DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware

Min Zheng, Mingshen Sun, John C.S. Lui
Computer Science & Engineering Department
The Chinese University of Hong Kong

Abstract—**Smartphones and mobile devices are rapidly becoming indispensable devices for many users. Unfortunately, they also become fertile grounds for hackers to deploy malware. There is an urgent need to have a "*security analytic & forensic system*" which can facilitate analysts to examine, dissect, associate and correlate large number of mobile applications. An effective analytic system needs to address the following questions: *How to automatically collect and manage a high volume of mobile malware? How to analyze a zero-day suspicious application, and compare or associate it with existing malware families in the database? How to reveal similar malicious logic in various malware, and to quickly identify the new malicious code segment?* In this paper, we present the design and implementation of *DroidAnalytics*, a signature based analytic system to automatically collect, manage, analyze and extract android malware. The system facilitates analysts to retrieve, associate and reveal malicious logics at the "*opcode level*". We demonstrate the efficacy of DroidAnalytics using 150,368 Android applications, and successfully determine 2,494 Android malware from 102 different families, with 342 of them being *zero-day* malware samples from six different families. To the best of our knowledge, this is the first reported case in showing such a large Android malware analysis/detection. The evaluation shows the DroidAnalytics is a valuable tool and is effective in analyzing malware repackaging and mutations.**

I. Introduction

Smartphones are becoming prevailing devices for many people. Unfortunately, malware on smartphones is also increasing at an unprecedented rate. Android OS-based systems, being the most popular platform for mobile devices, have been a popular target for malware developers. As stated in [1], the exponential growth of mobile malware is mainly due to the ease of generating malware variants. Although there are number of works which focus on Android malware detection via permission leakage, it is equally important to design a system that can perform comprehensive *malware analytics*: analyze and dissect suspicious applications at the *opcode level* (instead at the permission level), and correlate the application with existing malware in the database to determine whether it is mutated malware or zero-day malware, as well as which legitimate applications are infected.

Challenges: To realize an effective analytic system for Android mobile applications, we need to overcome several technical hurdles. First, how to systematically *collect* malware from the wild. As indicated in [2], new malware variants are always hidden in many different third-party markets. Due to the competition of anti-virus companies and their fear of accidentally releasing malware to the public, companies are usually reluctant to share their malware database to researchers. Researchers can only obtain a handful of mobile malware samples. Hence, how to *automate a systematic process* to collect these malicious applications is the first hurdle we need to overcome.

The second hurdle is how to identify *repackaged applications* (or mutated malware) from the vast ocean of applications and malware. As reported in [3], hackers can easily transform legitimate applications by injecting malicious logic or obfuscated program segments so that they have the same structure as the original application but contain malicious logic. Thus, how to determine whether an application is a repackaged or obfuscated malware, and which legitimate applications are infected is very challenging.

The third hurdle is how to *associate* a malware with existing malware (or application) so as to facilitate security analysis. The existing approach of using cryptographic hash or package name as an identifier is not effective because hackers can easily change the hash value or package name. Security analysts often need to go through a laborous process of reverse engineering to discover malicious functions and structure. There is an urgent need to have an efficient method to associate malware with other malware in the database, so to examine their commonalities at the opcode level.

Contributions: We present the design and implementation of *DroidAnalytics*, an Andorid malware analytic system for malware collection, signature generation, information retrieval, and malware association based on similarity score. Furthermore, DroidAnalytic can efficiently detect zero-day repackaged malware. The contributions of our system are:

- DroidAnalytics automates the processes of malware collection, analysis and management. We have successfully collected 150,368 Android applications, and determined 2,494 malware samples from 102 families. Among those, there are 342 zero-day malware samples from six different malware families. We plan to release the malware database to the research community. (for our malware database, please refer to http://ansrlab.cse.cuhk.edu.hk/software/droidanalytics).
- DroidAnalytics uses a *multi-level signature algorithm* to extract the malware feature based on their semantic

meaning at the *opcode level*. This is far more robust than a cryptographic hash of the entire application. We show how to use DroidAnalytics to combat against malware which uses repackaging or code obfuscation, as well as how to analyze malware with dynamic payloads (see Sec. III).

- Unlike previous works which associate malware via "*permission*", DroidAnalytics associates malware and generates signatures at the app/class/method level. Hence, we can easily track and analyze mutation, derivatives, and new generation of a malware. DroidAnalytics can reveal malicious behavior at the method level to identify repackaged malware, and perform class association among malware/applications (see Sec. IV).
- One can use DroidAnalytics to detect *zero-day* repackaged malware. We have found 342 zero-day repackaged malware in six different families (see Sec. V).

II. Design & Implementation of DroidAnalytics

AndroidAnalytics consists of modules for automatic malware collection, signature generation, information retrieval and association, as well as similarity comparison between malware. One can also use these functions to detect zero-day repackaged malware.

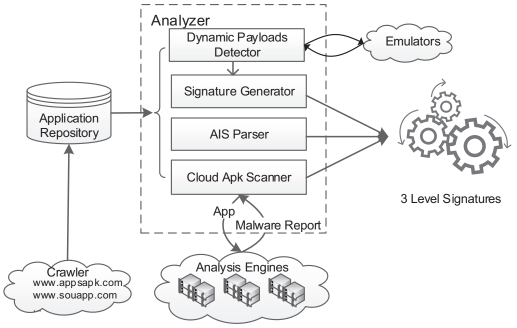


Figure 1. The Architecture of the DroidAnalytics

A. Building Blocks of DroidAnalytics

Figure 1 depicts the architecture of DroidAnalytics and its components. Let us explain the design of each component.

• **Extensible Crawler:** In DroidAnalytics, we implement an application crawler based on Scrapy [4]. Users can specify official or third party market places, as well as blog sites and the crawler will perform regular mobile application download. The crawler enables us to systematically build up the mobile applications database for malware analysis and association. So far, we have collected 150,368 mobile applications and carried out detailed security analysis.

• **Dynamic Payloads Detector:** To deal with malware which dynamically downloads malicious codes via the Internet or attachment files, we have implemented the dynamic payloads detector component, which determines malicious trigger code within malware packages and tracks the downloaded application and its behavior in virtual machine. Firstly, it scans the package to find suspicious files such as `.elf` or `.jar` file. Hackers usually camouflage malicious files by changing their file type. To overcome this, this component scans all files and identifies files using their *magic numbers* instead of file extension. Secondly, if an application has any Internet behavior (e.g., Internet permission or re-delegating other applications to download files [5]), the dynamic payloads detector will treat these files as the target, then runs the application in the emulator. The system will use the forward symbolic execution technique to trigger the download behavior. For detail, please refer to [6]. Both the suspicious files within the package and dynamically downloaded files from the Internet will be sent to the signature generator (which we will shortly describe) for further analysis.

• **Android App Information (AIS) Parser:** AIS is a data structure within DroidAnalytics and it is used to represent `.apk` information structure. Using the AIS parser, analysts can reveal the cryptographic hash (or other basic signature) of an `.apk` file, its package name, permission information, broadcast receiver information and disassembled code, . . ., etc. Our AIS parser decrypts the `AndroidManifest.xml` within an application and disassembles the `.dex` file into `.smali` code. Then it extracts package information from source code and retains it in AIS so analysts can easily retrieve this information.

• **Signature Generator:** Anti-virus companies usually use cryptographic hash, e.g., MD5, to generate a signature for an application. This has two major drawbacks. Firstly, hackers can easily mutate an application and change its cryptographic hash. Secondly, the cryptographic hash does not provide sufficient flexibility for security analysis. In DroidAnalytics, we use a *three-level* signature generation scheme to identify each application. This signature scheme is based on the mobile application, classes, methods, as well as malware's dynamic payloads (if any). Our signature generation is based on the following observation: For any functional application, it needs to invoke various Android API calls, and Android API calls sequence within a method is difficult to modify (unless one drastically changes the program's logic, but we did not find any from the 150,368 applications we collected that used this obfuscation technique). Hence, we generate a method's signature using the API call sequence, and given the signature of a method, create the signature of a class which composes of different methods. Finally, the signature of an application is composed of all signatures of its classes. We like to emphasize that our signature algorithm is not only for defense against malware obfuscation, but more importantly, facilitating malware analysis via class/method association (we will show in later sections). Let us present the detail of signature generation.

(a) Android API calls table: Our system uses the API calls table of the Android SDK. The `android.jar` file is the framework package provided by the Android SDK. We use the Java reflection [7] to obtain all descriptions of the API calls. For each API, we extract both the *class path* and the *method name*. We assign each full path method a hex number as part of the ID. For the current version of DroidAnalytics, we extract 47,126 full path methods in the Android SDK 4.1 version as our API calls table. Table I depicts a *snapshot* of API calls table, e.g., `android/content/Intent;-><init>` is assigned an ID 0x30291.

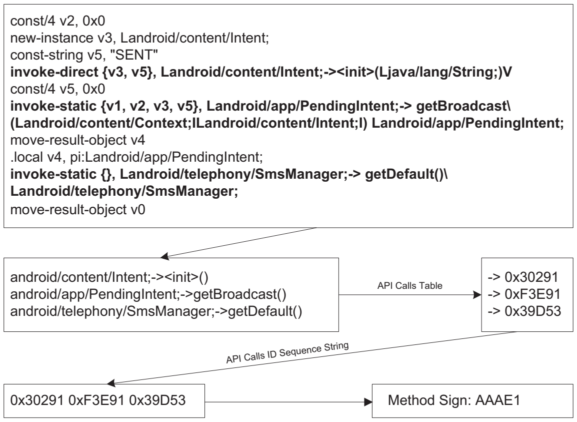
Full Path Method	**Method ID**
`android/accounts/Account;-><init>`	0x00001
:	:
`android/content/Intent;-><init>`	0x30291
`android/content/Intent;->toUri`	0x30292
`android/telephony/SmsManager;->getDefault`	0x39D53
`android/app/PendingIntent;->getBroadcast`	0xF3E91

Table I
EXAMPLE OF THE ANDROID API CALLS TABLE AND ASSIGNED IDS

(b) Disassembling process: Each Android application is composed of different classes and each class is composed of different methods. To generate signatures for each class or method, DroidAnalytics first disassembles an `.apk` file, then takes the Dalvik opcodes of the `.dex` file and transforms them to methods and classes. Then DroidAnalytics uses the Android API calls table to generate signatures.

(c) Generate Lev3 signature (or method signature): The system first generates a signature for each method and we call this the *Lev3 signature*. Based on the Android API calls table, the system extracts the API call ID sequence as a string in each method, then hashes this string value to produce the method's signature. Figure 2 illustrates how to generate the Lev3 signature of a method which sends messages to another mobile phone. Figure 2 shows that the method contains three API calls. Using the Android API calls table (as in Table I), we determine their IDs. Signature of a method is generated by cancatenation of all these IDs. Note that DroidAnalytics will not extract the API calls which will not be executed in run time because these codes are usually generated via obfuscation. Furthermore, if a method (except the main method) will not be invoked by any other methods, signature generator will also ignore this method because this may be a defunct method generated by malware writers.

(d) Generate Lev2 signature (either class signature or dynamic payload signature): Next, DroidAnalytics proceeds to generate the Lev2 signature for each *class*, and it is based on the Lev3 signatures of methods within that class. Malware writers may use various obfuscation or repackaging techniques to change the *calling order* of the methods table in a `.dex` file. To overcome this problem, our signature generation algorithm will first *sort* the level 3 signatures within that class, and then concatenate all these level 3 signatures to form the level 2 signature.



```
const/4 v2, 0x0
new-instance v3, Landroid/content/Intent;
const-string v5, "SENT"
invoke-direct {v3, v5}, Landroid/content/Intent;-><init>(Ljava/lang/String;)V
const/4 v5, 0x0
invoke-static {v1, v2, v3, v5}, Landroid/app/PendingIntent;-> getBroadcast\
(Landroid/content/Context;ILandroid/content/Intent;I) Landroid/app/PendingIntent;
move-result-object v4
.local v4, pi:Landroid/app/PendingIntent;
invoke-static {}, Landroid/telephony/SmsManager;-> getDefault()\
Landroid/telephony/SmsManager;
move-result-object v0
```

```
android/content/Intent;-><init>()
android/app/PendingIntent;->getBroadcast()
android/telephony/SmsManager;->getDefault()
```

```
-> 0x30291
-> 0xF3E91
-> 0x39D53
```

```
0x30291 0xF3E91 0x39D53
```

```
Method Sign: AAAE1
```

Figure 2. The Process of Lev3 Signature Generation

Some malicious codes are dynamically downloaded from the Internet during execution. DroidAnalytics uses the *dynamic payloads detector* component to obtain the payload files. For the dynamic payloads which are `.dex` file or `.jar` file, DroidAnalytics treats them as classes within the malware. Given these files, the system checks their API call sequence and generates a Lev2 signature for each class within an application. For the dynamic payloads which contain, say, `.elf` file or `.so` file, DroidAnalytics treats them as a single class within that malware, then uses the cryptographic hash value (e.g., MD5) of the payload as its Lev2 signature. For the dynamic payloads which are `.apk` files, DroidAnalytics treats each as a new application and a class within the malware. DroidAnalytics first uses the cryptographic hash value (e.g., MD5) of the new `.apk` file as one Lev2 signature of that malware. Because the payload is a new application, DroidAnalytics will use the method we discussed to carry out a new signature generation.

(e) Generate Lev1 signature (or application signature): The Lev1 signature is based on the level 2 signatures, e.g., signatures of all qualified classes within an application. In addition, the signature generator will ignore the class (except the main class) which will not be invoked by any other classes since these defunct classes may be generated via obfuscation. Malware writers may use some repackaging or obfuscation techniques to change the order of the classes table of the `.dex` file, our signature algorithm will first *sort* all Lev2 signatures, then concatenate these Lev2 signatures to generate the Lev1 signature.

Figure 3 summarizes the signature genration framework. For example, the Lev3 signatures of `AAAE1` and `B23E8` are the two method signatures within the same class. Based on these two (sorted) signatures, we generate the Lev2 signature of the corresponding class, which is `53EB3`. Note that the Lev2 signature of `C3EB3` is generated from a `.dex` file which is a dynamic payload used to execute the malicious behavior. Based on all sorted Lev2 signatures of all classes, we generate the Lev1 signature, `F32DE`, of the application.

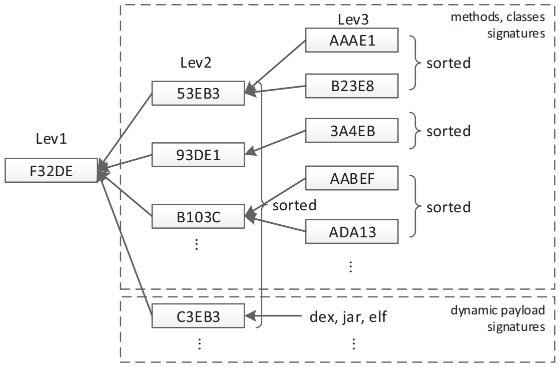


Figure 3. Illustration of signature generation: the application (Lev1) signature, class level (Lev2) signatures and method level (Lev3) signatures.

For the current DroidAnalytics platform, we use a server which is of 2.80 GHz Duo CPU processor, 4GB memory and 2 TB hard disk, with two virtual machines in the server to implement the anti-virus engine. We carried out experiment to study the processing time to scanning and generating signatures. On average, it takes around 60 seconds to scan one application (includes the dynamic analysis), and around three seconds to generate all three level signatures, five seconds to generate AIS information, and one second to insert information into the database. As of November 2012, our system have downloaded 150,368 mobile applications from the following places: Google Play [8], nine Android third party markets (e.g., [9]–[11]), two malware forums [12], [13] and one mobile malware share blog [2]. The size of all downloaded application is 468GB.

III. Utility & Effectiveness of Signature Based System

Here, we illustrate how DroidAnalytics' signatures can be used to analyze (and detect) malware repackaging, code obfuscation and malware with dynamic payloads.

A. Analyzing Malware which uses Code Obfuscation: A malware writer can use a disassembler (e.g., Apktool [14]) to convert a `.dex` file into `.smali` files, then injects new malware logic into the `.smali` code, rebuilds it back to a `.dex` file. Based on this, malware writers can apply various code obfuscation techniques while preserve the behavior as the original one in order to bypass the anti-virus detection. As shown in [3], [15], many mobile anti-virus products are not effective to detect code obfuscated variants.

DroidAnalytics will not extract the API calls in methods and classes which will not be executed in run time (refer to Sec. II) because they are defunct and can be generated by obfuscators. In addition, our signature generation does not depend on the name of methods or classes and so name obfuscation has no effect on our signature. Furthermore, the signature generation of DroidAnalytics is based on the analyst-defined API calls table. So one can flexibly update the table to defend against various code obfuscation.

Experiment. To illustrate the effectiveness of DroidAnalytics against code obfuscation, we chose 30 different malware from three different families (10 samples in each family) and Table II illustrates our results. The malware families in our study are: Basebrid (or Basebridge), GoldDream, and Kungfu. We use ADAM [16], a system which can automatically transform an original malware sample to different variants by various repackaging and obfuscation techniques (e.g. inserting defunct methods, modifying methods name, ...etc). We generate seven different variants for each malware. Then we put these 240 new malware samples into the DroidAnalytics system and check their signatures. After our signature calculations, the result shows that for each malware, the original sample and seven mutated variants have *distinct* MD5 hash values (3 repackaging, 4 code obfuscation), but all of them have the *same level 1 signature*. This shows that DroidAnalytics' signature system is effective in defending against code obfuscation.

B. Analyzing Malware with Attachement Files or Dynamic Payloads: Some malware will dynamically download file which contains the malicious code. Also, some attachment files within a package may contain malicious logic but they can be concealed as other valid documents (e.g., `.png` file, `.wma` file). DroidAnalytics will treat these files as dynamic payloads. By using static and dynamic analysis techniques described in Section II, DroidAnalytics accesses these payloads and generates different signatures.

Experiment. We carried out the following experiment. From our malware database, we used our signature system and detected some malware contain the same file with a `.png` filename extension. But when we check the magic number of this file, it is actually an `.elf` file. Upon further analysis, we found that this file is a root exploit and this malware belongs to the GinMaster (or GingerMaster) family. Another example is the Plankton family. By using dynamic analysis, DroidAnalytics discovered that all malware in this family will download a `plankton_v0.0.4.jar` (or similar `.jar`) when the main activity of the application starts. Further analysis revealed the `.jar` file contains malicious behavior, i.e., stealing browser's history information, making screen shortcuts and botnet logic. Table III depicts DroidAnalytics system detects some representative malware using dynamic payloads.

MD5	Dynamic Payload	Description	Malware Family
34cb03276e426f8d61e782b8435d3147	/assets/runme.png	ELF file to expoit root	GinMaster
a24d2ae57c3cee1cf3298c856a917100	/assets/gbfm.png /assets/install.png /assets/installsoft.png /assets/runme.png	ELF file to expoit root	GinMaster
9e847c9a27dc9898825f466ea00dac81	/assets/gbfa.png /assets/install.png	ELF file to expoit root	GinMaster
dcbe11e5f3b82ce891b793ea40e4975e	plankton_v0.0.4.jar	download in runtime	Plankton

Table III
EXAMPLES OF DYNAMIC PAYLOADS

We also carried out experiment to analyze repackaged malware. For detail, please refer to [6].

MD5	Pakage Name	Lev1 Signature	Malware Family	Description
f7967f71b2f32287660ae9a3fa366022	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Original malware
f2e2d727f95fa868fd7ff54459e766e3	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Repackaging
e01f573cca83fdf2737be6ecee35fe33	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Repackaging
d383ceeb9c6ffcff8c0dd12b73ec43e3	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Repackaging
cd040541693693faca9fec646e12e7e6	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Obfuscation
401952d745cd7ca5281a7f08d3e2eede	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Obfuscation
271c3965c7822ebf944feb8bbd1cfe7f	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Obfuscation
8b12ccdc8a69cf2d6a7e6c00f698aaaa	com.tutusw.phonespeedup	3c83ed0f80646c8ba112cb8535c293dd	Kungfu	Obfuscation

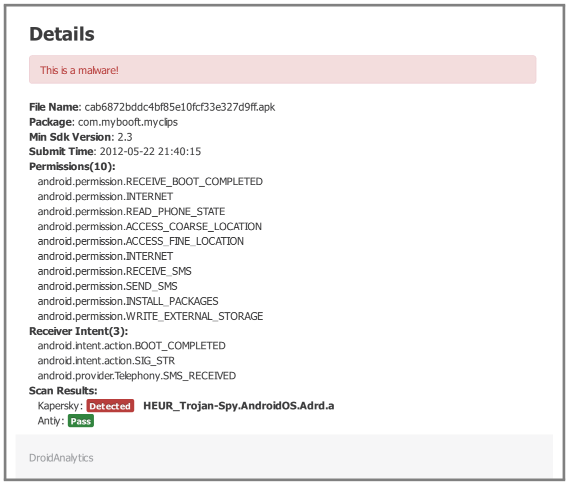
Table II
EXAMPLES OF CODE OBFUSCATION

IV. Analytic Capability of DroidAnalytics

We conduct three experiments and show how analysts can study malware, carry out similarity measurement between applications, as well as perform class association among 150,368 mobile applications in the database.

A. Detailed Analysis on Malware: Using DroidAnalytics, analysts can also discover which class or method uses suspicious API calls via the *permission recursion* technique.

• **Common Analytics on Malware.** First, using the AIS parser, DroidAnalytics can reveal basic information of an application like the cryptographic hash (i.e., MD5 value), package name, broadcast receiver, . . ., etc. This is illustrated in Figure 4. In addition, DroidAnalytics has a built-in cloud-based APK scanner that supports diverse anti-virus scan results (e.g., Kaspersky and Antiy) to help analysts for reference. Our cloud-based APK scanner is *extensible* to accommodate other anti-virus scan engines. Last but not least, DroidAnalytics can disassemble the `.dex` file and extract class number, method number, and API calls number in each application, class or method. These functionalities are useful so analysts can zoom in to the meta-information of a suspicious malware. Figure 5 shows these functionalities.



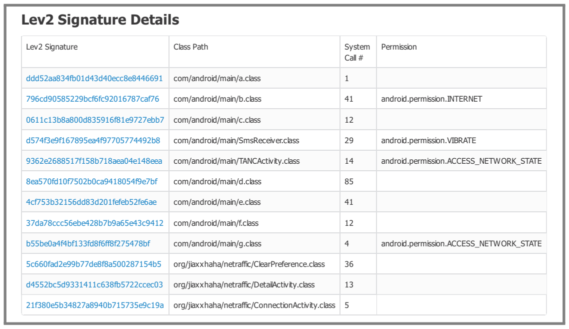
Details

This is a malware!

File Name: cab6872bddc4bf85e10fcf33e327d9ff.apk
Package: com.mybooft.myclips
Min Sdk Version: 2.3
Submit Time: 2012-05-22 21:40:15
Permissions(10):
android.permission.RECEIVE_BOOT_COMPLETED
android.permission.INTERNET
android.permission.READ_PHONE_STATE
android.permission.ACCESS_COARSE_LOCATION
android.permission.ACCESS_FINE_LOCATION
android.permission.INTERNET
android.permission.RECEIVE_SMS
android.permission.SEND_SMS
android.permission.INSTALL_PACKAGES
android.permission.WRITE_EXTERNAL_STORAGE
Receiver Intent(3):
android.intent.action.BOOT_COMPLETED
android.intent.action.SIG_STR
android.provider.Telephony.SMS_RECEIVED
Scan Results:
Kapersky: Detected **HEUR_Trojan-Spy.AndroidOS.Adrd.a**
Antiy: Pass

DroidAnalytics

Figure 4. Screen Capture of Common Analytics on Malware



Lev2 Signature Details

Lev2 Signature	Class Path	System Call #	Permission
ddd52aa834fb01d43d40ecc8e8446691	com/android/main/a.class	1	
796cd90585229bcf6fc92016787caf76	com/android/main/b.class	41	android.permission.INTERNET
0611c13b8a800d835916f81e9727ebb7	com/android/main/c.class	12	
d574f3e9f167895ea4f97705774492b8	com/android/main/SmsReceiver.class	29	android.permission.VIBRATE
9362e2688517f158b718aea04e148eea	com/android/main/TANCActivity.class	14	android.permission.ACCESS_NETWORK_STATE
8ea570fd10f7502b0ca9418054f9e7bf	com/android/main/d.class	85	
4cf753b32156dd83d201fefeb52fe6ae	com/android/main/e.class	41	
37da78ccc56ebe428b7b9a65e43c9412	com/android/main/f.class	12	
b55be0a4f4bf133fd8f6ff8f275478bf	com/android/main/g.class	4	android.permission.ACCESS_NETWORK_STATE
5c660fad2e99b77de8f8a500287154b5	org/jiaxxhaha/netraffic/ClearPreference.class	36	
d4552bc5d9331411c638fb5722ccec03	org/jiaxxhaha/netraffic/DetailActivity.class	13	
21f380e5b34827a8940b715735e9c19a	org/jiaxxhaha/netraffic/ConnectionActivity.class	5	

Figure 5. Screen Capture of Detailed Lev2 Signature

• **Permission Recursion.** Current state-of-the-art systems examine the `AndroidManifest.xml` to discover permissions of an application. This is not informative enough since analysts do not know which class or which method uses these permissions for suspicious activities. In DroidAnalytics, we can discover the permission within a class or a method. Since each permission is related to some API calls [17]. In DroidAnalytics, we tag permission to API calls in each method. We combine the method permissions within the same class as class permission, and combine all class permissions as application permission. This helps analysts to quickly discover suspicious methods or classes.

– **Experiment.** In this experiment, we choose a popular malware family Kungfu and examine the permissions at the application/class/method levels. Malware in the Kungfu family can obtain user's information such as IMEI number, phone model,..., etc. It can also exploit the device and gain root privilege access. Once the malware obtained the root level access, it installs malicious application in the background as a back-door service.

We use DroidAnalytics to generate all three-level signatures. Figure 6 shows the partial structure of the signatures with permission recursion of two Kungfu malware: `A4E39D` and `D2EF8A`, and together with a legitimate application `BEDIE3` (Lev1 signature). Firstly, based on the malware reports by our cloud APK scanner, we identify `A4E39D` and `D2EF8A` are malware which come from the Kungfu family with different package names, `net.atools.android.cmwrap` and `com.atools.netTrafficStats` respectively. By analyzing the Lev2 signature, we discover that `BCEED` is the common class which is within the two Kungfu applications. Secondly, from the Lev3 signature of `BCEED`, we use the permission recursion method to expose the method `6F100`

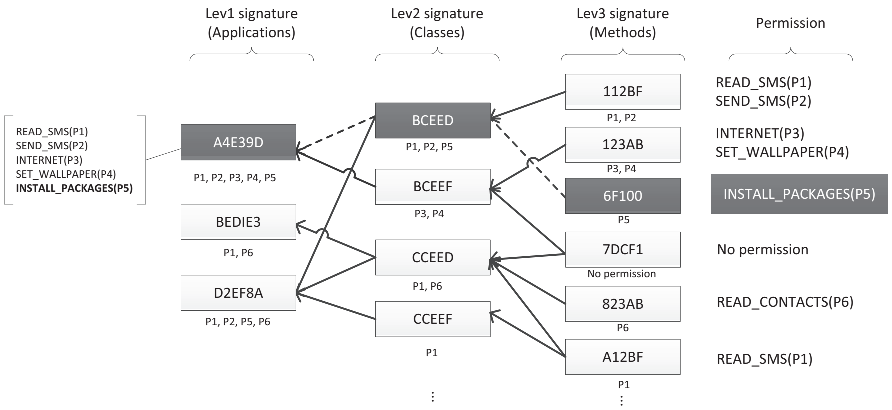


Figure 6. Illustration of Detail Analytics on Malware

with the `INSTALL_PACKAGES` permission. Note that the `INSTALL_PACKAGES` permission is a system permission which the application can install other unrelated applications. This shows how DroidAnalytics helps analysts to quickly discover the malicious code of methods and classes of the Kungfu family.

• **Class Association.** Traditional analysis on malware only focuses on one malware but can not associate malware with other malware or applications. DroidAnalytics can associate legitimate applications and other malware in the class level and/or method level. Given a class signature (or Lev2 signature), DroidAnalytics keeps track of how many legitimate applications or malware using this particular class. Also, with the methodology of permission recursion, DroidAnalytics can indicate the permission usage of this class signature. By using class association, we can easily determine which class or method may possess malicious behavior, and which class is used for common task, say for pushing advertisement. Lastly, for class signatures which are used by many known malware but zero legitimate application, these are classes that analysts need to pay special attention to because it is very likely that they contain malicious code and are used in many repackaged or obfuscated malware.

− **Experiment.** Using DroidAnalytics, we carry out the class association experiment using 1,000 legitimate appliactions and 1,000 malware as reported by Kaspersky. After the class association, we discover a class (Lev2) signature `2bcb4f8940f00fb7f50731ee341003df` which is used by 143 malware and zero legitimate application. In addition, the 143 malware are all from the Geinimi family. Furthermore, this class has 47 API calls and uses `READ_CONTACTS` and `SEND_SMS` permissions. Therefore, we quickly identify this class contains malicious codes. In the signature database, we also find a class (Lev2) signature `9067f7292650ba0b5c725165111ef04e` which is used by 80 legitimate applications and 42 malware. Further analysis shows that this class is used by similar number of legitimate applications and malware, and this class uses an advertisement library called DOMOB [18]. Another class (Lev2) signature `a007d9e3754daef90ded300190903451` is used by 105 legitimate applications and 80 malware. Further examination shows that it is a class from the Google official library called AdMob [19].

For detail, all experimented malware samples are presented in [6], and they represent the known malware in our database. Readers who are interested in these malware samples can refer to [6]. The detection results is based on the cloud anti-virus engine using various detection engines (i.e., Kaspersky and Antiy(linux version)). Note that in the last column, R (G) represents repackaged (generic) malware family. For those malware families with less than five samples, we lump them as “others” in the table. Antiy (linux version) [20] is a commercial anti-virus product that we obtained from the company, and the product is known to run the same engine for detecting malware in smartphones. The rows are sorted in alphabetical order. Highlighted rows show the common malware families detected by both Kaspersky and Antiy. Others are uniquely detected by one anti-virus product. The penultimate row shows there are 1,295 common malware samples detected by these two anti-virus products. Hence, the number of unique malware samples is 2,148.

V. Zero-day Malware Detection

Here, we show a novel methodology in using DroidAnalytics to detect the zero-day repackaged malware. We analyze three zero-day malware families to illustrate the effectiveness of our system.

A. Zero-day Malware

Zero-day malware is a new malware that current commercial anti-virus systems cannot detect. Anti-virus software usually relies on signatures to identify malware. However, signature can only be generated when samples are obtained. It is always a challenge for anti-virus companies to detect the zero-day malware, then update their malware detection engines as quickly as possible.

In this paper, we define an application as a zero-day malware if it has malicious behavior and it cannot be detected by popular anti-virus software (e.g., Kaspersky, NOD32, Norton) using their latest signature database. As of November, 2012, we use DroidAnalytics and have successfully detected 342 zero-day repackaged malware in six different families: AisRs, Aseiei, AIProvider, G3app, GSmstracker and YFontmaster (please refer to Table IV for reference). In this paper, we use the name of the injected package (not the name of the repackaged applications) as the name of its malware family. Furthermore, all samples are scanned by Kaspersky, NOD32, Norton and Antiy using their latest database in November, 2012. We also uploaded these samples to the virustotal [21] for malware detection analysis. Note that none of the submitted samples was reported as a malware by these engines when we carried out our experiments.

Family Name	Number	Malicious Package Name
AisRs	87	com.ais.rs
Aseiei	64	com.aseiei
AIProvider	51	com.android.internal.provider
G3app	96	com.g3app
GSmstracker	10	com.gizmoquip.smstracker
YFontmaster	34	com.yy.fontmaster
All	342	

Table IV
ZERO-DAY REPACKAGED MALWARE SAMPLES

In [22], [23], authors reported that nearly 86.0% of all Android malware are actually repackaged versions of some legitimate applications. By camouflaging to some legitimate applications, repackaged malware can easily deceive users. Given the large percentage of repackaged malware, we explore the effectiveness of using DroidAnalytics to detect the *zero-day repackaged malware*.

B. Zero-Day Malware Detection Methodology

The process of detecting zero-day repackaged malware can be summarized by the following steps.

Step 1: We first construct a white list for common and legitimate classes. For example, we add all legitimate level 2 signatures, such as those in utility libraries (e.g., Json library) or advertisement libraries (e.g., Google Admob library, Airpush library) to the white list. All level 2 signatures in the white list will not be used to calculate the similarity score between two applications.

Step 2: We calculate the number of common API calls between two given applications in the database. This can be achieved by using the similarity score in Equation (1).

$$S_{app}(f_a, f_b) = S(f_a \cap f_b), \tag{1}$$

where f_a and f_b as two level 2 signature sets of two applications a and b respectively, $S(x)$ is a function to indicate the total number of API calls in the set x. The above similarity score between two repackaged malwares focus on the *common* repackaged API calls that correspond to the malicious logic, and ignore the effect of other API calls in these two applications.

Step 3: Assume we have N applications in the database, then we start with N clusters. The distance between two clusters is the similarity score we mentioned in Step 2. We select two applications which have the largest similarity score and combine them into one cluster. For this new cluster, we re-calculate the similarity score between this new cluster with other $N - 2$ clusters. The new similarity score is computed by averaging all similarity scores between all applications in two different clusters.

Step 4: Again, we combine two clusters which have the largest similarity score. We continue this step until the similarity score between any two clusters is less than a pre-defined threshold T (say T is 100).

Step 5: After we finish the clustering process, we will use anti-virus engines to scan all of these N applications. Each application may be classified as *legitimate* or *malicious*.

Step 6: If a cluster has more than n applications (say $n = 10$) and a small fraction f (say $f \leq 0.2$) is classified as malicious. This should be a *suspicious cluster* since it is very unlikely that more than n applications are similar (in terms of class functionality) in the real-world, and most of them are classified as benign. Hence, the similarity comes when some of these applications are repackaged. We then extract their common classes (using our level 2 signature) and examine these classes. Once we find any malicious logic in these common classes, we discover a zero-day repackaged malware family.

Experimental results in discovering zero-day repackaged malware: Let us present the results of using DroidAnalytics to discover three zero-day repackaged malware families.

- AisRs family: We discover 87 samples of AisRs family in our database. All the malware are repackaged from legitimate applications (e.g., *com.fengle.jumptiger*, *com.mine.videoplayer*) and all of them have a common malicious package named "`com.ais.rs`". This malware contains a number of botnet commands that can be remotely invoked. When the malware runs, it will first communicate with two remote servers. These two servers are camouflaged as software download websites. If any of these two servers is online, the malware will receive some commands like downloading other `.apk` files. They are not necessary malware, but they contain advertisement from the website, "http://push.aandroid.net". It is interesting to note that the address is "**a**android.net", not "android.net". Also, one of the many botnet commands is to save the user's application installation lists and system information to the .SQLite file, then upload this file to a remote server.

- AIProvider family: We discover 51 samples of AIProvider family in our database. All the malware are repackaged from legitimate applications (e.g., *jinho.eye_check*, *com.otiasj.androradio*) and all of them have a common malicious package named "com.android.internal.provider". There are several interesting characteristics of this malware. Firstly, the malicious package name is disguised as a system package name. Since DroidAnalytics does not detect malware based on the package name, so our system can easily discover this repackaged malware. Secondly, this malware uses DES to encrypt all SMS information (e.g., telephone numbers, SMS content) and store them in the DESUtils class. Thirdly, the malware will start a service called OperateService in the background when it receives "BOOT_COMPLETED" broadcast. This service will decrypt the SMS information in the DESUtils class and use this information to send SMS messages without any notification.

- G3app family: We discover 96 samples of G3app family in our database. All the malware are repackaged from legitimate applications (e.g., *com.openg3.virtua lIncomingCall*, *com.cs.android.wc3*) and all of them have a common malicious package named "com.g3app". There are several malicious behaviors in the G3app malware family. Firstly, the malware will frequently pop up notification on the status bar and entice users to select it. Secondly, the malware will inject trigger codes to every button of the legitimate application. If the user presses any button in the repackaged application or the notification in the status bar, the malware will download other applications from the remote server. We believe the hackers want to use repackaged malware to publicize their applications and use these advertisements for financial gain.

VI. Related Work

Before the rapid increase of Android malware in 2011, researchers focused on the permission and capability leaks of Android applications. E.g., David Barrera *et al.* [24] propose a methodology to explore and analyze permission-based models in Android. Stowaway [17] is a tool for detecting permission over privilege in Android, while ComDroid [25] is a tool which detects communication vulnerabilities. Woodpecker [26] analyzes each application on a smartphone to explore the reachability of a dangerous permission from a public, unguarded interface. William Enck *et al.* [27] propose a lightweight mobile phone application with a certification-based permission. In this paper, instead of malware detection, we focus on designing an analytic system that helps analysts to dissect, analyze and correlate Android malware with other Android applications in the database. We propose a novel signature system to identify malicious code segments and associate with other malware in the database. Our signature system is robust against obfuscation techniques that hackers may use.

In August 2010, Kaspersky reported the first SMS Trojan, known as FakePlayer in Android systems [28]. Since then, many malware and their variants have been discovered and mobile malware rapidly became a serious threat. Felt *et al.* study 18 Android malware in [29]. Enck *et al.* [30] carry out a study with 1,100 android applications but no malware was found. Zhou Yang *et al.* [31] study characterization and evolution of Android malware with 1,260 samples. However, they did not show how to systematically collect, analyze and correlate these samples.

Yajin Zhou *et al.* [32] were the first to present a systematic study for the detection of malicious applications on Android Markets. They successfully discover 211 malware. Their system, DroidRanger, needs malware samples to extract the footprint of each malware family before the known malware detection. For zero day malware, DroidRanger serves as a filtering system. After the filtering process, suspicious malware needs to be manually analyzed. DroidMOSS [33] is an Android system to detect repackaged applications using fuzzy hashing. As stated in [33], the system is not designed for general malware detection. Furthermore, the similarity score provided by DoridMoss is not helpful in malware analysis. In addition, obfuscation techniques can change the order of classes and methods execution, and this will introduce large deviation in the measure used in DroidMOSS. Michael Grace et al. develop RiskRanker [34] to analyze whether a particular application exhibits dangerous behavior. It uses class path as the malware family feature to detect more mutations. However, obfuscation can easily rearrange the opcode along an execution path. So using class path for malware feature is not effective under obfuscation attack. Our system overcomes these problems by using a novel signature algorithm to extract the malware feature at the opcode level so it captures the semantic meaning for signature generation.

For PC based malware, a lot of research work focus on the signature based malware detection. For example, authors in [15] discussed the limitation of using signature to detect malware. In [35], authors described the obfuscation techniques to hide malware. Authors in [36] presented an automatic system to mine specifications of malicious behavior in malware families. Paolo Milani Comparetti *et al.* [37] proposed a solution to determine malicious functionalities of malware. However, mobile malware has different features as compared with PC based malware. It is difficult to transform a PC based malware detection solution for mobile devices. For example, [3] reported that many anti-virus products have poor performance in detecting Android malware mutations, although these products performed reasonably well for PC based malware. Repackaging is another characteristic of android malware. Authors in [33] showed there are many repackaged applications in Android third party markets and significant number of these applications is malware. Based on the above studies, it is clear that a more sophisticated methodology to detect and analyze Android malware is needed.

VII. Conclusion

We present DroidAnalytics, an Android malware analytic system which can automatically collect malware, generate signatures for applications, identify malicious code segment (even at the opcode level), and at the same time, associate the malware under study with various malware and applications in the database. Our signature methodology provides significant advantages over traditional cryptographic hash like MD5-based signature. We show how to use DroidAnalytics to quickly retrieve, associate and reveal malicious logics. Using the permission recursion technique and class association, we show how to retrieve the permissions of methods, classes and application (rather than basic package information), and associate all applications in the opcode level. Using DroidAnalytics, one can easily discover repackaged applications via the similarity score. Last but not least, we have used DroidAnalytics to detect 2,494 malware samples from 102 families, with 342 zero-day malware samples from six different families. We have conducted extensive experiments to demonstrate the analytic and malware detection capabilities of DroidAnalytics.

References

[1] McAfee, "McAfee Threats Report: Fourth Quarter 2011," Tech. Rep., 2012.

[2] Contagio Mobile. [Online]. Available: http://contagiominidump.blogspot.com/

[3] M. Zheng, P. P. C. Lee, and J. C. S. Lui, "ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-Virus Systems," in *Proceedings of the 9th Conference on DIMVA*, 2012.

[4] Scrapy. [Online]. Available: http://www.scrapy.org

[5] A. P. Felt, H. Wang, A. Moshchuk, S. Hanna, and E. Chin, "Permission Re-Delegation: Attacks and Defenses," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[6] M. Zheng, M. Sun, and J. C. S. Lui, "Droidanalytics: A signature based analytic system to collect, extract, analyze and associate android malware," Tech. Rep. [Online]. Available: http://arxiv.org/abs/1302.7212

[7] G. McCluskey. (1998) Using Java Reflection. [Online]. Available: http://docs.oracle.com/javase/tutorial/reflect/index.html

[8] Google Inc. Google Play Store. [Online]. Available: https://play.google.com/store

[9] AndroidPIT. [Online]. Available: http://www.androidpit.ru

[10] AppChina. [Online]. Available: http://www.appchina.com/

[11] Android App Download Website: SouApp. [Online]. Available: http://www.souapp.com/

[12] Mobile Malware Forum. [Online]. Available: https://groups.google.com/forum/?fromgroups=#!forum/mobilemalware

[13] Android Security Discuss Forum. [Online]. Available: https://groups.google.com/forum/?fromgroups=#!forum/android-security-discuss

[14] Apktool. [Online]. Available: http://code.google.com/p/android-apktool/

[15] A. Moser, C. Kruegel, and E. Kirda, "Limits of Static Analysis for Malware Detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.

[16] M. Zheng, P. P. C. Lee, and J. C. S. Lui. ADAM. [Online]. Available: http://ansrlab.cse.cuhk.edu.hk/software/adam/

[17] 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.

[18] DOMOB. [Online]. Available: http://www.domob.cn

[19] AdMob. [Online]. Available: http://www.google.com/ads/admob

[20] Antiy Labs. [Online]. Available: http://www.antiy.net/en/index.html

[21] Virustotal. [Online]. Available: http://www.virustotal.com

[22] Abhi, "Android Malware Injected through Repackaging of Apps," Tech. Rep., 2012.

[23] G. Sims, "Android Malware Genome Project shows that 86% of all malware delivered via repackaging of legitimate apps," Tech. Rep., 2012.

[24] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A Methodology for Empirical Analysis of Permission-based Security Models and its Application to Android," in *Proceedings of the 17th ACM conference on Computer and communications security*, 2010.

[25] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing Inter-application Communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[26] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[27] W. Enck, M. Ongtang, and P. McDaniel, "On Lightweight Mobile Phone Application Certification," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[28] C. A. Castillo, "Android Malware Past, Present, and Future," *White Paper of McAfee Mobile Security Working Group*, 2011.

[29] 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.

[30] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX conference on Security*, 2011.

[31] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, 2012.

[32] 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 *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[33] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*, 2012.

[34] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012.

[35] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The Hidden Malware," *IEEE Security and Privacy*, 2011.

[36] M. Christodorescu, C. Kruegel, and S. Jha, "Mining Specifications of Malicious Behavior," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007.

[37] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, and S. Zanero, "Identifying Dormant Functionality in Malware Programs," in *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.