MASTER

# Detecting repackaged android apps using server-side analysis

Hu, H.

*Award date:*
2016

Technische Universiteit **Eindhoven** University of Technology

Department of Mathematics and Computer Science
Information security technology

MASTER THESIS

# DETECTING REPACKAGED ANDROID APPS USING SERVER-SIDE ANALYSIS

HAOSHI HU

Graduation supervisor:

prof. dr. Milan Petkovic (TU/e)

Daily supervisors:

ir. Paul Koster (Philips)

ir. Peter van Liesdonk (Philips)

Extended committee member:

dr. ir. Richard Verhoeven (TU/e)

Eindhoven, August 2016

# Abstract

A repackaged app is an original app that has been modified by an attacker to have some additional purpose. The attackers can get the app’s source code using reverse engineering, and then insert malicious code before repackaging it.

In this thesis, we focus on software based server-side verification to detect repackaged Android apps. Based on the background of reverse engineering techniques and current preventing techniques, we propose a two-phase model by improving existing detection schemes and adding an analysis phase. This analysis enable the server-only analysis of an app’s behavior.

In the detection phase, we improve an attestation scheme by using hidden channel and dynamic code injection techniques in order to detect known attacks, especially dummy folder attacks.

In the analysis phase, we design and implement an experiment to find an feasible method to distinguish repackaged applications and original applications. It is based on calculating distance between HTTP traffics sets.

# Contents

# List of figures

# List of tables

# Chapter 1 Introduction

## 1.1 Background

In recent years, smartphones and convenient mobile apps are becoming an integrated part of everyday life for most of people. According to the statistical analysis from statista.com [1], the number of smartphones is forecast to reach 2.08 billion in the end of 2016. The rich and colorful apps in smartphones extend people's life by empowering the users to socialize, entertain, surf online and make life convenient. In the meanwhile, smartphones have also become attractive target of attackers.

With the growth of smartphone features, attackers can get more information from users including bank account information, GPS data and photo records. Since the value of smartphone data is becoming more and more valuable, the malware is also growing rapidly. Since the first Android malware was detected in August 2010, over 300 malware families and more than 650,000 individual pieces of Android malware are recorded [2]. With the growth of malware, they are getting smarter to avoid and counter detection methods.

One particular threat facing mobile apps especially on the Android platform, is from repackaged apps. A repackaged app is an original app that has been modified by an attacker to have some additional purpose. That purpose can be either good or bad. According to the research result [3], 86% of investigated android malware are repackaged apps. The JAVA language used for Android apps makes it easy to apply reverse engineering. The attacker can get the app's source code using reverse engineering, and then insert malicious code before repackaging it. Since the attacker can inject any code inside the source code, repackaged apps could be a threat for both customers and app developers depending on the purpose of attackers.

## 1.2 Problem overview

Since repackaged apps can bring tremendous risks to the entire mobile ecosystem, several countermeasures for detecting and preventing the use of repackaged apps are proposed in recent years. We can divide the defense methods into two groups: user-side verification and server-side verification by verification groups. In this thesis, we will mainly focus on software based server-side verification of repackaged apps.

Considering the user-side verification, we assume that users are honest. The main purpose of verification is to help users determine the authenticity of downloaded apps. Similarly, the main purpose of sever-side verification is to help server determine the authenticity of communicating apps. However, the server does not know the users are honest or not in the case.  If the attacker bypass the server-side verification successfully, the technique can be used for game cheating, ICS(Industrial Control Systems) intruding or companies competing.

In this thesis, we will mainly focus on software based server-side verification of repackaged apps. Using reverse-engineering, an advanced attacker can get any secret keys embedded in the app, or modify source codes to bypass the authentication module of server. Thus, the common challenge-response attestation method could be bypassed by the attacker easily. Based on this background, [Jihwan Jeong] [4] proposed an unpredictable attestation method to solve such problems in 2014. However, their solutions can be easily circumvented: if an attacker includes the complete code of the original application (e.g. in a dummy folder), no matter what attestation module the server sends, he can calculate the correct response value by executing the attestation module on the dummy folder.  Thus, an advanced server-side attestation scheme is required to detect repackaged apps. In chapters 3 and 4 we will study such methods and propose an improvement.

Since attackers are getting smarter and more skillful nowadays, a new attack can be proposed anytime. Thus, it would be good for the server to have an ability of identifying unknown attacks. In Chapter 5, we propose and implement a method to detect varietal attacks using HTTP traffic analysis. HTTP traffic analysis turns out to be a good option to identify unknown attacks and distinguish repackaged apps with original apps.

## 1.3 Project tasks

One of our challenge in this project is to analyze the current state of the art in server-side attestation schemes for detecting repackaged apps and propose an advanced server-side attestation module which is solid enough to prevent current repackaging attacks.

Apart from the attestation module, we need to build up an HTTP traffic analysis module behind attestation module in order to double-check the application and identify unknown attacks.

## 1.4 Structure of the thesis

In chapter 2, we introduce the current techniques related unofficial apps.

In chapter 3, we summarize attack models used in current techniques and propose a new two-phase model.

In chapter 4, we propose an improved attestation scheme based on the knowledge of covert channel and dynamic code injection techniques. We also design a proper covert channel and method to implement it in this chapter.

In chapter 5, we propose an efficient HTTP traffic analysis method to distinguish repackaged apps that works behind attestation scheme in order to double check the application. We propose an efficient analysis method, implemented it and tested it on a variety of knows repackaged apps.

In chapter 6,  We present our conclusion and suggest directions of future works.

# Chapter 2 Current techniques

In this chapter, we introduce several related techniques to show that how the repackaged applications generated and what kind of techniques are currently used to detect and prevent repackaged applications.

## 2.1 Reverse engineering

Reverse engineer is the processes of extracting information from the system. The process often involves disassembling something and analyzing its components and workings in detail.
As for the Android apps, the reverse engineering is done primarily through decompilation of the DEX file, which can be decompiled into either Java code or smali code[4], depending on the technique used [5].

In order to obtain the Java code, several tools can be used to covert the Dalvik VM bytecode into JVM bytecode such as undx [6] and dex2jar [7]. Then, a Java decompiler can be used to recover the Java code. The Java class files have to undergo optimization when they are converted by the dx converter during compilation for use with the Dalvik VM. However, the decompilation result and the actual source code do not closely match although we can obtain Java code which is written in a high-level language.

Obtaining the smali code is another option which we can use. It involves getting the code in lower level language. It will be in the form of Dalvik VM assembly, which is mapped one-to-one with Dalvik VM bytecode. The analysis time will be longer since it is a low level language, but the development environment does not have to be set up to match that used for the original source code because the DEX file is created by directly converting the Dalvik VM instructions into Dalvik VM bytecode during repackaging. Thus, obtaining the smali code is suitable for app modification purposes.

The process of reverse engineering using a decompiling technique [5] is shown as follows.

1. Modification point search: The activity information, UI layout and app execution flow are gathered. In the meanwhile, the points at which code is inserted are selected. One of the tool called Logcat [8] can be used to gather the activity information of the app. The OnCreate function of the activities can be decompiled then in order to obtain the UI information.

2. Decompilation: After extracting the DEX file in the APK file, baksmali [9] can be used to generate the smali source code.

3. Code injection and modification: Code containing arbitrary Dalvik VM instructions is inserted at the modification point or the existing code is modified.

4. Manifest change: The package name needs to be changed in the app manifest. By doing this, the app will not be conflict with existing apps and registered on the Android Market.

5. Self-signing: The modified app is self-signed to complete the repackaging process.

## 2.2 Remote attestation

Remote attestation is the activity of making a claim about properties of a target by supplying evidence to an appraiser over a network [10]. Equipped with a remote attestation method, a server can prevent an unauthenticated remote software from obtaining some secret information or some privileged service. For example, only the gaming consoles who pass the remote attestation can be allowed to connect to the gaming networks. The remote attestation can be divided into hardware-based attestation and software-based attestation [11].

Hardware-based attestation uses an external hardware component, such as Trusted Platform Module (TPM) to compute a cryptographic hash of the system's hardware and software configuration and attest it.  Trusted Platform Module (TPM) is a tamper proof piece of hardware (usually a chip) which provides secure cryptographic services, such as encryption/decryption and hashing, and secure storage. It follows the standards specified by Trusted Computing Group (TCG) [12].

According to its specification, a TCG-compliant platform (the attester), with an embedded Trusted Platform Module chip, attests to a remote platform (the challenger) with a digital signature upon the hashes of the states of its software components. The challenger evaluates the trustworthiness of the attester by verifying the signatures [13].

Software-based attestation is proposed to verify the resource constrained devices without using any additional security hardware. For example, SWATT is a software-based attestation scheme that uses the response timing of the memory checksum that uses the response timing of the memory checksum computation to identify the compromised embedded devices. In order to prevent replay attack, the prover's memory is traversed in SWATT in a random manner based on a challenge sent from the verifier [14].

Based on the communication procedure of the attestation, we can divide the generic software-based attestation into several types such as: challenge-response attestation, one-way attestation and unpredictable attestation. For the challenge-response attestation, the verifier sends a challenge to the target and verifies the response back from the target, such as SWATT. In one-way attestation, the target directly sends the value generated by the built-in algorithm to the verifier, the verifier checks the value by using the correct answer generated by the same algorithm. Unpredictable attestation is proposed to prevent reverse engineering attack, which dynamically generates the attestation module and sends it to the target. By doing this, the attacker cannot understand the detail of attestation algorithm by reverse engineering the target application. The detail of these three types are discussed in the chapter 3.

In our case, we focus on the detection of repackaged app on the mobile phone. Since the mobile device is resource constrained, it is not suitable for the hard-ware based remote attestation. Thus, We will focus on the software-based attestation in this paper.

## 2.3 Obfuscation

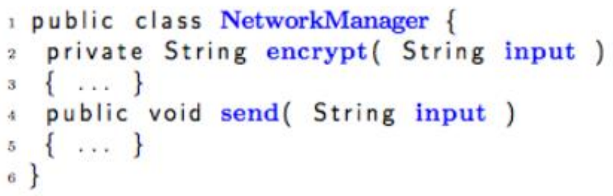
Software obfuscation is the set of techniques used to transform a program into another one which has the same sematic but much harder for human to understand [11]. To prevent the reverse engineering, developers always obfuscate their code before uploading. However, the code obfuscation can be used for both software developers and malware creators; On the one hand, the software developers use the technique to make it harder to plagiarize their work. On the other hands , the malware creators use it to elude anti-virus detectors [15]. In general, the obfuscation techniques must not change the function or behavior of programs, but their outer representation [16].

### 2.3.1 Obfuscation Techniques

**Identifier mangling**

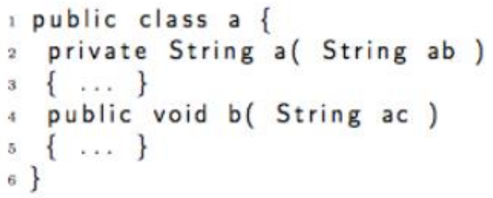
Identifier are the names for packages, classes, methods, field which present as strings in the code. Identifier mangling is the technique to change the identifiers from meaningful words to meaningless strings.  In the personal program, identifier shows the information and intention of the following code which is convenient for the developer to modify or reuse the code. However, the identifier can also leak these kind of information in the public environment, making the reverse engineering and static code analysis easier.

Identifier mangling tries to reduce the meta information contains in the identifiers. Instead of the meaningful identifiers wrote by developer, the random strings contain no information about the program behavior.  As we can see from the Figure 2-1 and Figure 2-2 [16].  The identifiers in Figure 2-1 imply this part of code deal with the network connection and encryption which is the useful information to attackers. Figure 2-2 shows the result after obfuscation, the meaningful words are replaced by random strings which contains no information about the program behavior.



```java
public class NetworkManager {
 private String encrypt( String input )
 { ... }
 public void send( String input )
 { ... }
}
```

**Figure 2-1 Original Java source code with highlight identifiers**



```java
public class a {
 private String a( String ab )
 { ... }
 public void b( String ac )
 { ... }
}
```

**Figure 2-2 Obfuscation Java source code**

The obfuscator ProGuard [17] uses the similar technique to obfuscate the codes. It uses minimal lexical-sorted strings like {a,b,c..aa,ab..} instead of random strings to save the space on mobile devices.

### String obfuscation

This technique is always used for data transformation in source code level. Apart from the string content, a reverse engineering attacker can also identify functions of interest by interpreting the context [18]. Therefore , these kind of strings should be obfuscated by application developer during the development. String obfuscation prevents not only the metadata information extraction but also static analysis.

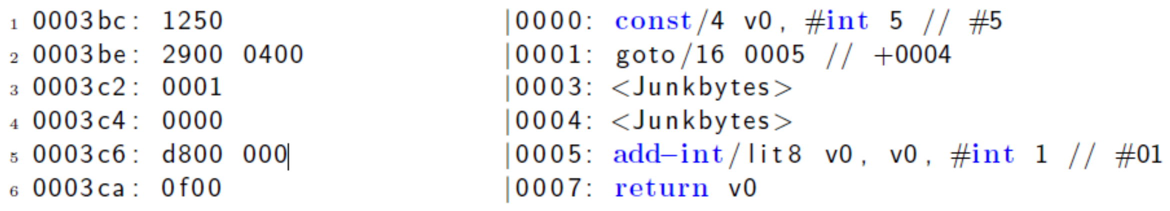
One important property of string obfuscation is the same as identifier mangling: obfuscating doesn’t change the semantic of the program. To achieve this, an invertible function F is required which transforms an arbitrary string into another string. The F(S) should be stored in the code as the obfuscation output. Whenever the original string is needed during runtime, the program can reverse the string S=F-1(F(S)) in order to not change the behaviour of the program [18]. The function F could be a standardized algorithm like XOR cipher or AES [19].

The technique aims to reduce the amount of extractable meta information. However, this technique can be defeat by dynamic analysis. Since the behaviour is not changed by the obfuscator in the runtime, the attacker could extract the content of string when the certain string is used. Consequently, string obfuscation can't prevent the information extraction entirely but it can make the string information extraction harder.

**Junkbytes**

Junkbyte insertion is a famous technique under x86 architecture [20]. The basic idea of junkbyte is confusing disassemblers by inserting some junkbyte in selected locations where the attacker expects an instruction.  Upon disassembly, the disassembler will interpret the junkbyte as a opcode. This would get the totally different program comparing with the program without junkbyte.

The position of junkbyte should be selected based on the disassemble procedure in order to get maximal obfuscated code. Apart from above, the junkbyte will never be executed to ensure the semantic of the program. Thus, a junkbyte must be located in basic block which will never be executed by adding unconditional branch in front of the block or conditional branch with certain constant result [21].
An example of Junkbyte is shown below [16].



```
0003bc:  1250                 |0000: const/4 v0, #int 5 // #5
0003be:  2900 0400            |0001: goto/16 0005 // +0004
0003c2:  0001                 |0003: <Junkbytes>
0003c4:  0000                 |0004: <Junkbytes>
0003c6:  d800 000             |0005: add-int/lit8 v0, v0, #int 1 // #01
0003ca:  0f00                 |0007: return v0
```

**Figure 2-3 Disassembly output with correct detection of junkbytes**

```
0003bc: 1250                       |0000: const/4 v0, #int 5 // #5
0003be: 2900 0400                  |0001: goto/16 0005 // +0004
0003c2: 0001 0000 d800 0001        |0003: packed-switch-data (4 units)
0003ca: 0f00                       |0007: return v0
```

**Figure 2-4 The linear sweep algorithm used by dexdump [22].**

```
0003bc: 1250                       |0000: const/4 v0, #int 5 // #5
0003be: 3c00 0400                  |0001: if-gtz v0, 0005 // +0004
0003c2: 0001 0000 d800 0001        |0003: packed-switch-data (4 units)
0003ca: 0f00                       |0007: return v0
```

**Figure 2-5 The recursive traversal algorithm produce disassembly errors**

In the Figure 2-3, we have inserted an unconditional branch at the address 0x3BE followed by two junkbytes. The unconditional branch ensure that the junkbytes will be jumped to address 0x3C6. This is a correct output generated by recursive traversal disassembling algorithm. However, as shown in Figure 2-4, the disassemblers like dexdump using linear sweep algorithm will get an incorrect output. By using the simple junkbyte insertion using unconditional branch, we lead the linear sweep algorithm to misinterpret the bytecode. In order to also cover recursive traversal algorithms, we can use incorporate conditional branches as shown in the Figure 2-5 [16].

By using the technique of junkbytes, the developers can confuse disassemblers. Although an analyst could modify the code after manually checking, the time consuming will be necessary. Thus, junkbyte insertion is still a valuable obfuscation technique [21].

### 2.3.2 Obfuscation Tools

Based on the obfuscation techniques we mentioned above, there are several obfuscation tools for Android code:

- ProGuard: ProGuard is a free Java obfuscator in Android SDK. It can protect the codes by removing unused code and renaming classes, fields, and methods with semantically obscure names. [17]ProGuard uses lexical-sorted strings instead of random characters to replace the original identifier in order to minimize memory usage.

- DexGuard: DexGuard is an advanced Android obfuscator which protects against both static analysis and dynamic analysis. DexGuard uses the not only techniques like string obfuscation, identifier obfuscation but also the tamper detection and SSL pining [23].

- Allatori: Allatori is a paid obfuscator includes the string obfuscation, junkbyte injection and watermarking techniques [24].

- Dalvik-obfuscator: Dalvik-obfuscator obfuscates bytecode by using the technique junk-byte insertion.  The goal of using this obfuscation technique is to combine junk byte injection with such instructions in order to hide as much of the original bytecode as possible [25].

## 2.4 Repackaging detection

Since it is not hard to reverse engineer the Dalvik bytecode in the DVM, Android applications are in danger of repackaging. The plagiarist can easily steal others' code by using the technique of reverse engineering. According to the study [3], among the analyzed 1260 malware samples, 86% were repackaged versions [26].

However, the repackaging detection techniques are also printing the progress based on the several repackaging detection methods proposed in the recent year. According to the procedure, we'd like to divide the methods into three parts, fuzzy hashing based detection, PDG based detection and feature hashing based detection.

### 2.4.1 Fuzzy hashing based detection

Fuzzy hashing algorithms are a type of compression functions for computing the similarity between individual files. It is first proposed in 2006 by Jesse Kornblum [27]. In the field of traditional cryptographic hashing like MD5 or SH1, the hash function can only determine whether the inputs are exactly same or not. However, the fuzzy hashing can hold a tolerance and calculate the similarity between two different digital files [28].

Based on the theory of fuzzy hashing, DroidMOSS [29] can measure the similarity between two apps. The DroidMOSS calculates the hash value of each opcode sequences unit of classes.dex and then concatenates all the values into a whole. Specifically, the author get plenty of apps from the official Android market and choose the extract some distinguishing feature from apps. After that, he generates the fingerprints of each apps based on fuzzy hashing and uses the similarity to detect the repackaging apps.

Since it divides the application into several pieces, DroidMOSS can efficiently identify the re-packaged applications when the code manipulation was only performed at a few interesting points. It has a very high true positive rate based on the experiment performed by the author.

### 2.4.2 PDG based detection

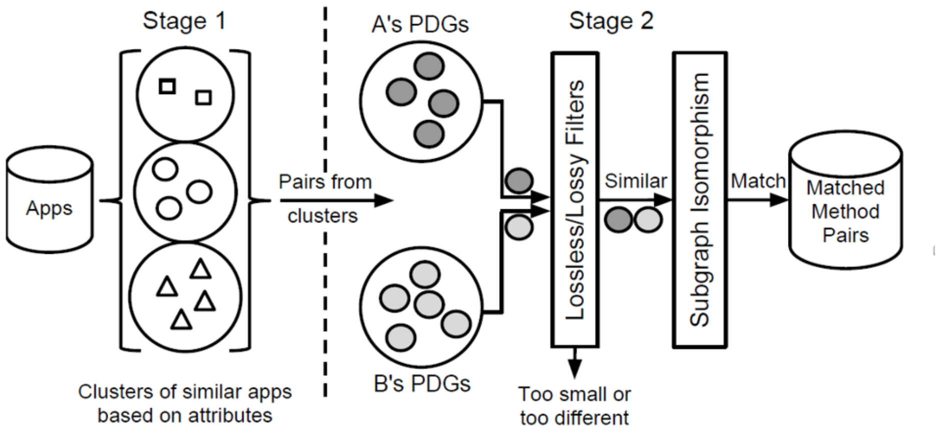


**Figure 2-6 Overview of DNADroid [30]**

A program dependence graph(PDG) is a graph representation of the source code of a program. It can represents the dependence of each operation in a program. Dendroid [30] uses the tool dex2jar to convert Dalvik byte codes to java byte codes, so they can utilize WALA [31] to compute the PDGs for every method. The similarity comparison between two different applications is based on the PDG pairs extracted from the apps.

As we can see from the Figure 2-6 above, Dendroid computes the PDG of each apps after clustering stage. After that, we apply lossless and lossy filters as first step. Once the apps pass the filters, we calculate the subgraph Isomorphism which attempts to find the mapping between nodes in two PDGs. Finally, the similarity scores are calculated based on the previous data.
In the Program Dependence Graph, each node is a statement and each edge shows the dependency between statements. The dependencies data and control can explicit express the structure of a program [32]. Since these dependencies do not change after the control flow obfuscations or signature modification, PDG-based detection is much more robust than normal structure based detection.

### 2.4.3 Feature hashing based detection

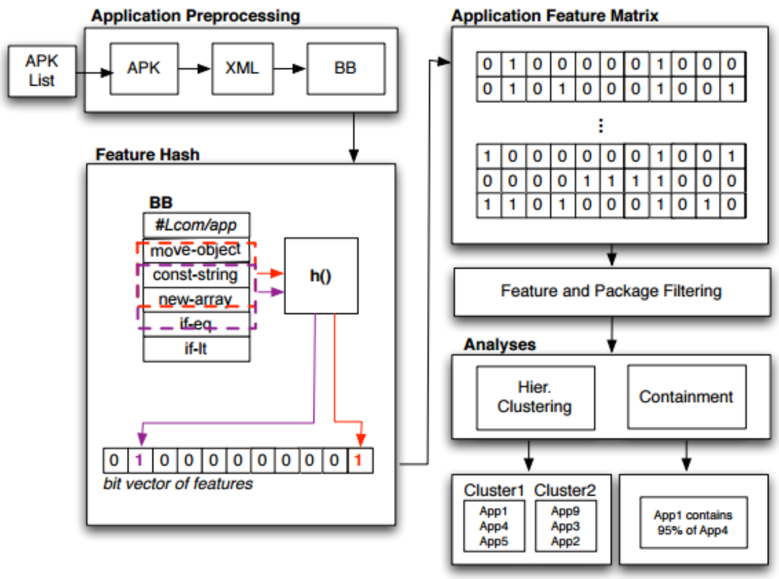


**Figure 2-7 The JuxtApp work flow [33]**

Feature hashing is a powerful technique for reducing the dimensionality of the data being analyzed. It is a fast and space-efficient way of vectorising features. It works by applying a hash function to the features and using their hash values as indices directly, rather than looking the indices up in a n associative array [34].

JuxtApp [33] is a code-reuse detection scheme based on feature hashing. In the pre-processing stage, JuxtApp extract the information from the DEX format file including class structure, function information, etc. In the feature extraction stage, JuxtApp use k-grams of opcodes and feature hashing to extract features from apps. As we can see from the Figure 2-7 above, it extracts each k-gram using a moving window of size k, and hash it using hash function dkb2. In the analyses stage, the JuxtApp clusters the applications and calculate the similarities between different apps.

However, the work [33] does not perform evaluation on the tool's false negative rate. Based on the analysis in [30], the certain code manipulation and special features of the program will lead the false negative rate of their detection phase. Although JuxtApp can reduce the false negatives by decreasing the size of sliding widow, it will also reduce the feature space and lead to more false positives.

In general, feature hashing based detection is still an advanced method against various obfuscation techniques.

## 2.5 App watermark

Watermarking embeds a secret message into a covert message [35], the techniques are used mainly in the entertainment industry to identify multimedia files such as audio video files. By using the same concept of watermarking, embedding watermarks in Android apps could potentially resist repackaged applications. The embedded watermark can be used to identify the owner of apps. By extracting watermarks from apps and comparing them, we can identify the repackaged pairs [36] .

There are two types of watermarks: static watermarks and dynamic watermarks. The static watermarks are stored in the application itself, whereas dynamic watermarks are constructed at runtime and dynamic state of the problem.

Moskowitz [37] and Davidson [38] are two techniques representative of typical static watermarks. Moskowitz embeds the watermarks in an imaging using one of the many media watermarking algorithm. Davidson describes a static watermark by encoding the fingerprint in the basic block sequence of program's control flow graphs. However, all the watermarks are susceptible and easy to defeat. For instance, any code motion optimization technique will destroy Davidson's method.

The dynamic watermark uses runtime information as the watermark such as running paths and memory status [39]. There is a method called AppInk [40], embeds a graph-based dynamic watermark into Android apps. By constructing a special list of objects as the watermark and using the distance between the objects in the list to represent a number. AppInk can embed any number as the watermark into an app. However, the code for constructing the graph is easy to be identified.

Although the dynamic watermark is harder for the attacker to crack than static watermark, there are still some typical attack approaches which are shown below [36].

1. The dependence analysis is an effective attack on semantics-based watermarks(e.g., graph-based watermarks). The smart attackers could perform data dependence analysis on the code of apps. They could possibly identify the watermark related code which does not have any relationship with the original code. Since there are several approaches embed watermark by adding independent code, the attack will separate the watermark code from the original code.
2. Assume the attackers know the characteristic of watermarks, the attacker can separate the watermarks related objects and break the watermark.

The Appmark [36] tried to combine watermarks with original apps by using picture-based watermark. One of the special characteristic for the pictures is, even if part of a picture is tampered, the picture could still be identified with high possibility. Thus, the Appmark is resilient to obfuscation since the inherent characteristics of pictures. It is also resilient to dependence analysis because it is highly combined with original code. However, the scheme just increases the difficulty for the attackers to crack but does not essentially prevent the attack.

# Chapter 3 Two-phase model for distinguishing apps

## 3.1 Introduction

Since repackaged apps can bring risks to both customer and app developer, several countermeasures for detecting and preventing the use of repackaged apps are proposed in recent years as we discussed in the chapter 2.

Obfuscation techniques and app watermark can increase the difficulty for the attacker to reverse engineer and repackage apps, however these techniques can only increase the cost of the attacks but cannot essentially prevent the attack related to repackaging application until now. The repackaging detection methods we introduced such as fuzzy hashing based detection and PDG based detection are mainly designed for detecting repackaged application in a group of applications. They cannot detect repackaged application real-time in the server side.

As we introduced in the chapter 1, we are going to focus on server-side Android app attestation which helps server to determine the authenticity of communicating apps in order to prevent repackaged apps. Thus, the remote attestation is the suitable technique which we can deep into. Since the server does not know the users are honest or not in our assumption, attackers can be both the person who uploaded the repackaged app to the app market and the user who repackaged the app himself with the authority of his device.

In this chapter, we first introduce and analyze the current models for server-side attestation of apps. Then, we describe the design principle of our two-phase model. In the end, we give an overview of our model and describe the functionality of each phase.

## 3.2 Previous attestation models

An attestation approach serves to confirm the integrity of a target. In our case, it checks whether the code of mobile app has been modified. Because of the hardware limitation of mobile devices, hardware-based attestation is not effective on mobile platform. Thus we focus on software-based attestation scheme and list three generic software-based attestation schemes. Among them, challenge-response attestation [41] and one-way attestation [42] are popular in current mobile apps, unpredictable attestation scheme [4] is proposed recently in order to improve them and proved to be viable.

**Challenge-response attestation**

A challenge-response protocol is that a device computes a response to a random challenge and returns to a verifier in order to check the integrity of the device. The verifier can detect the modification of the code in the device by checking the memory contents of the device included in the response.

The challenge-response attestation uses a suitable checksum function such as hash function to compute the checksum of the target apps. As we can see from the Figure 3-1 below, server transmits challenge C to the target application after receiving the accept request from target application. Then, the challenge C and the whole codes information of target app I will be calculated by the checksum function H, and the response R=H(C,I) will be sent back to the server. The server compares the challenge R with prepared answer R1 and makes the decision.

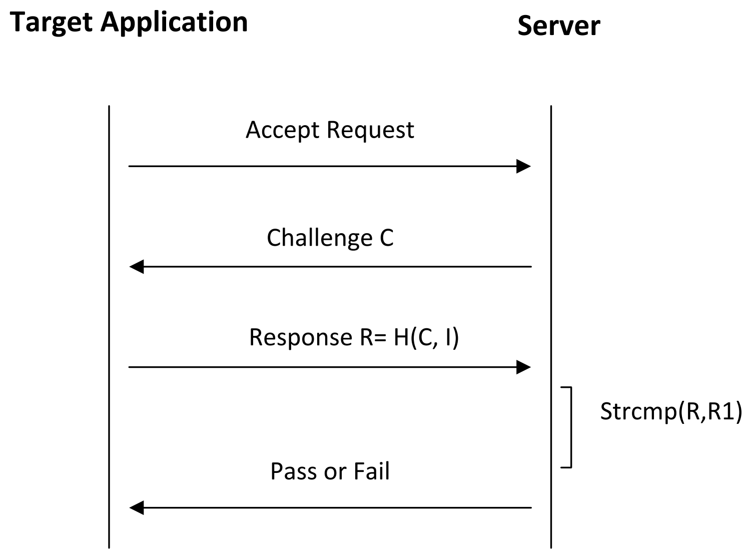


**Figure 3-1 Challenge-response attestation**

Since the response will be influenced by the changing of challenge, the replay attack can be avoided efficiently. However, since the attacker can also send the fake challenge to the target application and collect the response, he can store the challenge-response pair in the lookup table and perform a lookup table attack. Also, it is still susceptible to network attacks like Man-In-The-Middle attack unless they use SSL. Furthermore, since the attestation algorithm inside target application cannot be changed, the diversity of attestation algorithm is limited. After reverse-engineering the application, the attacker can build a suitable function to calculate and reply the correct response.

**One-way attestation**

In one-way attestation [42], a built-in algorithm is embedded in the target application. The target application sends the message to the server without any challenge. The server has the same built-in algorithm which can calculate the message based on the time. By checking the message, the server knows the memory content of the application to detect the modification inside target application.

Since the one-way attestation do not need a request from the server, the attacker cannot send fake requests to the application and build a lookup table. Furthermore, the attacker also cannot mislead application to send incorrect response to the server.

As we can see from the Figure 3-2 below. The client sends the message M without receiving any challenge from the server. The client has a built-in algorithm which shared with the verifier to generate the response content. After that, the server compares the received message M with M1 which synchronizing generated by server and makes a decision.

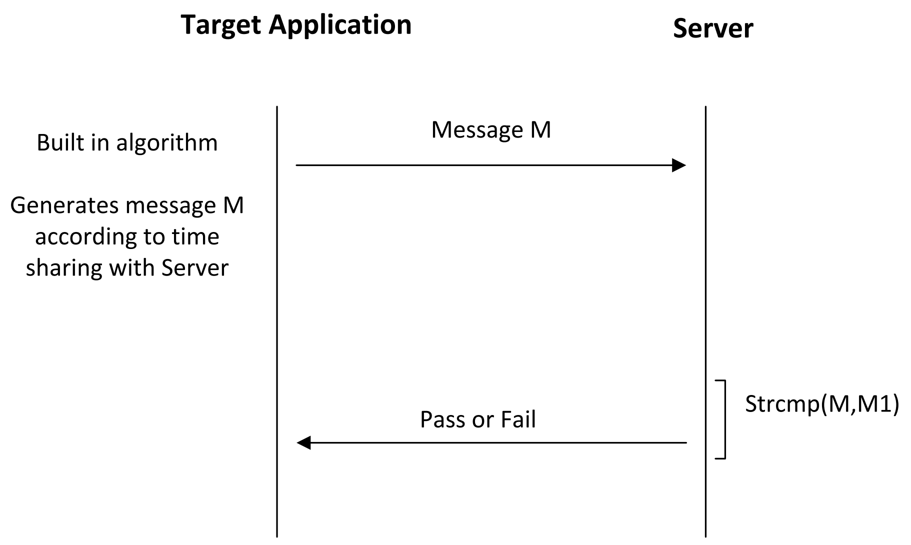


**Figure 3-2 One-way attestation**

The One-way attestation method is designed for preventing attacks related to challenge process such like lookup attack. However, since the built-in algorithm is located on the client side, the attacker can analyze the module by reverse engineering. After getting the information of the built-in algorithm, the attacker can pass the attestation easily.

**Unpredictable attestation**

In order to prevent the attacker from the current attack methods happens in Challenge-Response and One-way attestation scheme, Jihwang proposed a viable repackaged malapp detection scheme, called MysteryChecker [4], which uses unpredictable attestation algorithm. In both Challenge-Response attestation and one-way attestation, attestation algorithms are stored in the application in advance. However, for unpredictable attestation scheme, the server generates an attestation module with unpredictable attestation algorithm and transmits it after receiving the request from target application.

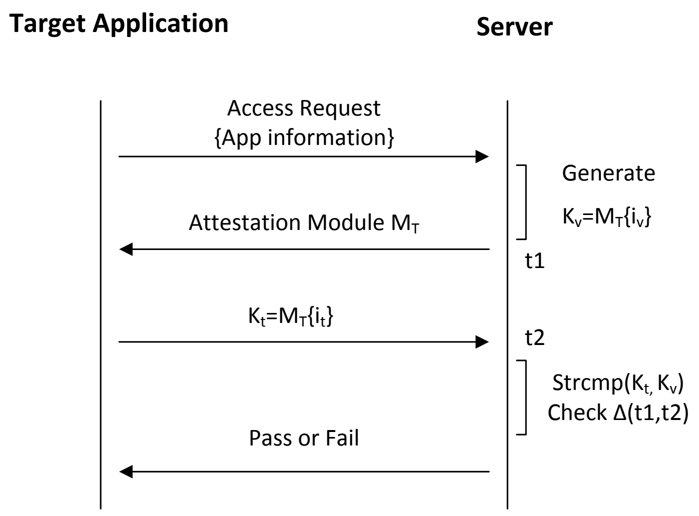


**Figure 3-3 Unpredictable attestation**

As Figure 3-3 shows above, the target application sends an access request with its information to the server to start the attestation procedure. The server randomly select information of target application $I_v$ and generates an unique attestation module $M_T$. The attestation module $M_T$ is transmitted to target application and the value $K_v = M_T(I_v)$ is stored in the server. The timer start in the meanwhile. The target application needs to execute the attestation module and calculates the required value $K_t = M_T(I_t)$. After receiving the response from target application, the server checks both required value $K_t$ and receiving time to judge the target application.

In this attestation scheme, the attestation algorithm needs to be generated by the server instead of storing in the application. Although the attacker can reverse engineer the target application and analyze all the functions inside, he cannot predict the next attestation algorithm because the server replaces the existing attestation module each time with an unpredictable new attestation module. Furthermore, if the attacker tries to analyze the new attestation module and performs the MITM attack, he cannot pass the time checking thus all the analysis becomes invalid.

Compare with the Challenge-Response attestation and One-way attestation, the unpredictable attestation scheme increases the diversity of attestation algorithm and prevents several existing attacks. However, the scheme is still possible to be cracked. The attacker can reverse engineer the client-side application and store the original application to a dummy folder. No matter what attestation algorithm the server sends, the attacker executes the attestation module on dummy folder with original codes instead of the repackaged codes. Thus the repackaged application can reply the server with correct response and pass the attestation. Thus, the unpredictable attestation still needs to be improved.

## 3.3 Attack models

We discuss several common attack models in this chapter in order to summary existing attack methods and try to avoid them in new attestation scheme.

1. Replay attack

   Replay attack is a form of network attack in which a valid data transmission is maliciously repeated or delayed. In our case, the attacker tries to store the response which sends by the original application and use it as the response when the server sends the request to his application.

   The replay attack is the one of the simplest attack method the attacker can perform, but it is still useful and efficient in some case which has barely security precaution. The diversity of challenges from server-side can prevent replay attack. Since the challenges are different in each time, the attacker cannot use the response collected before.

2. Lookup table attack

   The attacker can send fake challenges similar to server's challenges and receive responses to the fake challenges. The challenges and the received responses are stored in a table called lookup table. After collecting plenty of challenges-response pairs, the attacker can pretend to be the client to communicate with server by replying the certain response stored in the lookup table.

   The lookup attack is efficient in the case which the set of challenge-response pair is small or medium. The diversity of challenges will increase the time for constructing lookup table and make the method low-efficient. One-way attestation scheme can also prevent the lookup attack since the attacker cannot use the fake challenges to collect challenges-response pairs.

3. Brute force attack

Brute force attack in attestation scheme means that the attacker tries to access the server by trying all the combination as the response. It can be used in the attestation scheme which does not have huge responses set because the amount of responses will directly influence the time of attacking.

Although this attack method cannot be entirely prevented, it is really low efficient. The server can increase the scope of responses in order to increase the time for attacking and make it impossible.

4. Man-In-the-Middle attack

Man-In-the Middle(MITM) attack is an attack that the attacker relays or alters the communication between two parties who believe they are directly communicating with each other. In the cases of attestation, the MITM attack can be used to read and forward all the communication information including challenge, response or even attestation module. The MITM in attestation scheme is shown in the Figure 3-4 below.

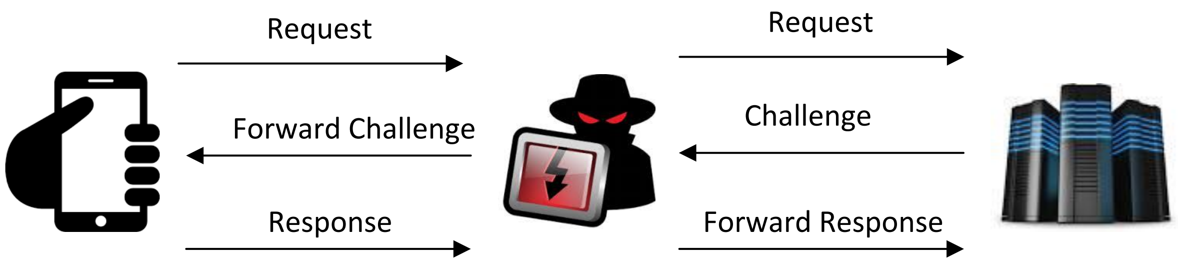


**Figure 3-4 Man-In-the-Middle attack**

The attacker performs as a middle man between original application and the server. When the attacker's app receives the challenge from server, he forwards the challenge to the original application and pretends to be the server. Then, the original application sends the response to the attacker which pretends to be a server. Finally, the attacker has the correct response and easily gets the access by forwarding the response.

The application and server can use HTTPS to prevent MITM attack since all the communication in HTTPS is encrypted. Although the attacker can still get the packages between the application and the server, he cannot decrypt them and communicate with the server using symmetric key. However, in the server-side attestation, the user can be compromised. Because the attacker can download the original application on his phone, he can do anything in his device including trust a certificate of his own proxy which can be used as a middle man. By doing this, the proxy pretends to be a server for the original application and pretends to be the original application for the server.

5. Reverse engineering attack

The attacker can reverse engineer the original application and get the source codes. All the functions and embedded information inside original application will be revealed after being analyzed by the attacker. The attacker knows the attestation algorithm after analyzing the source codes of original application. Thus, he can adjust his own application to get the correct response or build up a lookup table for the challenge.

Since the attacker can get all the source code of application, this attack is really strong to crack the attestation scheme. Although some of techniques such as obfuscation can be used to make the reverse engineering harder,  it cannot be totally resisted. Since the reverse engineering will take long time in some cases such as getting obfuscated code, the attacker needs to make a trade-off between getting potential value and costing. Consider a situation that an attacker costs two weeks to reverse engineer an application which updates once a week, it will be unnecessary and make no sense.

6. Dummy folder attack

This type of attack stores the original application into another place such as dummy folder.

When the server sends a challenge or attestation module to be executed, the repackaged application uses the original application in the dummy folder to authenticate instead of the original application. Then the repackaged application gets the correct response value which is expected by the server. We name the attack as dummy folder attack and the layout is shown below in the Figure 3-5.

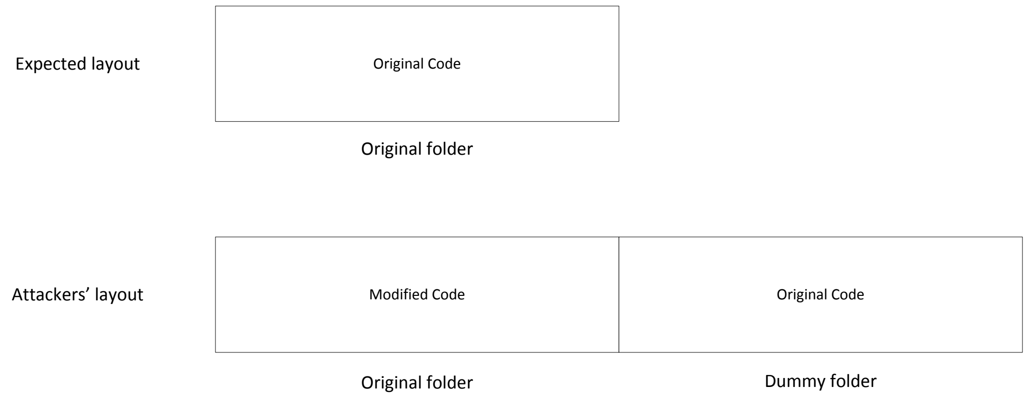


**Figure 3-5 Dummy folder attack**

Jihwan Jeong [4] discussed this attack method in mobile platform and proposed a precaution which uses absolute path to verify the execution environment. However, as we discussed before, the attacker who repackaged the app can also be the holder of the device, thus he can still forge the absolute path to avoid the attestation. We are going to improve the current attestation scheme in order to prevent dummy folder attack in Chapter 4.

7. Combination attack

The attack methods we mentioned above can be combined by the attacker based on his ability and the type of attestation scheme. For instance, the attacker can combine the technique of reverse engineering and MITM attack to reverse engineer the attestation module in the unpredictable attestation scheme.

The combination attack requires the attacker to have multiple skills. In the meanwhile, it increases the success possibility for an attacker to crack the attestation module. Thus, the attestation module should consider all the attack situations to prevent the combination attacks.

## 3.4 Two-phase model design

Since the existing server-side attestation solutions are still vulnerable as we discussed above , it is important to design a new attestation solution. As we can learn from the evolution of attestation schemes, a viable solution should contain the ability to detect not only the existing attack techniques but also unknown attacks. Combing with the other conditions in attestation module itself, we believe a viable approach to detecting repackaged applications should meet the next requirements:

**Low resource consumption**: Since the mobile smart-phone [4] is a resource-constrained environment, the smart phone has lower computation power than a traditional computer. The storage usage and computational time which can influence the usability of the application. Thus a lightweight computation is necessary.

**Procedure checking**: For each of the procedure in the attestation module, it should be properly checked to ensure it works in the expected way. In our case, the attestation module should be executed in correct environment.

**Behavior-based attack detection**: A viable attestation approach should have the ability to detect not only the existing attack techniques but also unknown attacks. The approach needs the ability to identify repackaged application by behaviors such as network behavior.

We design an two-phase model to meet the three requirements above. It consists two phases: 1) Detection phase 2) Analysis phase.  The detection phase is an improvement of existing attestation scheme and the analysis phase is added to perform a behavior-based attack detection.

The detection phase is designed to prevent the known attack methods by using techniques of hidden channel and dynamic code injection. It is based on the unpredictable attestation. In order to solve the problem of the unpredictable attestation scheme, we use hidden signal in HTTP channel which designed to be detected by the server, it ensures the execution environment of attestation module and avoid dummy folder attack. In the analysis phase, we analyze the distances of HTTP traffics in order to identify the repackaged application which passed the detection phase using unknown attack method. In the following, we explain the basic procedures of this attestation approach, as shown in Figure 3-6.

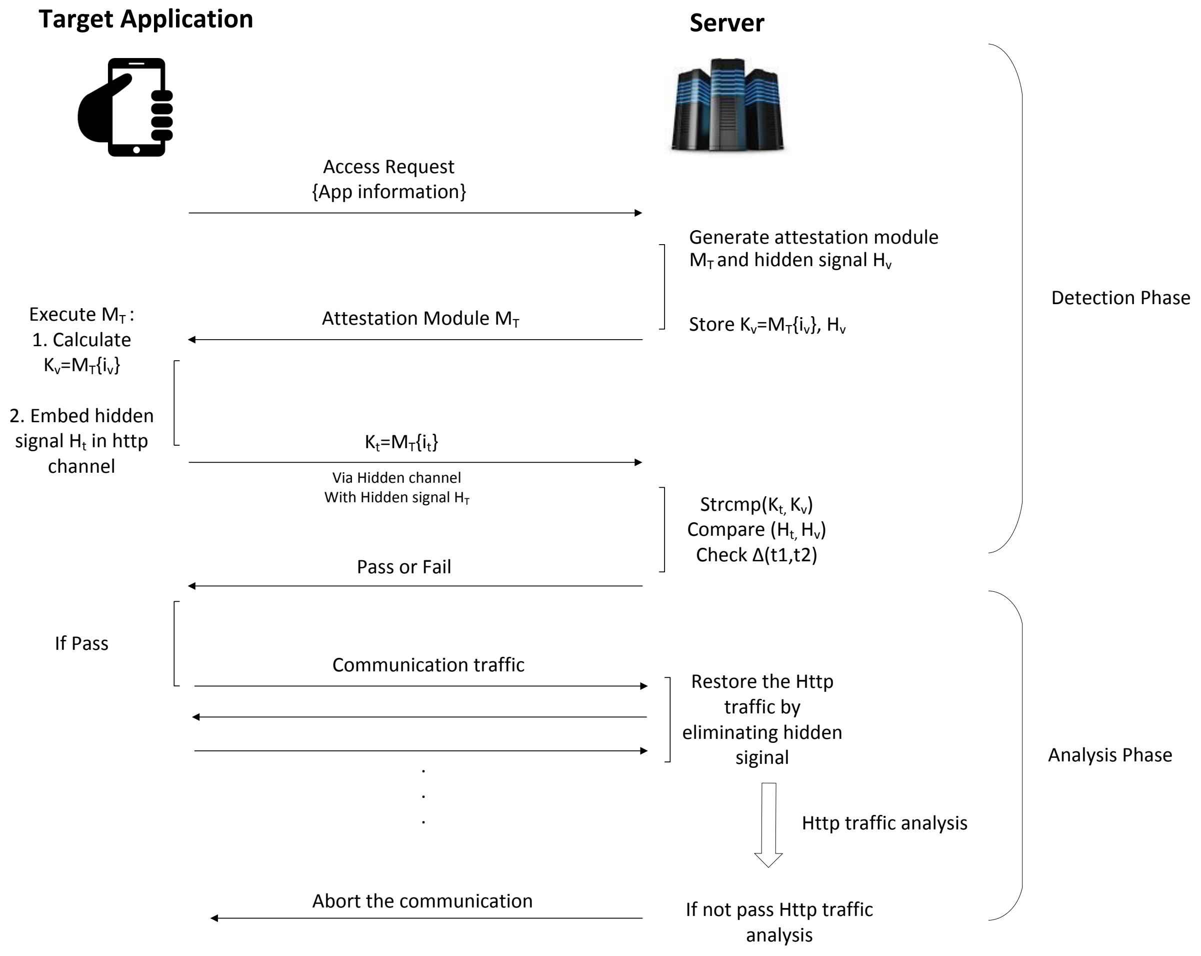

Target Application
Server
Access Request {App information}
Generate attestation module $M_T$ and hidden signal $H_v$
Detection Phase
Execute $M_T$ : 1. Calculate $K_v=M_T\{i_v\}$
Attestation Module $M_T$
Store $K_v=M_T\{i_v\}$, $H_v$
2. Embed hidden signal $H_t$ in http channel
$K_t=M_T\{i_t\}$
Via Hidden channel With Hidden signal $H_T$
Strcmp($K_t$, $K_v$)
Compare ($H_t$, $H_v$)
Check Δ(t1,t2)
Pass or Fail
If Pass
Communication traffic
Restore the Http traffic by eliminating hidden siginal
Analysis Phase
Http traffic analysis
Abort the communication
If not pass Http traffic analysis

**Figure 3-6 Two-phase model**

In the detection phase, the attestation module generator in the server-side randomly generates an attestation module after receiving the request from the target application(client). The attestation module consists two main parts, respectively verification part and modification part. The generator randomly selects a hash function and an attaching string for the verification part. As for the modification part, it is a function designed to modify the HTTP communication module inside the client in order to build up the covert channel in HTTP traffic. By using this covert channel (the detail is discussed in the chapter 4), we can verify whether the attestation module is executed in expecting way or not. The attestation module will be sent to client after generation.

The client executes the module and sends the result back to server via HTTP. The attestation module will not only calculate the hash value of the application but also modify the codes of HTTP communication part to embed the covert channel information. In the end of the detection phase, the server check the hash value of application, information inside HTTP covert channel and responding time. If all the values are correct, we consider the application pass the detection phase. Otherwise we consider the application as a repackaged application and the attestation will fail.

After passing the detection phase, the client communicate with server via HTTP traffic which will be analyzed. In the analysis phase, the server collects the HTTP traffic from client and calculates the distances(similarity) between collected HTTP traffic and standard HTTP traffic. (The detail of HTTP distance will be discussed in chapter 5.)  If the HTTP distance is abnormal, the server will abort the communication with client and we consider the application as a repackaged app.

## 3.5 Discussion

The detection phase ensures that the attestation module is unpredictable and the module can be executed in the correct environment. By using the technique of hidden channel and the dynamic code injection, the hidden signal can be embedded in the communication module of the target application. Once the attestation module is not executed in the correct folder, the hidden signal which server received will be different and expose the attack. Thus our model reaches the requirement “procedure checking”.

The analysis phase has the ability to identify varietal attacks by analyzing the HTTP traffic distances. The HTTP traffics are the behavior we are analyzing in this case, thus our model also reaches the requirement “behavior-based attack detection”.

All the calculations in this attestation solution are light-weight and they will not influence the usability of the application. According to the paper of unpredictable attestation scheme [4], the execution time of the attestation module will take 0.08-0.35 second which is affordable for the mobile device. Changing the parameter of the communication module in the target application is the only additional calculation in the target device by using our new scheme comparing to the unpredictable attestation scheme. The “low resource consumption” is also reached and the new solution meets three requirements we mentioned above.

# Chapter 4 Improving attestation scheme using hidden channel

## 4.1 Introduction

To overcome limitations of attestation schemes discussed in the previous chapter, we introduce the detection phase, which is responsible for identifying the repackaged applications and preventing the existing attack models.

According to the analysis in Chapter 3, the current attestation methods cannot prevent existing attack such as dummy folder attack. Among current attestation methods, unpredictable attestation scheme can detect and prevent almost all the existing typical attack models except dummy folder attack. In dummy folder attack, the attacker stores the code of original application into a dummy folder, he can calculate the required value by using the code in that folder and reply a correct value to the server no matter which part of the code the server need to check or what type of authentication function the server given. Thus we need to improve the unpredictable attestation scheme to prevent the dummy folder attack.

In this chapter, we design an improved attestation scheme which keeps the advantages of unpredictable attestation scheme and prevents the dummy folder attack. The scheme is used in the detection phase in our model.

## 4.2 Description

Based on the unpredictable attestation scheme, the server in our scheme still generates unpredictable attestation module for target application by using different hash functions and string transformation methods.  Moreover, in order to avoid the target app executing attestation module in dummy folder, the attestation module running at the client embeds a hidden signal in the communication which the server can detect. Thus, the dummy folder attack can be detected since the attestation module is executed in dummy folder and the hidden signal is embedded in the communication module inside dummy folder but not target application.

We briefly introduce our detection phase here and the procedures are shown in the Figure 4-1.

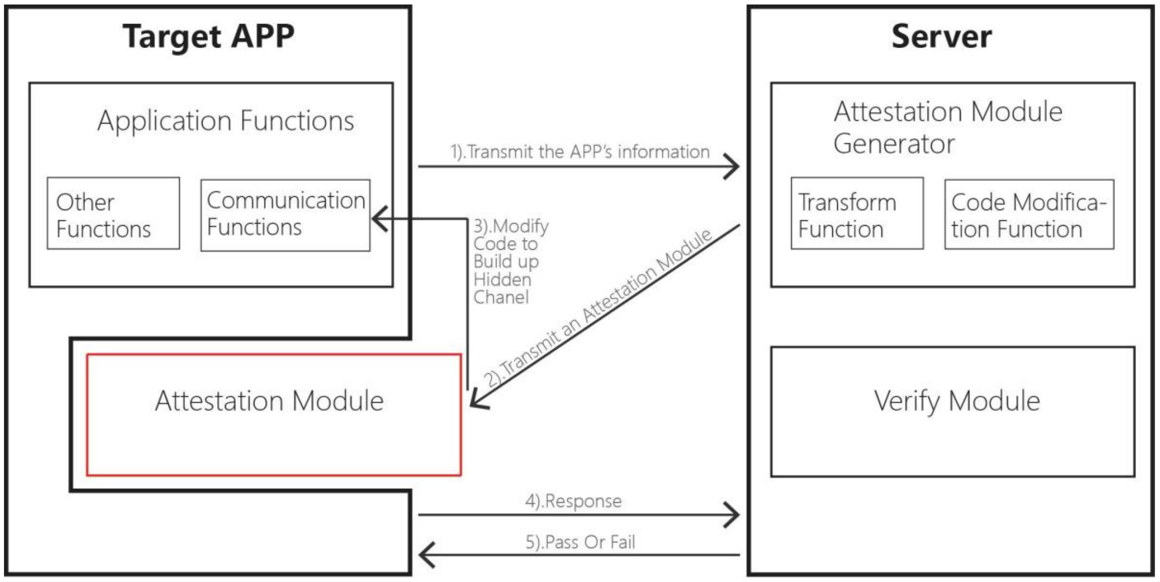

Target APP
Application Functions
Other Functions
Communication Functions
Attestation Module
Server
Attestation Module Generator
Transform Function
Code Modification Function
Verify Module
1).Transmit the APP's information
2).Transmit an Attestation Module
3).Modify Code to Build up Hidden Chanel
4).Response
5).Pass Or Fail

**Figure 4-1 Detection phase procedure flow**

When a user installs and launches a target application which needs to communicate with server, the server can detect whether the application is original or repackaged. (1) First, the application attempts to access the server. The target application sends application information with access request to the server. (2) The server generates an attestation module with hidden channel information and transmits it to the target application. The server starts a timer at the same time. (3) The target executes the attestation module and calculates the needed value. In the meanwhile, it will dynamically change the code of HTTP communication module in the original application to build up a hidden channel. (4) The target application sends the response value to the server via HTTP channel which has already been embedded as a hidden channel. (5) The server checks the response value, time and the hidden channel content. If all of the requirements are met, the target application pass the detection phase in server-side. Otherwise, the application is considered as non-original application. Since the hidden message has already embedded in the communication channel in the target application, the further communication will also be checked by the server. In some cases, the server does not give immediate feedback to the target application, the attacker will not know what he did wrong and cost plenty of time to figure it out.

Our improved attestation scheme in detection phase provides secure attestation through unpredictable algorithms and hidden channel. The unpredictable attestation algorithm prevents the attacker from predicting the attestation algorithm. In the meanwhile, the unpredictable hidden channel information prevents the dummy folder attack.

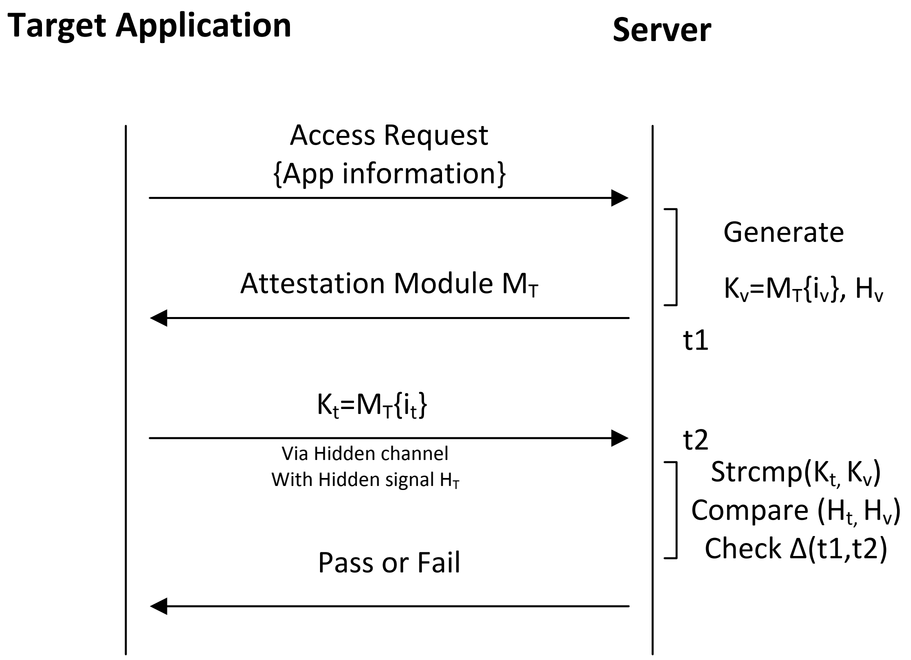


**Figure 4-2 Messages exchange in detection phase**

The message change of our new attestation scheme is shown in the Figure 4-2. The target application sends an access request with app information to start the attestation process. The server selects information $i_V$ and randomly generates an attestation module $M_T$ which calculates the value $K_v= M_T\{i_v\}$. In the meanwhile, the server also generates a hidden signal $H_v$ and embeds an code modification function in the attestation module based on the hidden signal $H_v$. The attestation module $M_T$ is transmitted to the target application while the timer t1 start. After executing the attestation module $M_T$, the target application sends the value $K_t$= $M_T\{i_t\}$ to the server via hidden channel with hidden signal $H_T$. The server checks strcmp(Kt, Kv), Compare(Ht, Hv) and $\Delta(t1,t2)$ before making a decision.

When a repackaged application executes the attestation module, the response value required by the server is sent via hidden channel. The response value will be incorrect because the attestation algorithm needs the hash value of the whole application.

If an attacker stores the original application code into the dummy folder and use it to calculate the required value, the hidden channel will be embedded into the HTTP communication module of the original code in the dummy folder. However, this HTTP communication module will not be used since the code in dummy folder is only designed for calculating required value. Thus, the attacker will be exposed when the repackaged application sends the required value via his own HTTP communication module which is not changed and embedded hidden information by attestation module.

Moreover, if a smart attacker knows the hidden channel in HTTP communication module, he needs to modify his own HTTP communication module to embed the correct information in hidden channel. Since the information in attestation module is different and unpredictable each time, the attacker needs to reverse engineer the attestation module to get the hidden channel information. However, this behavior will cost long time thus the application cannot pass the time checking in the server side.

The detection phase consists of three core techniques, respectively attestation module generation, dynamic code injection and hidden channel. We briefly introduce the techniques below and discuss the detail in the further chapters.

**1) Attestation module generation:**

An attestation module verifies the target application. The server contains an attention module generator which randomly generates an attestation module by using the unique information of target application, e.g. hash value of target application. The generator randomly selects information of target application and builds a transformation function chain for the attestation module. The transformation function chain is a stack or sequence of transformation functions such as string reordering, random string attaching and hashing by several possible hash algorithms [4]. Besides, the generator also randomly selects a covert message seed to generate the code modification function for the attestation module. Once the target application executes the attestation module, not only a required value will be calculated, but also the HTTP communication module in target application will be modified and embedded by the covert message.

**2) Hidden channel:**

Hidden channel is a technique of information hiding in order to transmit information in unnoticed way [43]. In our detection phase, we build up hidden channel via HTTP communication to ensure the authenticity of target application and prevent the dummy folder attack. We will show how to embed a hidden channel in HTTP communication and explain how to implement it by using dynamic code injection technique. The server generates a covert seed and builds up a code modification function based on the regulation. This code modification function will be used in the attestation module and applied in target application.

**3) Dynamic code injection:**

Dynamic code injection is a technique which allows the application to load additional code at runtime. The technique can be used either in good way or bad way, thus there are several papers such as [44] discussed about how to prevent the security issues of using this technique. We use the technique in good way which improves the attestation scheme in order to detect the repackaged application.

In our detection phase, the dynamic code injection is used for building up the hidden channel in target application. In order to embed the covert information in HTTP communication module of target application at runtime, we need the dynamic code injection technique. By using this technique, the covert information can be embedded by modifying the parameters of HTTP communication module in target application. The covert information is randomly generated by server and attached in attestation module. When the target application execute attestation module, the HTTP communication module of target application will be modified according to the covert information in attestation module. After the modification, all the messages sent via this communication module contains this covert information and server can identify the app by using this information. We use Dexclassloader to achieve this technique, the details are shown in the chapter 4.5.

## 4.3 Attestation module generation

When a user installs a target application, the application sends an access request with information of its name and version to the server in order to communicate with server. After receiving the information of target application, the server generates an attestation module for target application. This attestation module not only attests the target application by verifying the required answer, but also builds up the hidden channel and verifies the embedded covert message to ensure that the attestation module is executed over correct environment but not dummy folder. There are two main functions inside attestation module, respectively transformation function and code modification function. Transformation function is similar to the unpredictable attestation method which we discussed in chapter 3, code modification function is added in our module in order to prevent dummy folder attack. The whole process in attestation module generator is shown in the Figure 4-3.

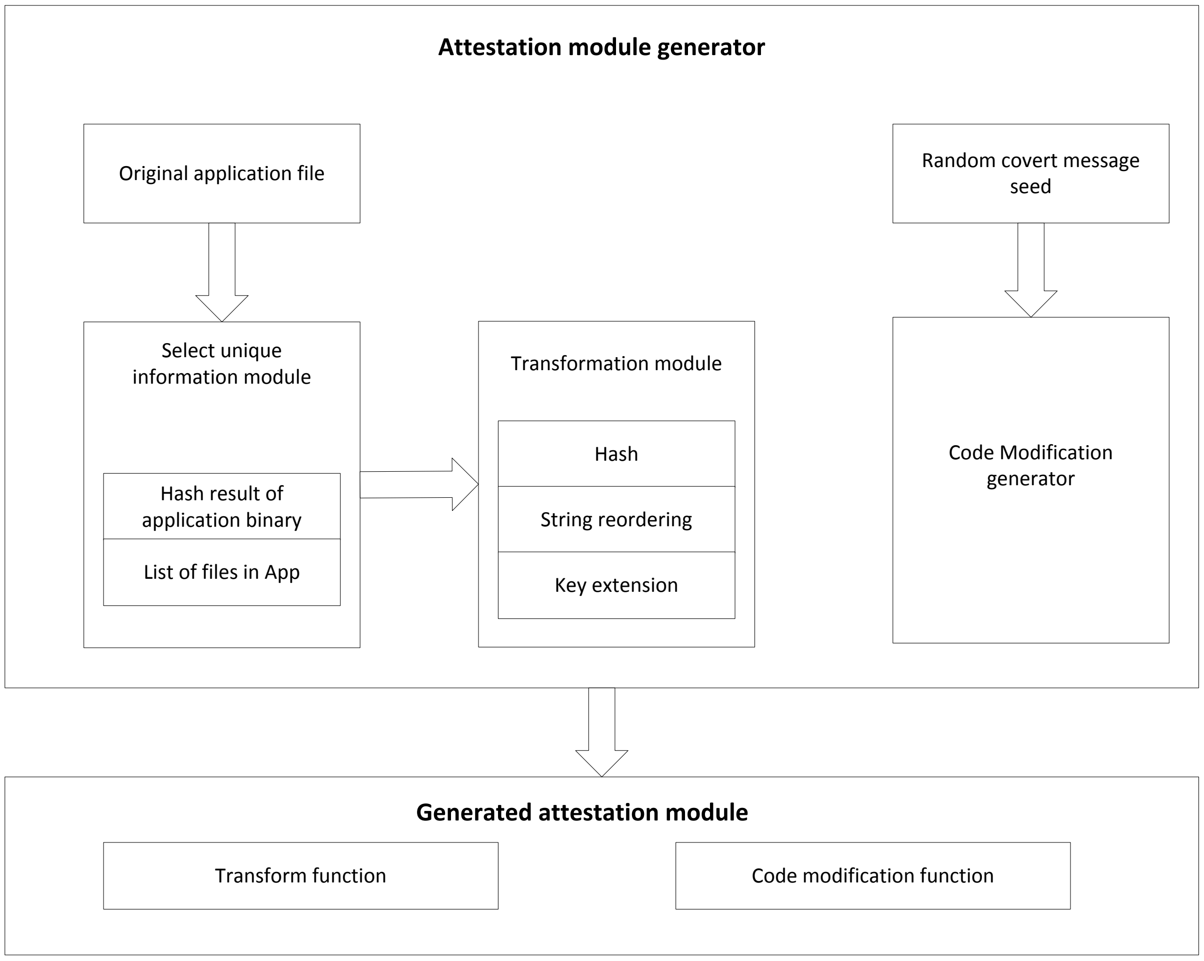

Attestation module generator
Original application file
Random covert message seed
Select unique information module
Hash result of application binary
List of files in App
Transformation module
Hash
String reordering
Key extension
Code Modification generator
Generated attestation module
Transform function
Code modification function

**Figure 4-3 Structure of attestation module generator**

For the transformation function of attestation module, the server randomly select one information of target application such like Hash result of application binary and list of files in the application. The original APK file and version information of the target application are stored in server. This selection step makes it harder for attackers to guess the right input parameter of the attestation module. After selecting information, the server builds a transformation function T which converts the selected information into the request value K. The transformation function T consists reordering functions R, random string attaching functions A and hash function H using several different crypto hash algorithms. The detail of reordering functions, random string attaching functions and crypto hash algorithms have described in the previous paper [4]. By using the method above, the transformation function inside attestation module is generated by the attestation module generator in the server.

In the meanwhile, the server randomly generates a covert message seed and puts it into the code modification generator. Since the covert message needs to be embedded in the hidden channel in HTTP communication module of the target application, the code modification generator translates the seed into covert message and builds up the code modification function based on the covert message. The code modification function will be used for building up hidden channel by modifying the parameter of HTTP communication module in target application and embedding the covert message in it. The detail of how to translate covert message seed is explained in the further chapter.

After transformation function and code modification function are generated, the server packages these two functions with selected information and builds the attestation module. The request value K and the random covert message seed are stored in the server in order for verifying application's response in next step.

## 4.4 Hidden channel

Hidden channel is a technique of information hiding in order to transfer information objects in the way that are not supposed to be detected by other people. It is a secret communication technique employed by two or more parties allowed to exchange information, while they assume the data channel in use is under surveillance [43]. In most of cases, the technique is used for the attackers to communicate or transfer assets in the way which is not allowed by computer security policy. However, we can also use the technique to hide the information from the attackers who can reverse engineer the application and study the original code.

There are several hidden channels can be built up such as IP Header Tunneling, DNS Tunneling and HTTP Tunneling. In our case, we focus on the HTTP hidden channel because the target application sends the requested value of attestation module to the server by using HTTP protocol.

### 4.4.1 HTTP protocol and hidden channel

The Hypertext transfer Protocol (HTTP) is an application protocol for distributed, collaborative, hypermedia information systems [44]. Development of HTTP was initiated by Tim Berners-Lee at CERN in 1989. Standards development of HTTP was coordinated by Internet Engineering Task Force (IETF) and the World Wide Web Consortium (W3C), culminating in the publication of a series of Requests for Comments (RFCs). The first definition of HTTP/1.1, the version of HTTP in common use, occurred in RFC 2068 in 1997, although this was obsoleted by RFC 2616 in 1999 [45].

HTTP is designed to permit intermediate network elements to improve or enable communication between clients and servers. A typical request is shown with following syntax:

```
GET https://googleads.g.doubleclick.net/mads/static/mad/sdk/native/sdk-core-v40.appcache HTTP/1.1
Host: googleads.g.doubleclick.net
Connection: keep-alive
If-None-Match: 11844712534071878399
User-Agent: Mozilla/5.0 (Linux; Android 4.4.4; SM-G8508S Build/KTU84P)
```

In order to build up a hidden channel and embed information in HTTP transactions, there are several methods we can find currently.  We list several methods below [43].

**Case-insensitivity.**

Since the field-names of HTTP protocol are case-insensitive, it can be the carrier for hidden bits. The bits can be differed by capital letters and lower case letters, bit ‘1’ represents capital letter and bit ‘0’ represents lower case letters. An example below shows how the covert information hidden in the HTTP header:

```
Connection: keep-alive
```

This header can be modified to carry the information bits ‘1010001010’. Ten digital means the length of the word ‘Connection’ and the ‘1’ and ‘0’ respectively represents the capital letter and lower case letter. Thus the header looks as follow:

```
CoNnecTiOn: keep-alive
```

By using the encoding method above, the information can be easily embedded in the HTTP header. However, it would be exposed if a visual inspection occurs by the checker. Thus it is not wise to hide too much information by using this method.

**White spacing.**

According to the RFC 2616, the field-content can be preceded and followed by an optional linear white spacing which includes non-compulsory CRLF and one or more space(SP) or horizontal tab (HT) character.  This allows additional (SP) or (HT) in HTTP messages and has not real meaning. Thus, it is possible to embed information by using space (SP) or horizontal tab (HT). For instance, we define SP as '0' and HT as '1' and we can embed information bits '1010001010' in following ways:

```
Connection: [HT][SP][HT][SP][SP][SP][HT][SP][HT][SP] keep-alive
Connection: keep-alive [HT][SP][HT][SP][SP][SP][HT][SP][HT][SP]
```

Theoretically speaking, there is no limit to the number of bits can be sent per message. However, the disguise could be revealed if the embedded HTTP message is checked by the visual examination. Thus it is better to embed only few information and distribute them in different headers when using this method for hiding information.

**Order of headers**

The HTTP messages use the generic-message syntax which does not specify the order of the headers should occur in the messages. Thus if both sides agree with using the hidden channel, the order of the headers would be capable to transmit information. For example in a basic hidden channel which has two headers, one of the header orders stands for '1' and another order stands for '0'.

```
Connection: keep-alive                        Can be decode as '1'
If-None-Match: 11844712534071878399

If-None-Match: 11844712534071878399           Can be decode as '0'
Connection: keep-alive
```

### 4.4.2 HTTP Hidden channel design in repackaged app detection

In the detection phase of our model, the HTTP hidden channel is used for verification but not communication. The target application do not need to translate the covert information, the covert channel is built up by using the attestation module and sends the covert information to server. The main goal of this hidden channel design is to raise the capability of information embedding and lower the risk of channel exposure.

#### a) Message hidden method

Since the target application transmits the request value to server via HTTP request, we build up the hidden channel in HTTP request headers. Some of the headers contains multiple options or different formats which can be used to embed information as a hidden channel.

For example, the header Accept-Language has a list including en (English), es (Spanish), de (German), it (Italian), en-US (English/United States), en-GR (English/United Kingdom), zh-CN (Chinese) etc. All this information can be predefined as a bit in the server and used for hiding information.

```
Accept-Language: zh-CN, en-US;q=0.8, de;q=0.7, es
```

As we can see from the example above, there is a value q in the content of the header Accept-Language. According to the RFC 2616, q is the quality factor which allows the user or user agent to indicate the relative degree of preference for that media-range, using the q-value scale from 0 to 1, the default value is 1. Since the server knows the design of original application, it does not require the information from target application. Thus we define the quality value q=0.3, 0.4, 0.5..0.9, 1 represents the binary cipher code from 000 to 111. Apart from this, the server needs to generate a random cipher code table of item types selected from the list. The matchup and the order will be different each time.  One of the possibilities is shown below as an example:

| Item Type | Cipher code |
|---|---|
| en-US | 00 |
| zh-CN | 01 |
| en | 10 |
| es | 11 |

**Table 4-1 Accept-language Cipher code table based on Item type**

| Quality factor | Cipher code |
|---|---|
| 0.3 | 000 |
| 0.4 | 001 |
| 0.5 | 010 |
| …. | … |
| 0.9 | 110 |
| 1 | 111 |

**Table 4-2 Accept-language Cipher code table based on Quality factor**

The first 2 digitals represent the item type in Accept-Language and the last 3 digitals represent the quality factor. According to the cipher code table randomly generated by the server, the information inside Accept-Language header is translated as following.

en-US;q=0.8 → 00101 (en-US → 00 and q=0.8 → 101)
Accept-Language: zh-CN, en-US;q=0.8, de;q=0.7, es → 01111 00101 10100 11111

The header Date and Last-Modified can also be used to embed information. HTTP application allowed three different formats for the representation of date/time stamps[RFC2616]:

```
Sun, 06 Nov 1994 08:49:37 GMT  ; RFC 822, updated by RFC 1123
Sunday, 06-Nov-94 08:49:37 GMT ; RFC 850, obsoleted by RFC 1036
Sun Nov  6 08:49:37 1994       ; ANSI C's asctime() format
```

Although the second format in RFC 850 has been obsoleted by RFC 1036, HTTP/1.1 clients and servers that parse the date value must accept all three formats. Thus we can use the difference of three formats to hide information. For example, the server can define the bits '00' to represent that he receives the Date header in first format, ‘01’ as the second format and ‘10’ as the third format.

```
Sun, 06 Nov 1994 08:49:37 GMT;  →  00
Sunday, 06-Nov-94 08:49:37 GMT;  →  01
Sun Nov  6 08:49:37 1994;  →  10
```

**b) Hidden message generation**

The server generates a hidden message seed and use it to build the code modification function in attestation module. We take a certain example to show the process of hidden message generation by server in this section.

At the beginning, the server generates a number n with range 1 to 4 (The range can be changed due to different requirement of different apps) to determine how many headers will be used to hide information. Then, the server randomly choose n headers in the available header list. We take n=4 and four chosen header Content-Type, Accept-Language, Accept-Encoding, Date as an example.

| Headers | Cipher code |
|---|---|
| Content-Type | 00 |
| Accept-Language | 01 |
| Accept-Encoding | 10 |
| Date | 11 |

**Table 4-3 Cipher table based on header types**

Then, the server generates the cipher code to matchup with these four headers. The cipher code indicates the order of these headers in a HTTP request.  Assume the server generates the HTTP request headers as below:

```
Accept-Language: zh-CN,en-US;q=0.8, de;q=0.7,es
Content-Type: application/json, text/html
Date: Sun, 06 Nov 1994 08:49:37 GMT
Accept-Encoding: gzip,deflate,bzip2,xpress
```

According to the cipher table of headers, the HTTP request headers can be translated as :

01 xx..xx 00 xx..xx 11 xx..xx 10 xx..xx

The digitals 00, 01, 10, 11 respectively represents the order of the header Content-Type, Accept-Language, Accept-Encoding, Date. The “xx..xx” represents the embedded information in the header which the front digitals represents.  The format of embedded information “xx..xx” depends on the type of the header.

The header Accept-Language uses the format we explained before, five digitals represents for one item, the first two digitals represent for the item type and the last three digitals represent for the quality factor. The server randomly generates the item type cipher table each time. We use the table 4-1 as an example, the header “Accept-Language: zh-CN,en-US;q=0.8, de;q=0.7,es” hides the information 01111 00101 10100 11111. In this case, all the information in the header is covert message and no original information included. Because the server has the information about the Accept-Language in original application, it does not require this information at all.

The header Content-Type and Accept-Encoding use the same format as Accept-Language. The server generates the cipher table for each of the headers and embeds the information based on the table. Because of the same format, we take 10111 01111 representing the information in Content-Type and 01111 00111 10111 11111 representing the information in Accept-Encoding as the example.

For the header Date, we use different date representing format to hide information as we discussed. There are three different formats thus we use two digitals to represent. We use 11 as an example to represent the format of the date” Date: Sun, 06 Nov 1994 08:49:37 GMT” although the server generates the cipher table each time.

Based on the information above, the whole HTTP request headers contains the covert message seed 01 01111 00101 10100 11111 00 10111 01111 11 11 10 01111 00111 10111 11111.

| | |
|---|---|
| Accept-Language: zh-CN,en-US;q=0.8, de;q=0.7,es | 01 01111 00101 10100 11111 |
| Content-Type: application/json, text/html | 00 10111 01111 |
| Date: Sun, 06 Nov 1994 08:49:37 GMT | 11 11 |
| Accept-Encoding: gzip,deflate,bzip2,xpress | 10 01111 00111 10111 11111. |

The whole process can be described as : 1) The server generates a covert message seed which can be translated into the group of headers H. 2) The seed is sent to the attestation module generator and the code modification function in the attestation module is built. The code modification function has the ability to modify the communication module in target application, and the modified communication module can send messages via HTTP tunnel with the header group H. 3) The server checks the H to finish the hidden channel verification.

The capability of information embedding for our hidden channel can be adjust by the parameter n, which represents the maximum numbers of information embedded headers in one HTTP hidden channel. Although this hidden channel is settled secretly, it still has possibility to be exposed. The higher capability of information embedding is, the higher risk of channel exposure is.

It is necessary to find a balance of information embedding capability and the channel exposure risk. However, the requirement of information embedding capability and the channel exposure risk depend on the type of different applications. Thus, the parameter n can be adjusted based on the type of the certain application.

### 4.5 Dynamic code injection

In order to build up the hidden channel in the target application, the application needs to load the code modification function inside attestation module at runtime. Thus, the dynamic code injection technique is needed in our detection phase.

The Android system allows the applications to load additional code from external sources at runtimes. There are several dynamic code injection techniques such like Class loaders and JNI (Java Native Interface). We choose Classloaders as our method to achieve our goal.

Classloaders are Java objects that allow program to load additional classes. Android applications can use class loaders to load classes from arbitrary files. It supports different file formats like APK, JAR, pure dex files, Optimized dex files [44]. By using one of the Classloaders called DexClassLoader, an application can download an APK file from Internet and invoke the classes' methods without installing the APK.

We write a test function in target application which allows the application loading and invoking the functions in attestation module "Attestation.apk". We assume the  "Attestation.apk" is sent by the server and stored in the same device with target application. The main function is shown as below:

```
public void useDexClassLoader() {



    String path = Environment.getExternalStorageDirectory() + File.separator;
    String filename = "Attestaion.apk";
    File dexOutputDir = getDir("dex",0);
    DexClassLoader classLoader = new DexClassLoader(path + filename, dexOutputDir.getAbso-
lutePath(),null, getClassLoader());



    try {
        Class mLoadClass = classLoader.loadClass("com.example.s142640.attenstation.MainAc-
tivity");
        Constructor constructor = mLoadClass.getConstructor(new Class[] {});
        Object obj = constructor.newInstance(new Object[] {});

        Method CodeModification = mLoadClass.getMethod("CodeModification", null);
        getMoney.setAccessible(true);
        CodeModification.invoke(obj, null);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    } catch (SecurityException e) {
        e.printStackTrace();
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (IllegalArgumentException e) {
        e.printStackTrace();
    } catch (InstantiationException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    }
```

By using the dynamic code injection technique, the target application can invoke the code modification function inside attestation module. Then the hidden channel can be built and the covert message can be embedded.

## 4.6 Discussion

**Replay attack and brute force attack**

The replay attack and brute force attack can be prevented since the diversity of attestation module. In our attestation scheme, the attestation module includes transformation function and code modification function. The code modification function needs to embed random covert message which generated by the server. The transformation function uses the same methods as unpredictable attestation scheme which can generates 2188*1/3 different attestation algorithm in one round. Thus, the unpredictable attestation algorithm with random covert message will be unique in each round which cannot be attacked by replay attack or brute force attack.

**Man in the middle attack**

Although the attacker can use the proxy to perform the MITM attack and intercept all the communication between original application and server, he cannot communicate with server using repackaged application. We assume that the attacker installs the original application in his phone which trusts prepared proxy. He uses the proxy to perform MITM attack and get the correct response from original application. However, the response only gives him the opportunity to communicate with the server, the further communication will still be checked by the server. Thus, the further request generated by the repackaged application without correct hidden message will be exposed.

**Reverse engineer attack**

In our attestation scheme, the attestation algorithms are generated by the server but not stored in the application. Thus, the reverse engineer of the target application cannot help attackers to pass the attestation scheme.

**Dummy folder attack**

We use the hidden channel and dynamic code injection to prevent dummy folder attack in our attestation scheme. Although the attacker can use dummy folder attack to calculate the correct response value, he will be exposed by the hidden channel since the communication module has not been changed in his repackaged application.

**Man in the middle attack combines reverse engineer attack**

In order to calculate the correct response value and embed the correct covert message in the communication module, the attacker can perform a MITM attack to get the attestation module and reverse engineer it. Since both the attestation algorithm and the covert message can be found in the attestation module, the attacker can modify his application to fit the attestation module in order to pass the attestation.  However, the time checking in our attestation scheme prevents this attack since the reverse engineering cannot easily success in the short period.

In the chapter, we designed an attestation scheme which is responsible for identifying the repackaged apps and preventing the existing attack models. By using the techniques of hidden channel and dynamic code injection, our attestation scheme improves the unpredictable attestation scheme to prevent the dummy folder attack. Based on the analysis of the performance in existing attack model, our attestation scheme is efficient.

Apart from the structure design of the new attestation scheme, we also looked into the implementation details. Based on the analysis of current hidden signal embedding methods in HTTP tunnel, we proposed a new hidden channel build-up method suitable for attestation scheme. The new method keeps the balance between information embedding capability and the channel exposure risk based on different type of applications. In order to embed the covert message on the target application, we also achieve the dynamic code injection procedure by using the DexClassLoader.

# Chapter 5 Distinguishing apps using HTTP traffic analysis

## 5.1 Introduction

Although the new attestation solution can prevent the current attack methods mentioned in the previous chapter, we cannot ensure the new attack model would not be proposed to break our attestation scheme in the future. Thus, it is necessary to bring in an ability to identify unknown attacks. We add an analysis phase behind the detection phase in order to double-check the application and identify unknown attacks.

During the communication between the server and the client, HTTP traffic is one of the information that can be stored in the server-side. In this chapter, we will focus on analyzing the similarity (distance) of HTTP traffic sets to distinguish repackaged apps and original apps.

We assume that the traffic sets have already passed the attestation phase. For each app, we collect the HTTP traffic sets generated by original apps as an sample and store them in the server. In the analysis phase, we calculate the distances between collected HTTP traffic sets and original HTTP traffic sets which stored in the server. Once the distances are abnormal, we judge the app which communicating with our server as a repackaged or totally different app.

Roberto [46] proposed a method to calculate the structural similarities(distances) between two HTTP requests in 2010, but he did not state how to calculate the distances between two HTTP traffic sets. In 2015, Xueping [47] combined this method with Hungarian algorithm in order to calculate the structural distances between two sets of HTTP traffic. However, we find some disadvantages of this algorithm(Chapter 4.3) , thus we need to design an experiment to find the most efficiency and accuracy method to distinguish the HTTP traffic sets generated by repackaged apps and original apps.

In this chapter, we build up our experiment environment and discuss the algorithms of calculating HTTP traffic set distances based on the experiment results. In the end, we design a proper analysis process to distinguish apps and identify unknown attacks.

## 5.2 Related techniques

### 5.2.1 HTTP traffic distance

In 2010, Roberto [46] proposed a novel network level behavioral malware clustering system. They focus on analyzing the structural similarities among malicious HTTP traffic traces generated by executing HTTP-based malware. Later on, the method of calculating similarities of HTTP traffics is also used for detecting repackaged Android applications in large-scale market [47].

In order to capture the features between HTTP traffics, they define a measure of distance (1-similarity) between two HTTP requests $r_k$ and $r_h$ generated by two different softwares. The Figure 5-1 shows the divided structure of HTTP requests.

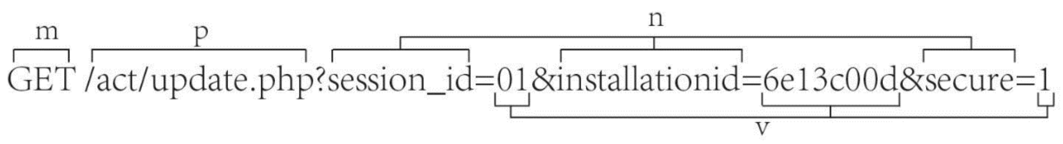

m
p
n
GET /act/update.php?session_id=01&installationid=6e13c00d&secure=1
v

**Figure 5-1 Structure of HTTP request**

- *m* represents *request method*(e.g., GET, POST). They define a distance function $d_m(r_k,r_h)$ that is equal to 0 if the requests $r_k$ and $r_h$ use the same method (i.e., both are GET method), otherwise the distance equal to 1.
- *p* stands for *page and path,* namely the first part of the URL that includes the path and page name. We define the distance $d_p(r_k,r_h)$ that is equal to the normalized Levenshtein distance [48] between these two part of strings that appear in the requests $r_k$ and $r_h$.
- *n* represents the *set of parameter names*(i.e., n={session_id, installationid, secure} in Figure 5-1 ). We define the distance $d_n(r_k,r_h)$ as the Jaccard distance [49] between the sets of parameter names in the requests $r_k$ and $r_{h.}$

- *v* represents the *set of parameter values*(i.e., 016e13c00d1). We define $d_v(r_k,r_h)$ equals to the normalized Levenshtein distance between strings obtained by concatenating the parameter values from these two requests.

They defined the overall distance between two requests as Formula 1 :

$$d_r(r_k,r_h) = w_m \cdot d_m(r_k,r_h) + w_p \cdot d_p(r_k,r_h) + w_n \cdot d_n(r_k,r_h) + w_v \cdot d_v(r_k,r_h) \qquad (1)$$

In this formula, the factors $w_x$ , $x \in \{m, p, n, v\}$ are predefined weights. After several times evaluations, the factors are assigned as $w_m = 10$, $w_p = 8$, $w_n = 3$, $w_v = 2$ [47].

By using the methods mentioned above, we can easily calculate the distances between two different HTTP requests(traffics). However, we need to define and calculate the distances between two different HTTP traffic sets in order to distinguish apps.

### 5.2.2 Hungarian algorithm

1) Hungarian algorithm in assignment problem

The Hungarian algorithm is a combinatorial optimization algorithm that solves the assignment problem in polynomial time [50]. The algorithm was published in 1955 by Harold Kuhn, who gave the name ”Hungarian method” because the algorithm was based on some early works of two Hungarian mathematicians.

Generally speaking, the Hungarian algorithm is designed to solve the assignment problem, thus we explain the assignment problem by a simple example as follow.

We consider an example [51] where four jobs (J1, J2, J3, J4) need to be executed by four workers (W1, W2, W3, W4), one job per worker. The table 5-1 below shows the cost of assigning a certain worker to a certain job. The objective is to minimize the total cost of the assignment.

|  | J1 | J2 | J3 | J4 |
|---|---|---|---|---|
| W1 | 82 | 83 | 69 | 92 |
| W2 | 77 | 37 | 49 | 92 |
| W3 | 11 | 69 | 5 | 86 |
| W4 | 8 | 9 | 98 | 23 |

**Table 5-1 Example cost in assignment problem**

There are four steps to solve this problem by using Hungarian algorithm respectively, Subtract row minima, Subtract column minima, Cover all zeros with a minimum number of lines and Create additional zeros. In the end the solution is: Worker 1 perform job 3, worker 2 performs job2, worker 3 performs job 1 and worker 4 performs job 4. The total cost of the optimal assignment is to 69+37+11+23=140.

2) Hungarian algorithm in distances of HTTP traffic sets

Although Hungarian algorithm is designed for assignment problem, it is also used to calculate the distances between two different HTTP traffic sets [47]. They defined two primary HTTP flow sets, P1 and P2, and their distance is the minimal value among the similarity of all HTTP flows. Comparing to the example in the assignment problem, P1 and P2 correspond to jobs and works and each distance of two HTTP flows corresponds to the cost of assigning a certain job to a certain worker.

They define HTTP requests rk and rh generated by two different apps. Thus the distance between two sets P1, P2 shows as follow.

$$d_p(P1, P2) = \frac{\sum_{(r_k, r_h) \in match(P1,P2)} d(r_k, r_h)}{|P1| + |P2|}$$

They distinguish similar apps in the app markets by using this definition of distance with proper threshold. The main purpose of their experiment is to distinguish the similar apps with other apps which are totally different, thus the accuracy is not highly required in their case. However, we need to distinguish original apps with repackaged apps which have highly similarities. We state the disadvantages of the current methods in chapter 5.3 and design an experiment to find a properly method in chapter 5.4.

## 5.3 Disadvantages of current method

As we can see from chapters above, the previous studies have already introduced the method to calculate the distance between two HTTP traffic sets by using Hungarian algorithm. However, they did not explain the reason and drawback of using Hungarian algorithm.

Since the Hungarian algorithm was designed to solve the assignment problem, we can model our traffic sets into a weighted balanced bipartite graph as Figure 5-2. The blue circles represent the requests inside the HTTP traffic set 1, the red circles represent the requests inside the HTTP traffic set 2 and the weighted lines between different circles represent the HTTP distances between these two requests.

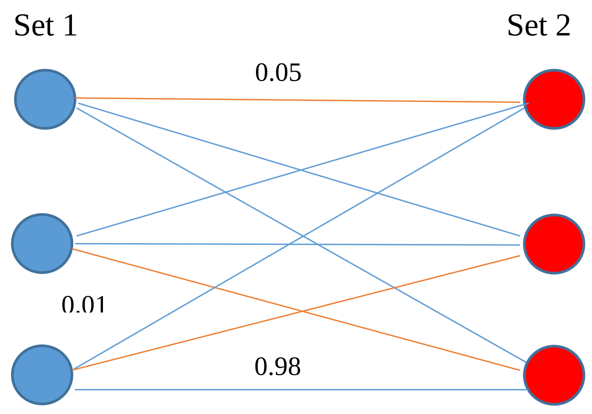


**Figure 5-2 Example of weighted balanced bipartite graph**

Before the calculation of distance, we assume that each request should find a match inside the opposite set and each request should be used only once. By using the algorithm of Hungarian, we can calculate a certain matches combination with minimal sum of all selected edges. Xueping Wu [47] set the average of this minimal sum as the distance between different HTTP traffic sets. However, there are several disadvantages by using this algorithm.

First of all, the amount of test set and original set should be exactly the same based on the operational condition of Hungarian algorithm. However, the size of HTTP traffics in test set could be different in each time. We can adjust the size of original set to adapt the changing of test set, but the accuracy of the distance will be unstable since we cannot ensure the quality of adjusted original set.

Second, the request can be matched only once based on the operational condition of Hungarian algorithm. Thus, the accuracy of HTTP traffic sets distance will be influenced by repeat or similar requests in the traffic sets.

| | Set 2 request 1 | Set 2 request 2 | Set 2 request 3 |
|---|---|---|---|
| Set 1 request 1 | 0.05 | 0.99 | 0.99 |
| Set 1 request 2 | 0.99 | 0.99 | 0.01 |
| Set 1 request 3 | 0.06 | 0.99 | 0.99 |

**Table 5-2 Distances between different requests**

Considering the situation as the requests distances in Table 5-2 above, we match the edges based on the result of Hungarian algorithm with red color in the table. The test set request 1 matches the original set request 1, the test set request 2 matches the original set request 3 and test set request 3 matches the original set request 2. According to the result of Hungarian algorithm, this is the best match with minimal sum of all edges. However, the request 3 in test set matches the original request 2 with 0.99 distance instead of the original set request 1 with 0.06 distance. The test set request 1 and test set request 3 both have low distance with the original set request 1, but the test set request 3 can't match the original set request 1 since the original set request 1 can be matched only once.

The situation described above will increase the distance between two different sets and cause mistakes when judging the repackaged apps. To avoid this situation, we cluster the similar requests before sending them into Hungarian algorithm or directly look up the lowest distance in the original set for each request of test set.

Third, the absolute threshold value can be influenced by different types of app. In the previous paper [47], the main purpose is to distinguish the potential repackaged apps pair in 7619 apps traffic set. They set the threshold 0.3 as the boundary of the repackaged apps with totally different apps. However, we are aim to identify the original app traffic with repackaged app traffic. Since the threshold of Hungarian distance depends on how much the codes has changed and we have the certain original traffic set in our server, we may set a relative Hungarian distance threshold instead of the absolute threshold.

In this chapter, we state three main disadvantages of current methods and purpose some concept ideas to improve them. Thus, we design an experiment to find the proper algorithm which can accurately distinguish repackaged apps and original apps.

## 5.4 Experiment

Since the previous method has several potential weakness, an experiment is necessary to improve the method. This experiment is designed to find the most efficiency and accuracy method to distinguish the HTTP traffics generated by repackaged apps and original apps. In the experiment, we collect HTTP traffic sets by simulating normal users' operation in different apps. The HTTP traffic sets are calculated by different methods and the thresholds are settled based on both absolute and relative way.

### 5.4.1 Experiment environment design

In this experiment, the prepared 10 apps are installed on Samsung Galaxy Alpha (SM-G850Y). We operate the apps as normal user and collect the HTTP traffic between smartphone and server. Since most of the apps are using HTTPs to protect the data, the normal sniff software like Wireshark [52]cannot collect the real traffics directly. Thus, we use Fiddler [53] as a HTTP/HTTPS proxy to collect the traffic we need.

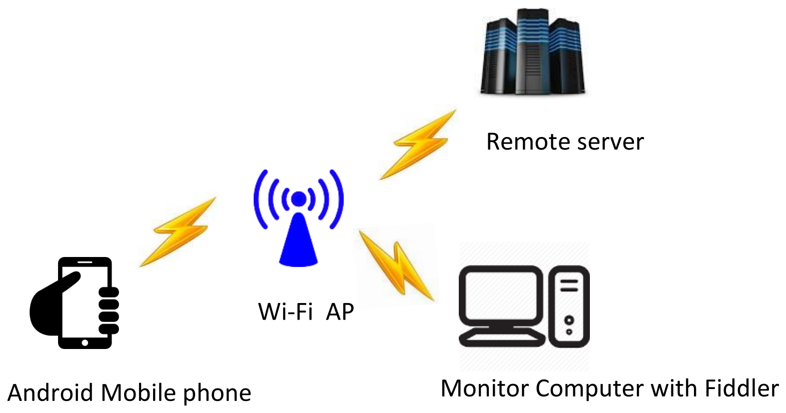


**Figure 5-3 Network topology**

As we can see from the Network topology Figure 5-3, the Android Mobile phone and Monitor computer with Fiddler are connected to the same Wi-Fi AP. Then we configure the equipment to collect the HTTPs traffic data communicating between Android Mobile phone and Remote server.

Firstly, we open Fiddler and choose “Tools->Fiddler Options…”. After that, we check “Capture HTTPS CONNECTs” and “…from remote clients only” in Tab “HTTPS”. In the next step, we set “Fiddler listens on port” as 8888 and check “Allow remote computers to connect” in Tab “Connections”.

Secondly, we configure the WLAN on the Android Mobile phone by adding proxy on the Connecting AP. The proxy IP address is the IP address of Monitor Computer and the port is “8888” as we set in the Fiddler before.

At last, we type the address “IP address:8888” in the browser of Mobile phone and download the Fiddler certificate. After the Mobile phone trusting the Fiddler certificate, all the HTTP/HTTPs traffic go through this phone can be monitored on the computer.

**Analysis**

Since the traffic of Mobile phone will go through the proxy as we designed in the Network topology, Fiddler cannot decrypt the traffic without the private key. However, the proxy can perform a successful middle man if the mobile phone trust the certificate from the proxy in advance. The procedures are shown below.

1. The Mobile phone sends a TLS handshake request to the remote server.
2. The remote server sends a certificate with server's public key back; The Fiddler intercepts the certificate and replaces it with its own certificate.
3. Since the mobile phone trusts the Fiddlers' certificate as we set before, it sends back the symmetric session key using the Fiddlers' public key.
4. The Fiddler intercepts the symmetric session and decrypt it by using its private key and send another acknowledgement encrypted with the session key to start the encrypted session with Mobile phone.
5. Since the Fiddler had the certificate from server side, it can pretend to be the phone and start the encrypted session with server.
6. From now on, the Fiddler gets the symmetric keys of both Mobile phone side and Server side. It can perform the MITM and get all the decrypted traffics.

By using this method, the Fiddler can perform a MITM attack and monitor all the HTTP/HTTPS traffics between Mobile phone and Server as Figure 5-4.

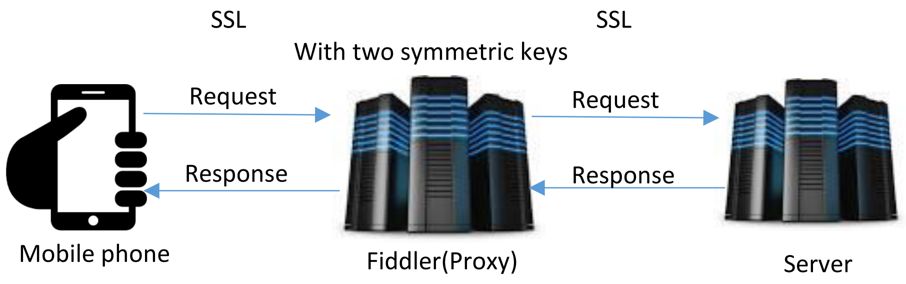


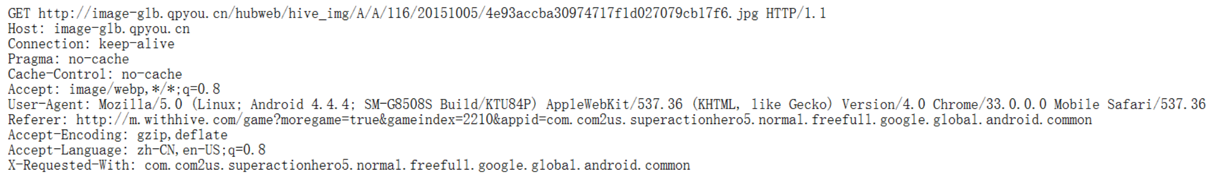
**Figure 5-4 Collect data by using MITM attack with proxy**

### 5.4.2 Experiment design

#### a. Data sets

We downloaded 5 original apps from google play (http://play.google.com/) respectively, angry bird, angry grandma run, eweather, ifighter and superhero. In the meanwhile, we downloaded the repackaged version of these apps in the third-party android market (http://www.mumayi.com/ and http://www.anzhi.com/). According to the introduction of these repackaged apps, most of the advertisements have already been blocked and all the functions are free to use.

We collect 100 HTTP requests each app as a HTTP traffic set. The collection of HTTP traffic sets repeat 3 times for each app. Thus, we collect 30 HTTP traffic sets with 3000 HTTP requests in total. One of the request generated by original app superhero is shown below as an example.



```
GET http://image-glb.qpyou.cn/hubweb/hive_img/A/A/116/20151005/4e93accba30974717f1d027079cb17f6.jpg HTTP/1.1
Host: image-glb.qpyou.cn
Connection: keep-alive
Pragma: no-cache
Cache-Control: no-cache
Accept: image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Linux; Android 4.4.4; SM-G8508S Build/KTU84P) AppleWebKit/537.36 (KHTML, like Gecko) Version/4.0 Chrome/33.0.0.0 Mobile Safari/537.36
Referer: http://m.withhive.com/game?moregame=true&gameindex=2210&appid=com.com2us.superactionhero5.normal.freefull.google.global.android.common
Accept-Encoding: gzip,deflate
Accept-Language: zh-CN,en-US;q=0.8
X-Requested-With: com.com2us.superactionhero5.normal.freefull.google.global.android.common
```

**Figure 5-5 One of the request generated by original app superhero**

#### b. Methods

To set up a contrast experiment, we use 3 different methods to calculate the distances of HTTP traffic sets.

**Hungarian algorithm**

As we introduced in the Chapter 4.2, Hungarian algorithm was designed for assignment problem. By using this algorithm, the amount of two comparing sets should be the same. We need to keep the same amount requests in two sets by cutting the longer set. With the same amount of test HTTP traffic set and original HTTP traffic set, we can directly use the original Hungarian algorithm to calculate the distances.

**Clustered Hungarian algorithm**

Since the similar requests inside the HTTP traffic sets can cause the instability as we discussed in the previous chapter, we can cluster the HTTP traffic set before we send it into Hungarian algorithm.

In the first step, we cluster the test HTTP traffic set by different threshold 0.1 and 0.05. The requests will be deleted if any of their distances with other requests inside the HTTP traffic set lower than the threshold.

Further on, we count the amount of new test HTTP traffic set and cut the original HTTP traffic set for the same length. The new test and original HTTP traffic sets will be processed by Hungarian algorithm in the final step.

**Exception algorithm**

To avoid the exception situation in HTTP distances comparison, we calculate both average exception distance and top 5 exception distance.

We define the distance $R_b$ between a single request R with request set B as the smallest distance value between R with the requests inside set B.

Thus, the average exception distance between two request sets A and B is defined as $(R_{1b}+R_{2b}+\ldots R_{nb})/n$ which $R_1$, $R_2$,..$R_n$ are all the requests in Set A. Similarly, the top 5 exception distance is defined as the average of top 5 value in the group $R_{1b}+R_{2b}+\ldots R_{nb.}$

**c. Experiment procedure**

For each of the original-repackage app pairs, we have 3 HTTP traffic sets generated by original version app and 3 HTTP traffic sets generated by repackaged version. They are stated as original 1, original 2, original 3, repackaged 1, repackaged 2 and repackaged 3 respectively.

At the first stage of the experiment, we choose the pairs of set generated by the same original version app, specifically original1-original2, original1-original3, and original2-original3. The distances of these pairs will be calculated by Hungarian algorithm, Clustered Hungarian algorithm(threshold 0.05 and 0.1), average exception and top 5 exception algorithm.

Secondly, we choose 3 pairs of HTTP traffic sets to compare the distances between original sets and repackaged sets, specifically original1-repackaged1, original2-repackaged2 and original3-repackaged3. The distances will be calculated by the same methods as we mentioned above.

After collecting the raw distances data, we analyze the performances of all kinds of distances calculation methods and compare the efficacy between absolute threshold and relative threshold.

## 5.5 Result

We get 5 pairs app raw data after calculating the distances between HTTP traffics by using the methods we mentioned in the chapter 5.4. For each of the Repackaged-Original app pair, we collect an Original-Original comparison table and a Repackaged-Original comparison table. Both tables include different data calculated by Hungarian algorithm, Average exception distance, Top 5 exception distance, Cluster Hungarian with 0.1 threshold and Cluster Hungarian with 0.05 threshold. One of the app pair raw data is shown below in Table 5-3 and Table 5-4 as example.

| **Ifighter Original** | **Set1/Set2** | **Set2/Set3** | **Set3/Set1** |
|---|---|---|---|
| **Hungarian** | 0.09670 | 0.0510 | 0.1060 |
| **Average Exception** | 0.0112 | 0.0305 | 0.0139 |
| **Top 5 Exception** | 0.1511 | 0.2871 | 0.1536 |
| **Clustering Hungarian (0.1)** | 0.0666 | 0.1290 | 0.0844 |
| **Clustering Hungarian (0.05)** | 0.0613 | 0.1953 | 0.1351 |

**Table 5-3 (lfighter) Distances calculation between different data set within original app by using different methods**

| Ifighter Original/Re-packaged | Original/Repackaged Set 1 | Original/Repackaged Set 2 | Original/Repackaged Set 3 |
|---|---|---|---|
| Hungarian | 0.2706 | 0.3157 | 0.3409 |
| Average Exception | 0.1341 | 0.1539 | 0.1557 |
| Top 5 Exception | 0.4137 | 0.4149 | 0.4211 |
| Clustering Hungar-ian (0.1) | 0.3635 | 0.3941 | 0.3453 |
| Clustering Hungar-ian (0.05) | 0.3302 | 0.3892 | 0.3559 |

**Table 5-4 (lfighter) Distances calculation between original data sets and repackaged data sets by using different methods**

As we can see from the raw data of Ifighter app above, the distance between Original set and Repackaged set are higher than the distance between two Original sets. After looking up the main different between these two HTTP traffic sets, we find that almost all the requests involving payment and advertisement are not exist in the repackaged set. It means that the repackaged app banned the advertisement and payment procedure of this app. One of the request involving payment in the original set is shown below. We cannot find the similar request in the repackaged set.

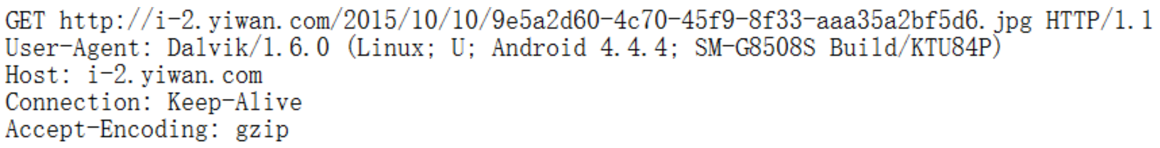

```
GET http://i-2.yiwan.com/2015/10/10/9e5a2d60-4c70-45f9-8f33-aaa35a2bf5d6.jpg HTTP/1.1
User-Agent: Dalvik/1.6.0 (Linux; U; Android 4.4.4; SM-G8508S Build/KTU84P)
Host: i-2.yiwan.com
Connection: Keep-Alive
Accept-Encoding: gzip
```

**Figure 5-6 Payment request in Original app Ifighter**

## Single Index performance

Based on the raw data, we compare the distances Original sets-Repackaged sets with the Original sets-Original sets within a same distance algorithm. The comparison based on Hungarian algorithm within Ifighter app pair is shown in Table 5-5 as an example.

| Hungarian Algorithm | Data 1 | Data 2 | Data 3 |
|---|---|---|---|
| Ifighter Original-Original | 0.0967 | 0.0510 | 0.1060 |
| Ifighter Original-Repackaged | 0.2706 | 0.3157 | 0.3409 |

**Table 5-5 Distances calculated by Hungarian algorithm as single index performance**

According to the comparison results of five pairs apps, several flow charts are drawn to analyze the performances of different algorithms. We define the expected algorithm as the method which can distinguish repackaged apps traffic and original apps traffic by obvious difference of distance values.

**Hungarian Algorithm**

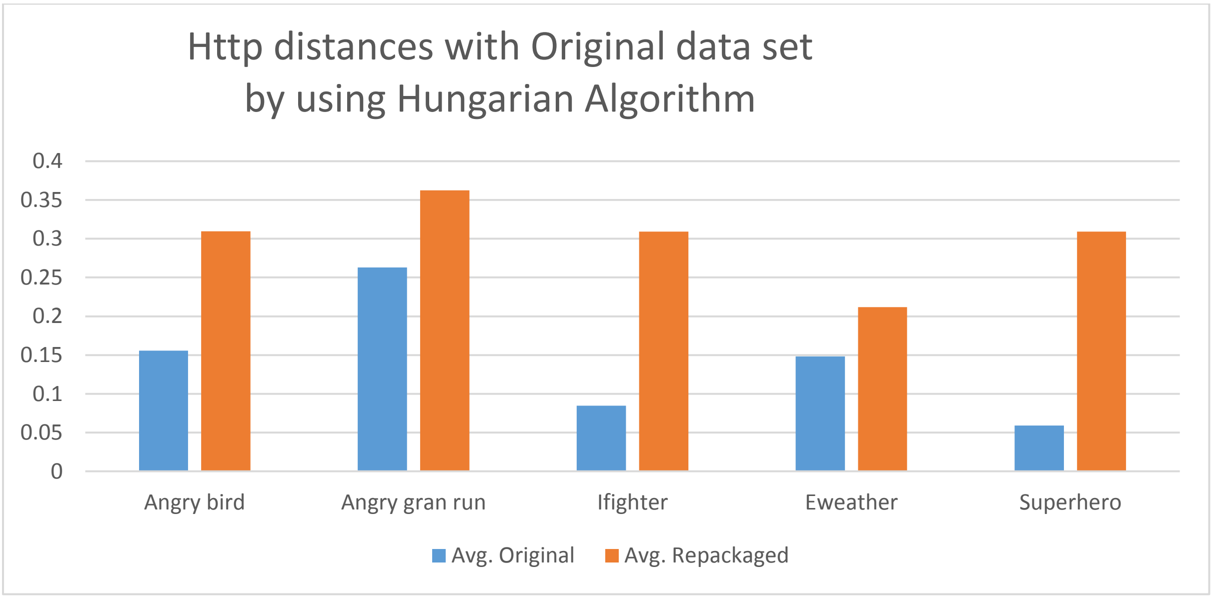


**Figure 5-7 HTTP distances with original data set by using Hungarian algorithm**

The Figure 5-7 shows the five pairs of apps' HTTP traffic distances calculated by Hungarian algorithm. We can see all the repackaged HTTP traffics have larger distances than the original traffics. The app Eweather has the most similar distances pair of original version and repackaged version, but the gap 0.06 is big enough to distinguish them.

However, the distance of Original version Angry gran run is higher than the distance of Repackaged version Eweather. It means that any absolute threshold to distinguish original and repackaged version in this case will lead a mistake.

**Clustering Hungarian**

Before the analysis the Clustering Hungarian performance, we compare the data with 0.1 clustering threshold and 0.05 clustering threshold. The result shows that threshold set as 0.1 will be stable than 0.05. Thus the analysis of  Clustering Hungarian is based on the clustering threshold 0.1.

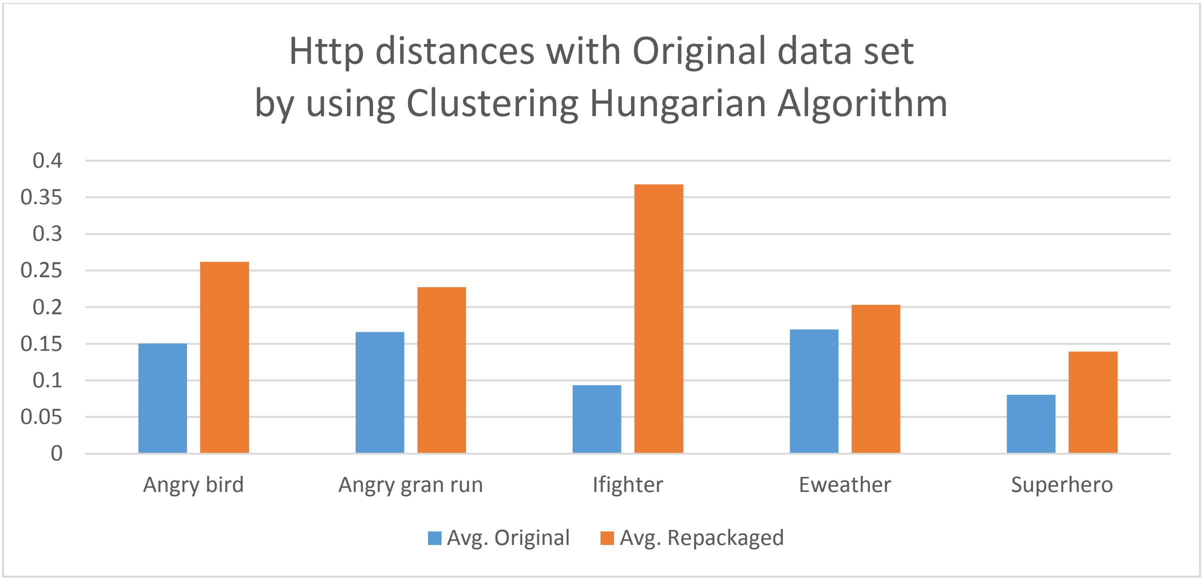


**Figure 5-8 HTTP distances with original data set by using clustering Hungarian algorithm**

The Figure 5-8 above shows the five pairs of apps' HTTP traffic distances calculated by Clustering Hungarian (0.1) algorithm. Although all of the average HTTP distance of repackaged version is higher than the original version, the lowest gap is only 0.03. The gap is not big enough to distinguish them.

In the experiment, the clustering Hungarian algorithm is designed to improve the Hungarian algorithm by fixing the problem that unequal amounts of similar requests inside test sets and original sets. We assumed that the clustering will fix the incorrect increased distance caused by the problem. It means that all the distances calculated by clustering Hungarian algorithm should be lower and more stable than Hungarian algorithm. However, Although most the distance values calculated by clustering Hungarian algorithm are lower than the values calculated by Hungarian algorithm, we find some special cases after analyzing.

| Ifighter Original/Re-packaged | Original/Repack-aged Set 1 | Original/Repackaged Set 2 | Original/Repackaged Set 3 |
|---|---|---|---|
| **Hungarian** | 0.2706 | 0.3157 | 0.3409 |
| **Clustering Hungarian (0.1)** | 0.3635 | 0.3941 | 0.3453 |

**Table 5-6 Example of raw data**

As we can see from the Table 5-6, the clustering Hungarian algorithm get the distances even higher than Hungarian algorithm. This result is unexpected so we decide to analyze the raw request sets to find reason.

After analyzing the raw HTTP requests from the app ifighter, we figured out that the clustering will not only decrease the distance by fixing the unequal similar requests problem, it can also increase the distance when the clustered requests have really low distance pairs in original set. Assume that we have four similar requests in test set and they can find the exactly same requests in the original set. The overall distance will be decreased because three of them with 0 distance are deleted by clustering.

Thus, the experiment shows the result of clustering Hungarian algorithm depends on the properties of data set and it cannot improve the Hungarian algorithm for calculating the distance as a single index.

**Average Exception**

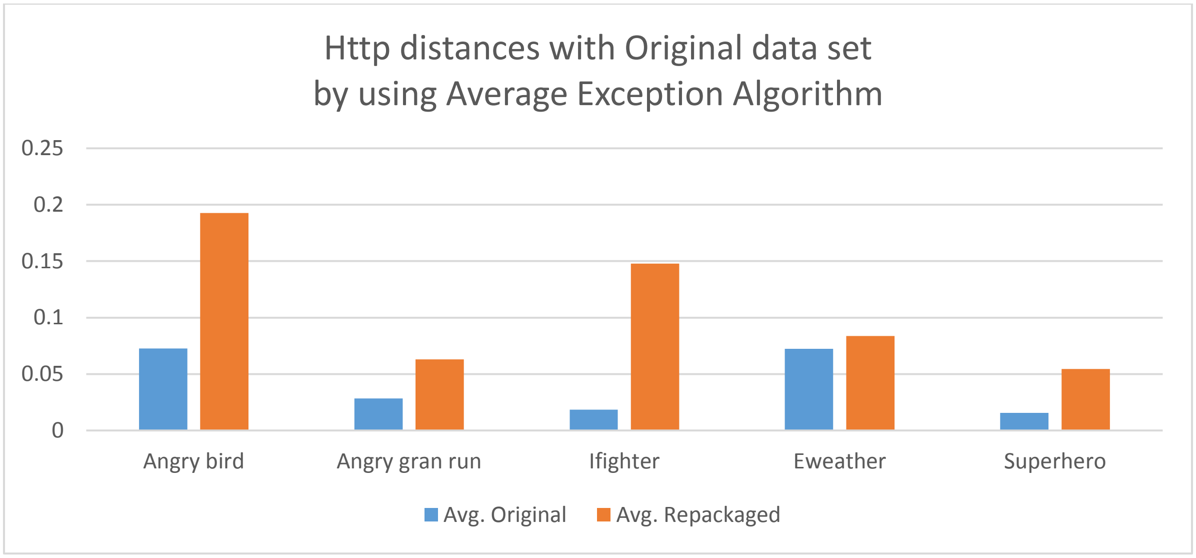


**Figure 5-9 HTTP distances with original data set by using average exception algorithm**

The Figure 5-9 shows the five pairs of apps' HTTP traffic distances calculated by Average Exception algorithm. It is obviously that all the repackaged HTTP traffics have larger distances than the original traffics. Unfortunately, the closest gap of HTTP traffic distances between repackaged version and original version is merely 0.01 in the app Eweather.

Furthermore, some of the distances of Avg. original version apps like Angry bird and Eweather are higher than distances of repackaged versions of apps like Superhero and Angry gran run. Thus, it is not possible to set a perfect threshold to distinguish repackaged apps and Original apps by the single index Average Algorithm.

**Top 5 Exception**

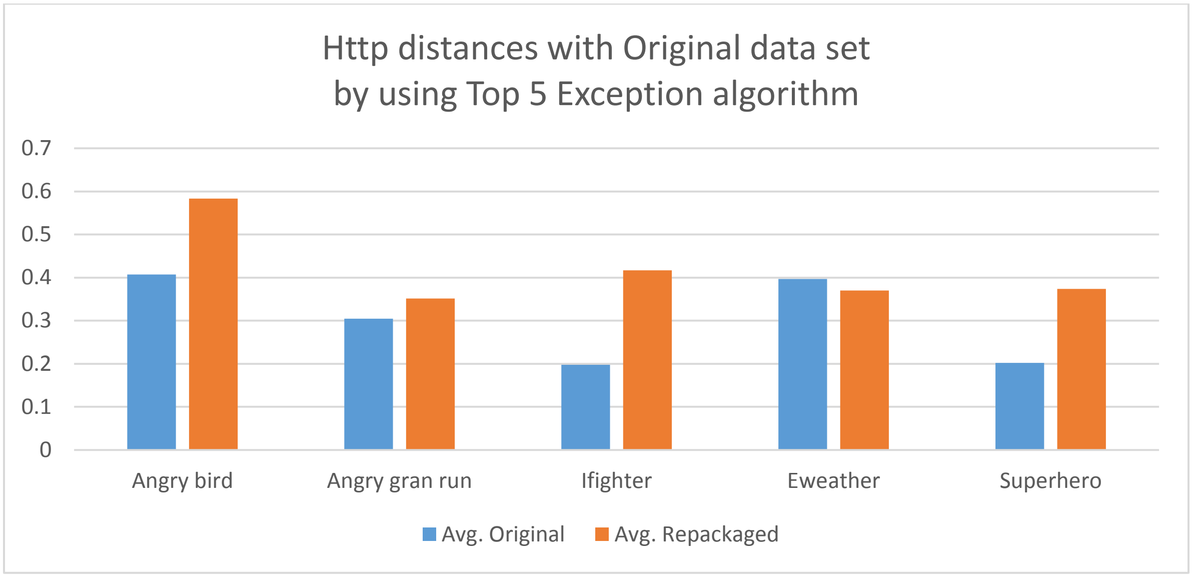


**Figure 5-10 HTTP distances with original data set by using Top 5 exception algorithm**

The five pairs of apps' HTTP traffic distances calculated by Top 5 Exception algorithm are shown in Figure 5-10. Unfortunately, the data are unstable to distinguish the repackaged apps with original apps. The Original HTTP traffic of Eweather has even larger distance than the Re-packaged HTTP traffic.

We assume it happens because the different type of apps have different method to communicate with server. Thus we consider to leave this index out of our judgement.

After we analyzing five different HTTP traffic sets distance algorithms, the data shows several results listed below.

1. The clustering Hungarian algorithm and exception algorithm are not better than the Hungarian Algorithm as we thought.
2. We cannot find an absolute threshold to distinguish repackaged apps and original apps when we use any of the algorithms as an single index.
3. The changing of the distances are influenced by the properties of apps' requests(e.g. repeat degree, writing style of communication module).
4. The top 5 Exception is not a good index to consider in our case.

**Relative threshold and multiple indexes**

In the previous paper [47], the main purpose was to distinguish the potential repackaged apps pair in 7619 apps traffic set. They set the threshold 0.3 as the boundary of the repackaged apps with totally different apps. However, we aim to identify the original app traffic with repackaged app traffic. We have original set of apps to compare with in the server.

Besides, the experiment does show that an absolute threshold cannot be set perfectly based on the results. Thus we decide to distinguish the HTTP traffic of repackaged apps and original apps by using relative threshold. We also investigate to use multiple indexes to increase the accuracy.

After several times evaluation, the threshold is set as follows:

1. The test set is judged as repackaged HTTP traffic set if the distance between test set and original set calculated by 1). Hungarian algorithm is larger than the Avg. distances between original sets+0.01; and 2). Clustering Hungarian algorithm is larger than the Avg. distances between original sets+0.01; and 3). Average exception algorithm is larger than the Avg. distances between original sets+0.01;
2. The test set is judged as repackaged HTTP traffic set if the distance between test set and original set calculated by 1). Hungarian algorithm is larger than the Avg. distances between original sets+0.05; or 2). Clustering Hungarian algorithm is larger than the Avg. distances between original sets+0.05; or 3). Average exception algorithm is larger than the Avg. distances between original sets+0.03.

To compare the related threshold with absolute criteria, 30 collected data sets (15 original version HTTP sets and 15 repackaged version HTTP sets) are used in the test. The results are shown in the Table 5-7 and Table 5-8.

| Relative threshold | Multiple Indexes | Hungarian Index | Clustering Hungarian Index | Average Exception Index |
|---|---|---|---|---|
| False Positives | 1 | 0 | 1 | 1 |
| True Positives | 15 | 13 | 10 | 10 |

**Table 5-7 Relative threshold by using different kinds of indexes**

| Absolute threshold Hungarian Index | Threshold 0.05 | Threshold 0.10 | Threshold 0.15 | Threshold 0.20 |
|---|---|---|---|---|
| False Positives | 14 | 7 | 2 | 0 |
| True Positives | 15 | 15 | 14 | 11 |

**Table 5-8 Absolute threshold by using Hungarian algorithm as the index**

As we can see from the tables above, the relative threshold with Multiple Indexes can identify all the 15 repackaged apps and have only one false positive. It is better than the relative threshold with single index or best result absolute threshold 0.15 with two false positives and one false negative.

After analyzing the results of our experiment, we conclude that the usage of relative threshold with multiple indexes can distinguish repackaged apps with original apps accurately and efficiently comparing to other methods.

In this chapter, a HTTP traffic analysis module is built in order to double-check the application and identify unknown attack. The HTTP traffic analysis is based on the calculation of the distance between two HTTP traffic sets to distinguish two different apps.

In order to find the efficiency and accuracy method to distinguish the HTTP traffics generated by repackaged apps and original apps, we designed an experiment to collect and analyze data. After analyzing the result, we conclude that the usage of relative threshold combine with multiple indexes which includes Hungarian index, Clustering Hungarian index and Average exception index can distinguish repackaged apps accurately and efficiently comparing to other methods. By using the new analysis method, the obvious difference from the HTTP traffic sets distance between repackaged app and original app can be observed.

# Chapter 6 Conclusion

In this thesis, we focused on software based server-side verification to detect the repackaged apps. Based on the background of reverse engineering technique and current preventing techniques, we proposed a two-phase model by improving the existing detection scheme and adding an analysis phase.

The improved attestation scheme keeps the advantages of unpredictable attestation scheme, in the meanwhile, it ensures the execution environment of attestation module by using hidden channel and dynamic code injection techniques. Thus, it can prevent the current known attacks especially dummy folder attack. An example of feasible HTTP hidden channel is HTTP header manipulation. Furthermore, we demonstrated a demo about implementing dynamic code injection technique by Dexclassloader.

As for the varietal attacks, the HTTP traffic sets distances can be used to identify the repackaged app. The analysis of current methods for calculating HTTP traffic sets distance shows that it can be improved to distinguish different applications.  An experiment we designed and implemented shows the feasibility of the approach. Based on the experiment, we concluded that the usage of relative threshold combined with multiple indexes (Hungarian index, Clustering Hungarian index and Average exception index) can distinguish repackaged applications accurately and efficiently comparing to other methods. Thus, by using the new analysis method, we can find the differences between repackaged app and original app.

Since the current attestation methods cannot efficiently prevent particular attacks such as dummy folder attack, we proposed a new attestation scheme which improves current status and prevents existing attacks. In additional,  a HTTP traffic analysis method was designed in order to prevent unknown attacks which can be reflected in HTTP traffic sets.  Overall, the two-phase model we proposed in this thesis can be used by the server to detect repackaged applications effectively and efficiently.

There are also several open questions left in the thesis. We proposed a concept of keeping the balance between information embedding capability and channel exposure risk in hidden channel, the implementation and detail study could be an interesting topic in the future. Additionally, the HTTP header manipulation is only an example of hiding information in the HTTP channel, there are still several potential methods that can be investigated in the future such as HTTP timing channel.

Furthermore, the HTTP traffic analysis experiment we did shows the feasibility of the improved approach, but the dataset collected are not big enough to ensure the accuracy of the threshold. In our experiment, we performed a MITM attack and manually simulated the regular users' action on the mobile phone and collect the data. Since we have demonstrated the feasibility of our approach, it will be valuable to develop an automated data collecting method in order to enlarge the dataset and ensure the accuracy of the threshold.

# References

[1] 2014. [Online]. Available: http://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/.

[2] V. Svajcer, "Sophos mobile security threat report.," Mobile World Congress, 2014.

[3] Zhou, Yajin, and Xuxian Jiang, "Dissecting android malware: Characterization and evolution," *Symposium on Security and Privacy. IEEE*, pp. 95-109, 2012.

[4] Jeong, Jihwan, et al., "MysteryChecker: Unpredictable attestation to detect repackaged malicious applications in Android," *Malicious and Unwanted Software: The Americas (MALWARE), 2014 9th International Conference on. IEEE*, pp. 50-57, 2014.

[5] Jung, Jin-Hyuk, Ju Young Kim, Hyeong-Chan Lee, and Jeong Hyun Yi. , "Repackaging attack on android banking applications and its countermeasures," *Wireless Personal Communications*, vol. 73(4), pp. 1421-1437, 2013.

[6] "UNDX," 2009. [Online]. Available: https://sourceforge.net/projects/undx/.

[7] "dex2jar," [Online]. Available: http://code.google.com/p/dex2jar/..

[8] "Lotcat," [Online]. Available: http://developer.android.com/guide/developing/tools/logcat.html.

[9] "An Assembler(smali) and disassembler(baksmali) for android dex format," [Online]. Available: http://code.google.com/p/smali/.

[10] Coker, George, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen., "Principles of remote attestation," *International Journal of Information Security*, vol. 10, no. 2, pp. 63-81, 2011.

[11] M. Kiperberg, "Preventing reverse engineering of native and managed programs," 2015.

[12] "Trusted Computing Group," [Online]. Available: http://www.trustedcomputinggroup.org/.

[13] L. Gu, X. Ding, R. H. Deng, B. Xie and H. Mei, "Remote attestation on program execution.," *Proceedings of the 3rd ACM workshop on Scalable trusted computing*, pp. 11-20, 2008.

[14] Li, Li, Hong Hu, Jun Sun, Yang Liu, and Jin Song Dong., "Practical analysis framework for software-based attestation scheme.," *International Conference on Formal Engineering Methods,* pp. 284-299, 2014.

[15] R. Harrison, "Investigating the Effectiveness of Obfuscation Against Android Application Reverse Engineering," Technical Report RHUL-MA-2015-7, Royal Holloway University of London, Surrey, UK, 2015.

[16] P. Schulz, "Code protection in android," *Insititute of Computer Science, Rheinische Friedrich-Wilhelms-Universitgt Bonn*, 2012.

[17] "ProGuard Java Obfuscator Manual," [Online]. Available: http://proguard.sourceforge.net/.

[18] A. Kovacheva, "Efficient code obfuscation for Android," *International Conference on Advances in Information Technology,* pp. 104-119, 2013.

[19] Chown, Pete, "Advanced Encryption Standard (AES), National Institute of Standards and Technology, US Department of Commerce," 2001. [Online]. Available: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf..

[20] Linn, Cullen, and Saumya Debray, "Obfuscation of executable code to improve resistance to static disassembly," *Proceedings of the 10th ACM conference on Computer and communications security*, 2003.

[21] E. Nylander, "Improved code obfuscation through automatic construction of hidden execution paths," 2014.

[22] "Android Open source project. Android SDK.," May 2012. [Online]. Available: http://developer.android.com/sdk/index.html.

[23] "DexGuard," [Online]. Available: https://www.guardsquare.com/dexguard.

[24] "Allatori," [Online]. Available: http://www.allatori.com/overview.html.

[25] "Dalvik-obfuscator," [Online]. Available: http://www.dexlabs.org/blog/bytecode-obfuscation.

[26] Huang, Heqing, Sencun Zhu, Peng Liu, and Dinghao Wu., "A framework for evaluating mobile app repackaging detection algorithms.," *International Conference on Trust and Trustworthy Computing,* pp. 169-186, 2013.

[27] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing.," *Digital investigation*, vol. 3, pp. 91-97, 2006.

[28] Li, Yuping, Sathya Chandran Sundaramurthy, Alexandru G. Bardas, Xinming Ou, Doina Caragea, Xin Hu, and Jiyong Jang., "Experimental study of fuzzy hashing in malware clustering analysis," *8th Workshop on Cyber Security Experimentation and Test (CSET 15)*, 2015.

[29] W. Zhou, "Repackaged Smartphone Applications: Threats and Defenses.," *North Carolina State University,* 2013.

[30] Crussell, Jonathan, Clint Gibler, and Hao Chen., "Attack of the clones: Detecting cloned applications on android markets," *European Symposium on Research in Computer Security.,* 2012.

[31] Zhou, Yajin, and Xuxian Jiang, "WALA," [Online]. Available: http://wala.sourceforge.net/wiki/index.php/..

[32] Chen, Jian, Manar H. Alalfi, Thomas R. Dean, and Ying Zou. , "Detecting Android Malware Using Clone Detection.," *Journal of Computer Science and Technology,* vol. 30, no. 5, pp. 942-956, 2015.

[33] Hanna, Steve, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song., "Juxtapp: A scalable system for detecting code reuse among android applications.," *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment,,* pp. 62-81, 2012.

[34] "Feature hashing," [Online]. Available: https://en.wikipedia.org/wiki/Feature_hashing.

[35] Collberg, Christian S., and Clark Thomborson, "Watermarking, tamper-proofing, and obfuscation-tools for software protection," *IEEE Transactions on software engineering,* vol. 28, no. 8, pp. 735-746, 2012.

[36] Zhang, Yingjun, and Kai Chen, "AppMark: A Picture-Based Watermark for Android Apps," *Software Security and Reliability (SERE),* 2014.

[37] S. A. Moskowitz and C. Marc, "Method for stega-cipher protection of computer code". U.S. Patent No. 5,745,569, 28 Apr. 1998.

[38] R. I. Davidson and M. Nathan, "Method and system for generating and auditing a signature for a computer program". U.S. Patent Patent No. 5,559,884, 24 Sep. 1998.

[39] Collberg, Christian, and Clark Thomborson, "Software watermarking: Models and dynamic embeddings," *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* 1999.

[40] Zhou, Wu, Xinwen Zhang, and Xuxian Jiang, "AppInk: watermarking android apps for repackaging deterrence," *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM, ,* 2013.

[41] Seshadri, A, Perrig, A, Van Doorn, L. & Khosla, P, "SWATT: Software-based attestation for embedded devices," *Security and Privacy, 2004. Proceedings,* pp. 272-282, 2004.

[42] Song, K, Seo, D. Park, H. Lee, H. & Perrig, A, "OMAP: One-way memory attestation protocol for smart meters," *Parallel and Distributed Processing with Applications Workshops (ISPAW),* pp. 111-118, 2011.

[43] Z. Kwecka, "Application layer covert channel analysis and detection," *Edinburgh Napier University,* 2006.

[44] Poeplau, Sebastian, et al, "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications," *NDSS,* vol. 14, 2014.

[45] Fielding, R, Gettys, J, Mogul, J, Frystyk, H, Masinter, L, Leach, P, & Berners-Lee, "RFC2616: Hypertext transfer protocol–HTTP/1.1. h ttp," 1999. [Online]. Available: org/html/rfc2186.

[46] Perdisci, Roberto, Wenke Lee, and Nick Feamster, "Behavioral Clustering of HTTP-Based Malware and Signature Generation Using Malicious Network Traces," *NSDI,* pp. 391-404, 2010.

[47] Wu, Xueping, et al, "Detect repackaged Android application based on HTTP traffic similarity," *Security and Communication Networks,* vol. 8, no. 13, pp. 2257-2266, 2015.

[48] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Soviet physics doklady,* vol. 10, 1966.

[49] "Jaccard distance," [Online]. Available: https://en.wikipedia.org/wiki/Jaccard_index.

[50] D. Bruff, "The assignment problem and the hungarian method.," *Notes for Math,* vol. 20.

[51] "Hungarian Algorithm," [Online]. Available: http://www.hungarianalgorithm.com/examplehungarianalgorithm.php.

[52] "Wireshark," [Online]. Available: https://www.wireshark.org/.

[53] "Fiddler," [Online]. Available: http://www.telerik.com/fiddler.

# Appendix A Implementation of HTTP traffic analysis

## A.1 Distance between two HTTP traffic sets using Hungarian algorithm

```
public static double solveAssignmentProblem(double[][] a) {
//   Distance between two HTTP traffic sets using Hungarian algorithm
//   Input a is the distances between different HTTP requests
      int n = a.length - 1;
      int m = a[0].length - 1;
      double[] u = new double[n + 1];
      double[] v = new double[m + 1];
      int[] p = new int[m + 1];
      int[] way = new int[m + 1];
      for (int i = 1; i <= n; ++i) {
            p[0] = i;
            int j0 = 0;
            double[] minv = new double[m + 1];
            Arrays.fill(minv, 10000);
            boolean[] used = new boolean[m + 1];
            do {
            used[j0] = true;
            int i0 = p[j0];
            double delta = 10000;
            int j1 = 0;
            for (int j = 1; j <= m; ++j)
                if (!used[j]) {
                    double cur = a[i0][j] - u[i0] - v[j];
                    if (cur < minv[j]) {
                          minv[j] = cur;
                          way[j] = j0;
                        }
                    if (minv[j] < delta) {
                          delta = minv[j];
                          j1 = j;
                        }
                      }
                    for (int j = 0; j <= m; ++j)
                      if (used[j]) {
                        u[p[j]] += delta;
                        v[j] -= delta;
                      } else
                        minv[j] -= delta;
                    j0 = j1;
            } while (p[j0] != 0);
            do {
                 int j1 = way[j0];
                    p[j0] = p[j1];
                    j0 = j1;
                  } while (j0 != 0);
      }
      return -v[0];
}
```

## A.2 Distance between two HTTP requests

```
public static double score(String str1,String str2){
// Calculate the distance between two Http requests
      double res=0;
      String m1,m2,p1,p2,n1,n2,v1,v2;
      m1="";m2="";v1="";p1="";p2="";n1="";n2="";v2="";
      int pos;
            pos=str1.indexOf(" ");
            if (pos!=-1){
                  m1=str1.substring(0,pos);//Get First Method string.
                  str1=str1.substring(pos+1);
            }else
                  m1=str1;

            pos=str2.indexOf(" ");
            if (pos!=-1){
                  m2=str2.substring(0,pos);//Get Second Method string.
                  str2=str2.substring(pos+1);
            }else
                  m2=str2;


            pos=str1.indexOf("?");
            if (pos!=-1){
                  p1=str1.substring(0,pos);//Get First Page string.
                  str1=str1.substring(pos+1);
            }
            else p1=str1;

            pos=str2.indexOf("?");
            if (pos!=-1){
                  p2=str2.substring(0,pos);//Get Second Page string.
                  str2=str2.substring(pos+1);
            }
            else p2=str2;

            pos=str1.indexOf("=");
            if (pos!=-1){
                  n1=str1.substring(0,pos);
                  str1=str1.substring(pos+1);
            }

            pos=str2.indexOf("=");
            if (pos!=-1){
                  n2=str2.substring(0,pos);
                  str2=str2.substring(pos+1);
            }
                  while(str1.indexOf("&")!=-1){
                        pos=str1.indexOf("&");
                        if (pos!=-1){
                              v1=v1+str1.substring(0,pos);
                              str1=str1.substring(pos+1);
                        }
                        pos=str1.indexOf("=");
                        if (pos!=-1){
                              n1=n1+" "+str1.substring(0,pos);
                              str1=str1.substring(pos+1);
                        }
                  }

                  while(str2.indexOf("&")!=-1){
                        pos=str2.indexOf("&");
                        if (pos!=-1){
                              v2=v2+str2.substring(0,pos);
                              str2=str2.substring(pos+1);
                        }

                        pos=str2.indexOf("=");
                        if (pos!=-1){
                              n2=n2+" "+str2.substring(0,pos);
                              str2=str2.substring(pos+1);
                              }
                        }

                  double m=1;
                        if (m1.equals(m2))
                              m=0;

                  double p=levenshteinDistance(p1,p2);
                  double n=jc(n1,n2);
                  double v=levenshteinDistance(v1,v2);
            res= (10*m+8*p+3*n+v)/22;
            return res;
      }
```

## A.3 Levenshtein Distance

```
public static double levenshteinDistance (CharSequence lhs, CharSequence
rhs) {
          int len0 = lhs.length() + 1;
          int len1 = rhs.length() + 1;
          double Maxlen=len1;
          if (len0>len1)
            Maxlen=len0;

          // the array of distances
          int[] cost = new int[len0];
          int[] newcost = new int[len0];

          // initial cost of skipping prefix in String s0
          for (int i = 0; i < len0; i++) cost[i] = i;

          // dynamically computing the array of distances

          // transformation cost for each letter in s1
          for (int j = 1; j < len1; j++) {
              // initial cost of skipping prefix in String s1
              newcost[0] = j;

              // transformation cost for each letter in s0
              for(int i = 1; i < len0; i++) {
                  // matching current letters in both strings
                  int match = (lhs.charAt(i - 1) == rhs.charAt(j - 1)) ? 0 :
1;

                  // computing cost for each transformation
                  int cost_replace = cost[i - 1] + match;
                  int cost_insert  = cost[i] + 1;
                  int cost_delete  = newcost[i - 1] + 1;

                  // keep minimum cost
                  newcost[i] = Math.min(Math.min(cost_insert, cost_delete),
cost_replace);
              }

              // swap cost/newcost arrays
              int[] swap = cost; cost = newcost; newcost = swap;
          }

          // the distance is the cost for transforming all letters in both
strings

          double re=cost[len0-1]/Maxlen;
          return re;
      }
```

## A.4 Jaccard distance

```
public static double jc(String s, String t)
    {
        String[] sSplit=s.split(" ");
        String[] tSplit=t.split(" ");

        //calculate intersection
        List<String> intersection=new ArrayList<String>();
        for(int i=0;i<sSplit.length;i++)
        {
            for(int j=0;j<tSplit.length;j++)
            {
                if(!intersection.contains(sSplit[i]))       //no duplicate
                    if(sSplit[i].equals(tSplit[j]))         //has intersec-
tion
                    {
                        intersection.add(sSplit[i]);
                        break;
                    }
            }
        }

        //calculate union
        List<String> union=new ArrayList<String>();
        if(sSplit.length>tSplit.length)                      //calculate big
tupple first
        {
            for(int i=0;i<sSplit.length;i++)
                if(!union.contains(sSplit[i]))
                    union.add(sSplit[i]);
            for(int i=0;i<tSplit.length;i++)
                if(!union.contains(tSplit[i]))
                    union.add(tSplit[i]);
        }
        else
        {
            for(int i=0;i<tSplit.length;i++)
                if(!union.contains(tSplit[i]))
                    union.add(tSplit[i]);
            for(int i=0;i<sSplit.length;i++)
                if(!union.contains(sSplit[i]))
                    union.add(sSplit[i]);

        }


        return 1-((double)intersection.size())/((double)union.size());
    }
```