apps. The *Drebin* [11] work applies machine learning (ML) techniques to Android malware detection. The authors made the set of *feature vectors* used in the ML work available to the community. More recently, *MassVet* [15] provides a method to detect malware apps by observing the repackaging traits (if any) compared to that of other apps. Rastogi, *et al.* [24] conducted research on identifying adware tricks and drive-by-download techniques. *Harvestor* [23] attempts to extract the *run-time* values from obfuscated apps to detect malware. Researchers have identified ways in which Android users can be deceived to misidentify a malicious app window as a legitimate app's [12]. Moreover, *CopperDroid* [27] is a dynamic analysis system which attempts to reconstruct the behaviors of Android malware. Our work complements these and other Android malware analysis work by providing a comprehensive dataset of Android malware with detailed label and behavior information, which can facilitate future research in this area.

6 Conclusion

We created a large volume of well-labeled and well-studied Android malware dataset containing 24,650 samples, categorized in 135 varieties among 71 families ranging from 2010 to 2016. For each variety of this dataset we conduct a comprehensive study to profile their behaviors and evolution trends. We document in details the process of creating this dataset to enable other researchers to replicate the process. We observe that Android malware are evolving towards monetization, and becoming sophisticated and persistent. The extensive usage of anti-analysis techniques in the malware samples shows the urgent need for advanced de-obfuscation and dynamic analysis methods. We will make the dataset available to research community.

Acknowledgment. This research is partially supported by the U.S. National Science Foundation under Grant No. 1622402. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation. This research is also partially supported by David and Amy Fulton Grant received by co-author Roy at Bowling Green State University.

References

1. Building Web Apps in WebView. https://developer.android.com/guide/webapps/webview.html
2. Contagio Mobile Malware Mini Dump. http://contagiominidump.blogspot.com/
3. Drive-by Download. https://en.wikipedia.org/wiki/Drive-by_download
4. hexdump. https://www.freebsd.org/cgi/man.cgi?query=hexdump&sektion=1
5. IDA. https://www.hex-rays.com/products/ida/index.shtml
6. VirusShare. https://virusshare.com/

7. VirusTotal. https://www.virustotal.com/
8. IDC: Smartphone OS Market Share 2015, 2014, 2013, and 2012 (2015). http://www.idc.com/prodserv/smartphone-os-market-share.jsp
9. A Growing Number of Android Malware Families Believed to Have a CommonOrigin: A Study Based on Binary Code (2016). https://community.fireeye.com/external/1438
10. Allix, K., Bissyand, T.F., Klein, J., Le Traon, Y.: AndroZoo: collecting millions of android apps for the research community. In: Proceedings of the Mining Software Repositories (MSR) (2016)
11. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: effective and explainable detection of android malware in your pocket. In: Proceedings of the NDSS (2014)
12. Bianchi, A., Corbetta, J., Invernizzi, L., Fratantonio, Y., Kruegel, C., Vigna, G.: What the app. is that? deception and countermeasures in the android user interface. In: Proceedings of the IEEE S&P (2015)
13. Nikita Buchka. Taking root (2015). https://securelist.com/blog/mobile/71981/taking-root/
14. Buchka, N., Kuzin, M.: Attack on Zygote: a new twist in the evolution of mobile threats (2016). https://securelist.com/analysis/publications/74032/attack-on-zygote-a-new-twist-in-the-evolution-of-mobile-threats/
15. Chen, K., Wang, P., Lee, Y., Wang, X., Zhang, N., Huang, H., Zou, W., Liu, P.: Finding unknown malice in 10 seconds: mass vetting for new threats at the google-play scale. In: Proceedings of the USENIX Security Symposium, pp. 659–674 (2015)
16. Hurier, M., Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: On the lack of consensus in anti-virus decisions: metrics and insights on building ground truths of android malware. In: Caballero, J., Zurutuza, U., Rodríguez, R.J. (eds.) DIMVA 2016. LNCS, vol. 9721, pp. 142–162. Springer, Cham (2016). doi:10.1007/978-3-319-40667-1_8
17. Kessem, L.: Android Malware About to Get Worse: GM Bot Source Code Leaked (2016). https://securityintelligence.com/android-malware-about-to-get-worse-gm-bot-source-code-leaked/
18. Li, Y., Jang, J., Hu, X., Ou, X.: Android Malware Clustering through Malicious Payload Mining. Technical Report 2017–1, Argus Cybersecurity Lab, University of South Florida (2017). http://www.arguslab.org/tech_reports/2017-1
19. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., Platzer, C.: 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), pp. 3–17. IEEE (2014)
20. Maggi, F., Bellini, A., Salvaneschi, G., Zanero, S.: Finding non-trivial malware naming inconsistencies. In: Jajodia, S., Mazumdar, C. (eds.) ICISS 2011. LNCS, vol. 7093, pp. 144–159. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25560-1_10
21. Mohaisen, A., Alrawi, O.: AV-Meter: an evaluation of antivirus scans and labels. In: Dietrich, S. (ed.) DIMVA 2014. LNCS, vol. 8550, pp. 112–131. Springer, Cham (2014). doi:10.1007/978-3-319-08509-8_7
22. Polkovnichenko, A., Koriat, O., Horde, V.: A New Type of Android Malware on Google Play (2016). http://blog.checkpoint.com/2016/05/09/viking-horde-a-new-type-of-android-malware-on-google-play/

23. Rasthofer, S., Arzt, S., Miltenberger, M., Bodden, E.: Harvesting runtime values in android applications that feature anti-analysis techniques. In: Proceedings of the NDSS (2016)
24. Rastogi, V., Shao, R., Chen, Y., Pan, X., Zou, S., Riley, R.: Are these ads safe: detecting hidden attacks through the mobile app-web interfaces. In: Proceedings of the NDSS (2016)
25. Sebastián, M., Rivera, R., Kotzias, P., Caballero, J.: AVclass: a tool for massive malware labeling. In: Monrose, F., Dacier, M., Blanc, G., Garcia-Alfaro, J. (eds.) RAID 2016. LNCS, vol. 9854, pp. 230–253. Springer, Cham (2016). doi:10.1007/978-3-319-45719-2_11
26. Shatilin, I.: Banking Trojans: mobile's major cyberthreat. https://blog.kaspersky.com/android-banking-trojans/9897/
27. Tam, K., Khan, S.J., Fattori, A., Cavallaro, L.: CopperDroid: automatic reconstruction of android malware behaviors. In: Proceedings of the NDSS (2015)
28. Wei, F., Roy, S., Ou, X., Robby: Amandroid: a precise and general inter-component data flow analysis framework for security vetting of android apps. In: Proceedings of the CCS (2014)
29. Woods, V., van der Meulen, R.: Gartner Says Emerging Markets Drove Worldwide Smartphone Sales to 15.5 Percent Growth in Third Quarter of 2015 (2015). http://www.gartner.com/newsroom/id/3169417
30. Zhang, Y.: Kemoge: Another Mobile Malicious Adware Infecting Over 20 Contries (2015). https://www.fireeye.com/blog/threat-research/2015/10/kemoge_another_mobi.html
31. Zhou, W., Chen, Z., Su, J., Xie, J., Huang, H.: SlemBunk: An Evolving Android Trojan Family Targeting Users of Worldwide Banking Apps (2015). https://www.fireeye.com/blog/threat-research/2015/12/slembunk_an_evolvin.html
32. Zhou, W., Deyu, H., Jimmy, S., Yong Kang, R.: The Latest Family of Android Malware Attacking Users in Russia via SMS Phishing (2016). https://www.fireeye.com/blog/threat-research/2016/04/rumms-android-malware.html
33. Zhou, W., Huang, H., Chen, Z., Xie, J., Su, J.: SlemBunk Part II: Prolonged Attack Chain and Better-Organized Campaign (2016). https://www.fireeye.com/blog/threat-research/2016/01/slembunk-part-two.html
34. Zhou, Y., Jiang, X.: Dissecting android malware: Characterization and evolution. In: Proceedings of the IEEE S&P (2012)