Android Malware Forensics: Reconstruction of Malicious Events

Juanru Li, Dawu Gu, Yuhao Luo
Dept. of Computer Science and Engineering
Shanghai Jiao Tong University
Shanghai, China
Email: dwgu at sjtu.edu.cn

***Abstract*—Smart mobile devices have been widely used and the contained sensitive information is endangered by malwares. The malicious events caused by malwares are crucial evidences for digital forensic analysis, and the main task of mobile forensic analysis is to reconstruct these events. However, the reconstruction heavily relies on the code analysis of the malware. The difficulties and challenges include how to quickly identify the suspicious programs, how to defeat the anti-forensics tricks of malicious code, and how to deduce the malicious behaviors according to the code. To address this issue, we propose systematic procedures of analyzing typical malware behaviors on the popular mobile operating system Android. Based on the procedures we discuss the deduction of Android malicious events. We also give a real malware forensic case as a reference.**

***Index Terms*—forensic analysis, Android, malware, reverse engineering.**

I. Introduction

With the fast evolvement of mobile OS such as Android[1] and iOS, and the growing processing capability of mobile hardware, the number of smart mobile devices grows exponentially. Mobile devices are nowadays widely used to deal with sensitive personal affairs, and are becoming an attractive platform for cybercriminals. A huge number of malwares are developed to threaten the data privacy and system security of smart mobile devices, and they bring new challenges to forensic analysts. A modern digital forensic analyst should know these threats and be able to employ forensic analysis against mobile malware.

Most of the forensic analysis on mobile devices focus on the data acquisition process[2][3]. However, the scope of mobile malware forensics extends from simple information retrieval to a series of events reconstruction. In other words, the requirement is to deduce the happened malicious events via the data. Digital detectives, like their legendary colleague *Sherlock Homles*, are challenged to reveal truth from the bytes. Typical scenario of mobile malware forensics often goes like this, She acquires a mobile device compromised by malwares and is asked to recover the malicious events caused by malwares (A harsher case is that only the memory dump of the device is given). She has the privilege to statically extract the data from the device, or even launches the device (without changing sensitive data) to employ some dynamic analysis and to observe the exceptions.

The mobile malware forensics often involves four aspects, *identification of suspicious programs, defeating the anti-forensics code, extracting malicious code from malwares* and *malicious functions deduction*. Traditional forensics discuss the process of collecting static data as digital evidence[4]. For malwares, even with the collected static program code, the malicious functions, which are very important digital evidences, is often ignored to be reconstructed. Program codes are raw information which can't bring the recovery of malicious events without deduction. The reconstruction of malicious events involves connecting the relationship between programs, operating system, hardware and I/O data. Tiny details may be main obstacles of malicious events reconstruction. Modern mobile malwares are designed towards certain platform. The complexity of both hardware (different CPU architectures, file systems) and software (new mobile operating system) challenge the inexperienced analysts. The architecture and design pattern of mobile applications differ widely from those of common applications on personal computers. Thus to deploy mobile program analysis is even harder for lacking of well-developed analyzing tools on mobile platform.

The purpose of this paper is to present a systematic process of Android malware forensic analysis, focusing on the deduction and reconstruction of malicious events. In detail we suggest three main steps to reconstruct the events. First, the identification of suspicious programs on Android platform. Second, how to defeat the Android based anti-forensics codes. Third, the recognition of the most typical malicious behaviors on Android that forensics should concern, and the effective deduction of criminal events. We give a thorough discussion on essential points of these steps. In order to describe the details, a particular Android mobile malware forensic analysis process is also given.

This paper makes the following contributions:

- We propose a systematic procedure for Android malware forensic analysis and malicious events reconstruction.
- We discuss in detail the code analysis of Android malware, which is ignored by most forensic research works.
- A real forensic analysis case is given and we suggest how to combine existing tools and techniques to help analysis.

The paper is organized as follows: Section II briefly introduces malwares on Android OS. Section III proposes the main principles and methods of malware identification on Android.

Section IV describes the common anti-forensics technique and proposes the countermeasure. Section V describes the deduction of malicious events via program code, how to take advantage of the relationship between specific Android developing feature and corresponding malicious function, and paying particular attention to typical malicious behaviors of Android Malwares. Section VI gives a concrete example of mobile malware forensic analysis. The conclusions are made in Section VII.

II. A brief introduction to Android malware

Most of the malwares on Android OS are developed using JAVA programming language and are executed on Dalvik VM engine[5] of the system. Although Android itself is a Linux based system, the best way of malware invasion is via normal application installation. Thus to analyze malwares on Android OS, the analyst should first understand the format of the Dalvik VM based program. The Dalvik Based Android applications are released and stored in the device with the *APK* format[6]. An application is first compiled and is then archived into one single *APK* file with all of its parts, including codes and assets. The *APK* file is actually an application in the form of a *ZIP* archive with codes, resources, assets, certificates and manifest file. The inner folders and files structure of this archive conform to the *JAR* file format specification. After the installation, the *APK* file is copied to a specific location in the system. For system applications, the location is typically */system/app* and for user-installed applications the location is */data/app*.

From the forensic analyst's point of view, an *APK* file contains three parts of abstract information: *signature*, *bytecodes* and *resources*.

A. Signature

The signature contains the message digest of the *APK* file. Since any modification to the *APK* file will change the message digest of the signature, one could quickly identify if an application is corrupted by checking the signature. Analyst could also collect signatures of malwares to find out malwares quickly.

B. Bytecodes

The executable part of the application, the *classes.dex* file in the archive, contains all compiled classes of the program in the form of bytecodes. For Android programs, the original JAVA bytecodes are converted to the instruction set used by the Dalvik VM, which is a register-based VM while JVM is stack-based.

C. Resources

Resources is the non-executable part of the application, it contains all additional data required by the application. Most resources in an application are user interface components, such as *bitmaps*, *menus*, *layouts*, *widgets*. In most cases, the malicious part of the malware runs in background and does not have any user interfaces. So these UI resources are seldom concerned. However, the resource file, *AndroidManifest.xml*, is important for it indicates crucial forensic information of an application. The *AndroidManifest.xml* file is encoded into binary format in the *APK* file. It contains the permission request of an application. The most important forensic information are *permissions* and *components*.

1) Permissions: In order to access some protected APIs of Android, the application will declare the permission request in *AndroidManifest.xml*, such as the permissions to read *message*, *contacts*, etc. Permission request is a very important clue to reveal malicious functions[7]. For instance, a normal application, such as calculator, declares a *READ_CONTACTS* permission, it can be very suspicious because a calculator should never need information about *contacts*. This character is unique for Android applications and is useful for analysis.

2) Components: Android applications are formed by components. The components of the application are divided into four kinds – *activities*, *services*, *broadcast receivers* and *content providers*. A malware who runs in background often has a *service* component and a *receiver* component in order to receive the boot *Intent* on system booting. By checking components and their received *intents*, analyst may have a brief view of the potential behavior of an application.

III. Identification of suspicious applications

In a typical smart mobile device there are as much as hundreds of applications. Malwares only occupy a few part of them and Most of the others are benign. The first step of forensic analysis is to identify the malicious programs from the benign ones. Although many research works and tools are claimed to support malware detection[8], there're still some unsolved problems for forensics. In one way, automatic tools need samples to generate malware database. The rapid evolvement of malware makes automatic detection tools difficult to follow. Moreover, some malwares are designed for attacking specific devices and yet are hard to be collected beforehand. In another way, forensics not only needs to find the suspicious programs, but also requires code analysis and events reconstruction. Thus manual check is helpful for later in-depth analysis, and manual methods are essential for forensic analyst to ensure the identification.

To identify malicious programs, one important conclusion is that malwares are always connected with some unusual features. These features indicate the potential suspicions. We suggest checking the following features to efficiently and effectively identify suspicious applications.

A. Message Digest

For excluding benign applications from affected ones, the message digest is a useful cryptographic feature. Often a trustful application is released via online market and the market provides its message digest. A database for normal applications can be built by collecting message digest information from online markets. Then the analyst simply checks an application's message digest and if the message digest of the checked application can't be found in database, it is

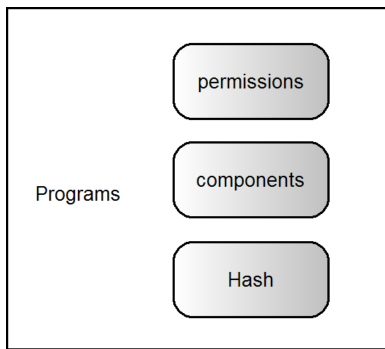

Fig. 1. Suspicious Features

possible that this application is malign. However, only with the message digest it is not cautious to determine the malicious applications. A more in-depth analysis should be employed to fulfil the identification.

B. Permission requirement

The permission requirement is a unique character for Android programs. Due to the design philosophy of the Android OS, the application only needs to apply for permissions when being installed and persistently own these permissions without repeatedly requesting. Users may ignored the initial request, and a common malware pretending to be an unharmful application with faked normal functions will ask for a set of permissions such as *SMS* and *Contacts* database access, even the faked functions of the application need not these permissions at all. Suspicious permission requirement is the leading clue to confirm an Android malware[9][10]. Most of the Malwares declare a list of high-privilege permissions to fulfil malicious functions. According to the *AndroidManifest.xml* file of an APK file, we can find out the permissions requested by the application and filter out the suspicious requests.

C. Components

At the very abstract level, Android application is formed by components. The structure of components can be used to judge the program's characters. As mentioned in Sec II, *service* component and *receiver* component are sensitive weapons for malwares. So, from the examination of components and their received intents, analyst could have a brief view of the potential malicious function of an application. and suspicious applications are to be distinguished from the normal ones.

IV. Defeat Anti-forensics codes

In this section we introduce three common anti-forensics techniques and discuss how to deal with them.

A. Anti-forensics techniques

Events could be deduced from the code. However, malware developers always try to stop the deduction or make it hard. Before code analysis, one important thing is to clean the barrier – anti-forensics codes. Anti-forensics codes are common inside malwares of commodity personal computers. For instance, many malwares detect the execution environment to check whether it is executed inside a virtual machine. Android Malwares inherit the property to inconvenience the forensic analysis.

- *obfuscation*. The obfuscation techniques of Android malware is as much the same as JAVA obfuscation[11], because the developing programming languages are similar. A very typical case is that in an obfuscated program all of the *packages*, *classes*, *methods*, *fields* are renamed to single alphabet such as *a, b, c.a(), d, e.b, f.a, g.b()*. So that analyst is hard to distinguish different parts of the code yet is difficult for her to understand the functionalities.
- *strings encryption*. For an experienced reverse engineer, strings in a program are valuable information sources. Many malwares use string encryption to avoid plaintext detection. Constant strings in malware are encrypted with symmetric algorithms such as *DES* and the *AES* and the key is fixed (dynamic key is seldom used because no matter how complex the key is, it will finally be used to decrypt the ciphertext). The encryption makes static analysis hard. However, if the analyst has the capability of dynamic execution she may manually extract key and decrypt the ciphertext, thus the information is still available for retrieving.
- *environment verification*. Some of the mobile malwares are designed to attack certain types of mobile devices. Specific symbols like Android system properties (from *android.os.BUILD*) are often verified to make sure the malware is not executed in an emulator or other types of devices. And the subscriber ID (*IMSI*) is used to make sure the malware is running on a certain device with the special *IMSI*. If verification fails, the malicious code will stop executing, and the analyzers could not simply reproduce the malicious behavior by emulation or using any improper devices. This anti-forensic technique lets malware deceive dynamic black-box analysis.

B. Countermeasures

We suggest some countermeasures to the anti-forensics techniques mentioned above.

1) decompilation and deobfuscation: For an Android application, the high level JAVA-like source code is much easier to read and to be understood than the bytecode. However, the State of Art decompilation tools cannot decompile programs perfectly. The decompiled source code typically contains mistakes or code absences. What's more, in many cases the bytecode of malware is obfuscated, which makes decompilation more difficult and inaccurate. Meanwhile, the bytecode is always correct and accurate although it is much more difficult to be analyzed. So analyst should utilize both bytecode and decompiled source code, and take both codes into analysis to compensate the shortcomings of each other.

Three main steps are suggested to employ decompilation and deobfuscation. First, the analyst could use *apktool*[12] to extract the bytecode (with *.dex* format). Then, the combination of *dex2jar*[13] and *jd-gui*[14] are helpful to decompile the

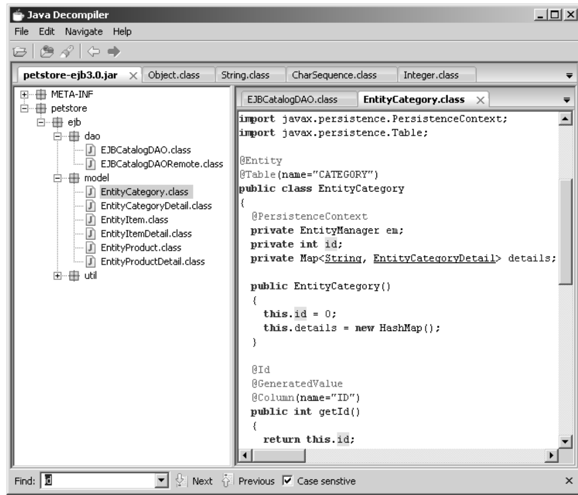

Fig. 2. JD-GUI decompiler

bytecode file to JAVA source code. The decompiled JAVA source code may contain huge number of errors. The following measures are possible options for code fixing.

- *removing empty classes*
- *renaming*
- *decompile errors correction*
- *control flow error correction*
- *name conflict correction*
- *missed information fixing*

2) strings decryption: Strings are important information sources and most constant strings (e.g. remote server URL) in malware are encrypted. Often a decryption process is required to extract these strings. The whole decryption process involves encryption algorithm recognition, secret key extraction and string decryption. One convenient aspect is that many malwares use system cryptographic APIs to deal with encryption and decryption. Analyst could filter out these situations and quick identify the key.

3) program patching: As mentioned above, to deceive dynamic analysis, system properties and the subscriber ID are often verified by the malware. In order to employ dynamic analysis, analyst could automatically search for these features and manually patch the code to avoid these verifications.

V. Malicious events deduction

The core part of mobile malware forensics is to reconstruct the malicious events via program code and additional information such as network flow. But in most cases the only form of malware provided is binary program. In order to understand the logic of the program, a reverse code analysis is essential. Although there is not a standard procedure for reverse code analysis, on Android some typical behaviors may be the key that helps analyst unlock the puzzle and understand the crime. These typical behaviors are always related to malicious code with obvious patterns. So analyst can follow the patterns to locate and then find the malicious behaviors, and finally combines the behaviors to deduce the events.

A. Specific suspicious behaviors on Android

The Android malwares steal private information such as *SMS* and *Contacts*, and automatically send them to remote servers. In detail, a malicious program may pretend to be a normal financial application while accessing users' personal information (*SMS*, *Contacts*, *ID*, etc.) with background service. The background service then waits for certain commands from particular remote server and sends the private information via self-defined protocols. Related suspicious functions that essential to the malicious behavior are listed below.

- *service core loop*. Most of the malwares contain a service that supports continually execution. On Android OS, this is often implemented using a *service* component.
- *self-defined communication protocol*. A malware often contains a self-defined protocol to communicate with a certain remote server with its own "language". Inside the malware some modules handle the communication between the client and the command server. Malwares often pack and encrypt the sent information, decrypt and parse data returned by the server. On Android OS, this function is always related to the permission of network access such as *android.permission.INTERNET* and *android.permission.ACCESS_NETWORK_STATE*.
- *cryptographic utilities*. The cryptographic utilities from system libraries support encryption operation of the malware, such as generating message digest of device information, decrypting encrypted strings, exchanging key with server and making encrypted communication with the server.
- *sensitive data access*. Sensitive data access is the core function of the malware from the point of privacy leakage. It needs high privileges to achieve the goal. First, data access permissions such as *android.permission.READ_SMS*, *android.permission.READ_CONTACTS* are required. Then, specific protected APIs and content providers are used to visit the database. An application with sensitive data access permission request is highly suspicious.

B. Taking Code analysis to reconstruct crime

It is hard to extract the malicious behaviors of malwares in the view of top level. Android malwares are written using JAVA programming language, and the bytecode of the malware contains all logic functions. In mobile malware forensic analysis, the direct evidence of malicious events is from the malicious code itself. A malware sample may be acquired after the crime. The criminal events is unknown for analyst. Only the code related to malicious behaviors helps analyst fast locating and analyzing the malicious event first. That is to say, through the reverse code analysis that aims at extracting program fragments first, analyst could then combine simple functions into a high-level abstract events.

The reconstructed events may include following information, the work flow of malicious code, sensitive information that the malware accessed, the encryption algorithms, and the

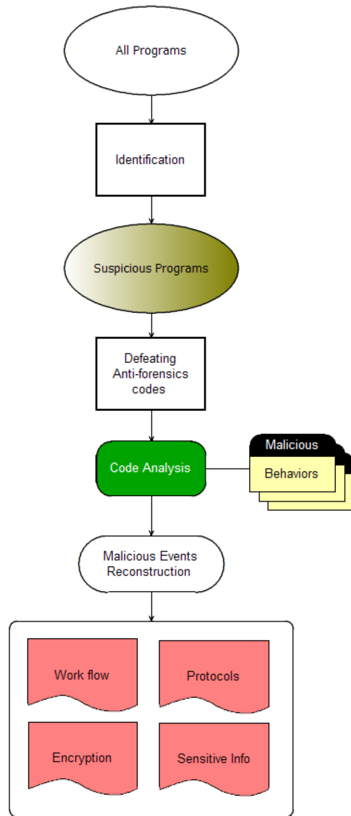

Fig. 3. Malicious Events Reconstruction - work flow

details of malware's communication protocols. The task of reconstruction and combination requires not only the mining of function inside codes, but also rearrangement of these functions into a correct order. Android provides a *logcat* mechanism to capture high-level operations such as the system API calls and services starting/stopping log. If allowed, analyst should try to reappear the execution of malware and record the occurred operations, and then draw the picture of the events.

Fig 3 shows a complete work flow of events reconstruction procedure.

VI. Case study

In this section we provide a complete forensic analyzing process to show some details of mobile malware analysis. The analyzed malware sample is from the *honeynet* Forensic Challenge 9[15]. Notice that we ignore the file recovery process for it is not directly related to this paper's topic. The interested reader may refer[16]. The process of forensic analysis could be divided into four parts and each part is introduced below.

A. Background

The challenge offers the exploration of a real smart phone compromised by mobile malware, based on Android, after a security incident. Analyst will have to analyze the image of a portion of the file system, extract all that may look suspicious, analyze the threat and finally give conclusion. The required knowledge extends from File System recovery to Malware reverse-engineering and PCAP analysis.

B. identification of suspicious program

There're totally ten applications and two tmp files contained in the provided corrupted memory dump. Among them, seven files' message digest can be found online and are considered as normal, trusted packages.

- *com.adobe.reader-1.apk*
- *com.google.android.stardroid-1.apk*
- *com.rovio.angrybirds-1.apk*
- *com.android.vending-1.apk*
- *com.google.android.apps.maps-1.apk*
- *com.google.earth-1.apk*
- *com.opera.browser-1.apk*

For the rest three packages, although their SHA1 and MD5 are not found online, we further examined the permission and component and decompiled these packages, found nothing suspicious for *com.google.android.apps.finance-1.apk* and *net.xelnaga.exchanger-1.apk*. We noticed that the only left application, *app/com.fc9.currencyguide-1.apk* is exactly the same as the tmp files *app/vmdl34052.tmp* and *lgdrm/TRYSYNC*, which means this application was possibly active when the memory was dumped.

Then, we checked the requested permission of this application and found the following permissions are requested.

- *android.permission.INTERNET*
- *android.permission.READ_PHONE_STATE*
- *android.permission.ACCESS_WIFI_STATE*
- *android.permission.WAKE_LOCK*
- *android.permission.ACCESS_NETWORK_STATE*
- *android.permission.RECEIVE_BOOT_COMPLETED*
- *android.permission.CAMERA*
- *android.permission.VIBRATE*
- *android.permission.ACCESS_COARSE_LOCATION*
- *android.permission.ACCESS_FINE_LOCATION*
- *android.permission.CALL_PHONE*
- *android.permission.SEND_SMS*
- *android.permission.READ_CONTACTS*
- *android.permission.RECEIVE_SMS*
- *android.permission.READ_SMS*

The application(see fig 4) however performed as a normal currency calculation tools when executed. The unusual number and type of unnecessary requested permissions makes it suspicious.

We then focused on this suspicious application *com.fc9.currencyguide-1.apk* to employ analysis in depth.

C. Anti anti-forensics

This malware uses all three anti-forensics techniques we mentioned above to interfere forensic analysis. We only employed *code error fixing* and *strings decryption* to get a neat version of decompiled code for static analysis.

Fig. 4. The suspicious application

1) Code error fixing and Refactoring: The raw code from the decompiler contains lots of errors. To fix these errors, we analyzed the bytecode, and rebuilt the source code. Then, because all malicious parts of the code are obfuscated by name renaming(e.g., one of the class is *com.fc9.currencyguide.daemon.g.a.a*), we need to do code Refactoring. The procedure of refactoring all source codes was a Depth-First Search. The atomic functions were refactored first then the complex ones. What's more, when looking into the decompiled code, we found some empty JAVA classes without fields or methods definition. Some of these situations are due to the decompiler's processing capability and the correct code should be manual added after further examination to the corresponding bytecode. And We confirmed that other situations are really empty. These classes were probably intended to add into the code by the obfuscator.

2) strings decryption: The most important data in this malware is string encrypted with DES, including the server address, command names, etc. We extracted the DES key from the code (0x63B252F6FAF4167F) and decrypted all encrypted strings with the key. For instance, one important string is the address of the command server, *http://faeacdeadbeefada.zonbi.org:443*.

D. Code analysis

1) Code entry: From the *AndroidManifest.xml* of *com.fc9.currencyguide-1.apk*, we found six components of the application,

- *Main_Activity*
- *Converter_Service*
- *PrefMenu_Activity*
- *com.fc9.currencyguide.daemon.fc9*
- *com.fc9.currencyguide.daemon.CCcomService*
- *com.fc9.currencyguide.daemon.BootReceiver*

By examining the code entry of each component, we found that the first three components, *Main_Activity, Converter_Service, PrefMenu_Activity*, fulfils the normal currency converter function that the application pretends, while the last three components are malicious. The component *com.fc9.currencyguide.daemon.fc9* is an activity started on application launching, *com.fc9.currencyguide.daemon.BootReceiver* is a receiver component who responds to the *BOOT_COMPLETED* intent. They may start the service *com.fc9.currencyguide.daemon.CCcomService*, which is the main malicious code entry. The design indicates the malicious service will run automatically when the application launched or system boot completed. And it contains a state machine and an infinite loop so that it will always active in background.

E. evidences and malicious event rebuild

According to the code analysis result, we rearranged the sequence of each functions and formed a complete malicious communication process. The process contains four parts.

1) key exchange: The first step of the communication between the malware and the server is a self-defined *diffie-hellman* key exchanging to establish a secret key. The typical *Diffie-hellman* key exchange algorithm is used in the negotiation, and then DES is used in encrypted communication. According to the source code and the captured network traffic record, we found the DES key is 0xc4c9973a45c7007d.

2) encrypted private information sending: The communication between malware and remote server is based on *HTTP* protocol. The following private information are sent via encrypted communication.

- *device infomation* The device ID (IMEI), subscriber ID (IMSI), network operator name(alphabetic name), network operator(numeric name), network ISO country code.
- *personal information* SMS (address and body for each sms, contained password)
- *contacts* contact id and display name
- *com.fc9.currencyguide.daemon.fc9*
- *com.fc9.currencyguide.daemon.CCcomService*
- *com.fc9.currencyguide.daemon.BootReceiver*

3) server command receiving: The malware requests command from server every 15 seconds, and executes the command. Some commands involve extra communication while executing. For commands "*getsms*", "*getcontacts*", the malware will send *SMS* or *Contacts* information to server when executing.

4) The "smsspy" communication: A very special command from the server is the "*smsspy*" command. When receiving, the malware will change the malicious mode into an "*smsspy*" mode and send any *SMS* to server whenever an *SMS* is coming from mobile network.

Finally, the reconstructed malicious event can be represented as fig 5. For more details, please refer to[16].

VII. Conclusion

In this paper we discuss the problem of Android mobile malware forensic analysis. The core task of mobile malware forensic analysis is to reconstruct the malicious events according to malware. We propose some systematic procedures for Android malware forensics and discuss the details. In addition,

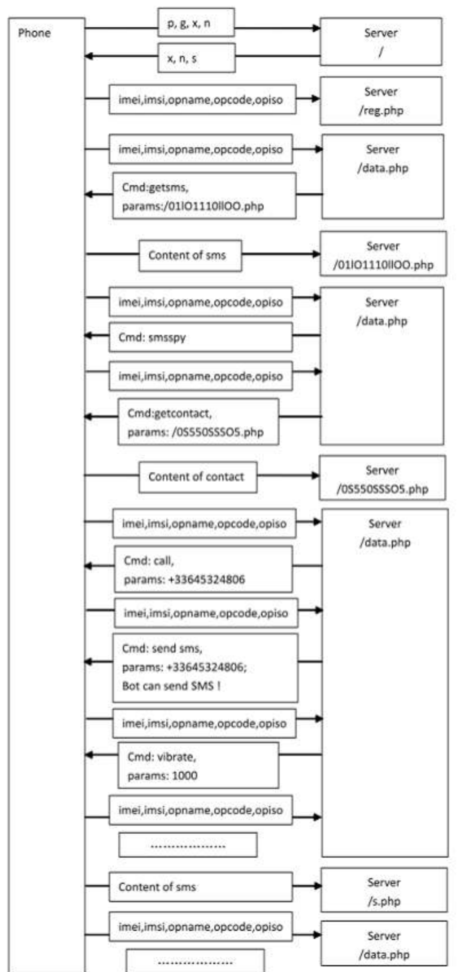

Fig. 5. The reconstructed malicious events

an example of malware forensic analysis is given to help further research.

ACKNOWLEDGMENT

The authors would like to thank SafeNet Inc. for the support.

REFERENCES

[1] Google, "Android," *URL http://www.android.com/accessed March*, 2012.

[2] J. Lessard and G. Kessler, "Android forensics: Simplifying cell phone examinations," *Small Scale Digital Device Forensics Journal*, 2010.

[3] Google, "Android forensics," *URL http://code.google.com/p/android-forensics/accessed March*, 2012.

[4] A. Hoog, *Android forensics*. Syngress., 2009.

[5] Google, "Code and documentation from android's vm team." *URL http://code.google.com/p/dalvik/accessed March*, 2012.

[6] Ophonesdn, "The structure of android package (apk) files." *URL http://en.ophonesdn.com/article/show/354/accessed March*, 2012.

[7] A. 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*. ACM, 2011, pp. 627–638.

[8] L. M. Security, "Android security for mobile." *URL https://www.mylookout.com/accessed March*, 2012.

[9] F. Di Cerbo, A. Girardello, F. Michahelles, and S. Voronkova, "Detection of malicious applications on android os," *Computational Forensics*, pp. 138–149, 2011.

[10] 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 and Distributed System Security Symposium*, 2012.

[11] D. Low, "Protecting java code via code obfuscation," *Crossroads*, vol. 4, no. 3, pp. 21–23, 1998.

[12] Brut.alll, "Android-apktool." *URL http://code.google.com/p/android-apktool/accessed March*, 2012.

[13] pxb1988, "dex2jar." *URL http://code.google.com/p/dex2jar/accessed March*, 2012.

[14] E. Dupuy, "Jd-gui: Yet another fast java decompiler." *URL http://java.decompiler.free.fr/?q=jdgui/accessed March*, 2012.

[15] Honeynet, "Forensic challenge 9 - "mobile malware"," *URL http://www.honeynet.org/node/751/accessed March*, 2012.

[16] Y. W. Luo YH, Li JR, "Submission of the honeynet forensic challenge 9." *URL http://www.honeynet.org/files/1317348062 _lyh62771gmail.com_ForensicChallenge2011-Challenge9-Submission.doc/accessed March*, 2012.