An Empirical Study of Insecure Communication in Android Apps

Yueheng Zhang, Junliang Shu, Yuanyuan Zhang, Juanru Li, Qing Wang, and Dawu Gu

Computer Science and Engineering Department,
Shanghai Jiao Tong University, Shanghai, China

Abstract. Android apps rely on secure communication protocol to prove the confidentiality of sensitive data transmission. However, inexperienced developers tend to adopt insecure communication and introduce security risks. To study how prevalent the insecure communication protocols are used by real world Android apps, we conducted an in-depth analysis to examine popular apps from Google Play and MyApp Android app market. We collect 435 apps from major categories, such as gaming, shopping and social networks, and we monitored the communication of those apps and classified their used protocols into three categories: secure, insecure, and proprietary. Then we investigated those proprietary ones to find potential insecure implementation. We designed and implemented RawDroid, a protocol audit system combining network monitoring and program analysis technique to systematically inspect the security of proprietary protocol. We found that a large number of developers frequently use non-standard proprietary protocols. Among all analyzed apps, our security audit revealed that 36.7% apps adopted a proprietary protocol, and all those proprietary protocols fail to achieve confidentiality: some of them send sensitive data in the form of plain-text to servers; some misuse cryptographic algorithms and lead to the exposure of transferred privacy even if the content is encrypted. We believe this kind of protocols pose great security threats to Android ecosystem.

Keywords: Android apps; Proprietary protocol; Security; Program analysis

1 Introduction

With the coming of the information era, Android smartphone becomes more ubiquitous and pervasive. Many network services such as gaming, online shopping and social networks deploy their client apps for Android platform. These apps usually deliver user's privacy to servers. Although this client/server communication pattern provides convenience and flexibility to developers, it also introduces security-related issues. The confidentiality of transmitted privacy data heavily relies on the security of protocols. In general, developers must build robust protocols, which guarantee apps to communicate with their servers securely against attackers. To protect from being attacked through untrusted network, security best practices suggest deploying HTTPS protocol. Nonetheless, many apps still adopt security-by-obscurity policy to deploy their own proprietary protocols. Therefore, security risks are introduced.

To our knowledge, although the security analysis of common application layer protocols has been widely studied in previous works. There is no systematic research on analyzing the prevalence of Android apps' insecure communication, especially on proprietary protocols. Most of previous works rely on the fact that the format of analyzed protocol is well defined in network traffic. Unlike common protocols, however, the formats of proprietary protocols vary from one to another, which makes it difficult to be analyzed. In addition, many proprietary protocols employ encryption to hinder the analysis. Hence new analysis should be proposed to address these issues.

In this paper, we conducted a systematically study to investigate the situation of insecure network protocol usage in Android apps. We fulfilled the study in two steps. The first step of our work involves an automated testing to find the use of potential insecure communication (non-HTTPS of those apps. Then, we further analyze used proprietary protocols to audit the security. We propose RawDroid, a lightweight and semi-automatic protocol audity system to detect security flaws of proprietary protocols. RawDroid combines network traffic analysis and dynamic program analysis to locate potential insecure protocols. It hooks all key functions in system libraries related to network to record the invoking and the parameters of them. RawDroid also captures the network traffic and system status for later analysis during the execution. With the help of these run-time information, RawDroid determines whether an Android app uses proprietary protocol, and enhances manual analysis through pinpointing related functions in code to efficiently evaluate the security of the used protocols.

To validate RawDroid, we selected 435 apps involving entertaining, online shopping, and social network communicating from Google Play and MyApp Android app market. The analysis of RawDroid found that insecure communication (non HTTPS) are used by 363 apps (83.4%). Among them, 158 apps (36.7%) use proprietary protocols rather than common application layer protocols (e.g., HTTP, HTTPS) to transmit data. And we manually audited 60 apps using proprietary protocols with the help of RawDroid, finding that all proprietary protocols are insecure. We found 54 apps send sensitive data in plain-text, while six apps misuse cryptography to build insecure encryption schemes. The cases of cryptographic misuses include hard-coded key in their custom cryptographic algorithms, key transmitting in plaint-text with encrypted data. The experimental results show that the potential threats are severe and can lead to serious attacks, leaking user's privacy. The results also raise the alarm to developers about the importance of applying security protocols instead of designing proprietary ones.

In this paper, we make the following contributions:

– We conduct an in-depth analysis on insecure network communications of Android apps, which provide a panoramic view of the usage of insecure communication in Android apps.

- We present RawDroid as a protocol audit system to analyze the security of data transmission through proprietary protocols in Android and use it to perform a large scale evaluation of proprietary protocols.
- Our study demonstrates that transferring data through proprietary protocols is common in modern Android apps. However, most of them fail to protect the data due to security flaws.

The remainder of the paper is organized as follows. We present the preliminaries in Section 2. In Section 3, we describe our study on classifying network protocols in Android apps, and how to apply RawDroid to audit the security of those protocols. We evaluate RawDroid in Section 4 and discuss details of crypto misuses in proprietary protocols. Related work is introduced in Section 5 and we conclude the paper in Section 6.

2 Background and Motivation

2.1 Android Security Mechanism

The security of the Android platform is based on the robust security of Linux kernel, and its sandbox and permission protection mechanism, which isolates and run Android apps on their own process and protect users sensitive data against malwares. However, a larger number of Android apps are designed to be open to transmit users data to servers through the network. This means that the protection of the sandbox and permission cannot cover the network aspect, and potential vulnerabilities to sensitive data still remain.

2.2 Network Communication in Android

Network protocols are a common method for exchanging data between client and server. Android apps rely on the security of network protocols to guarantee the confidentiality of private data. However, in order to finalize functions, inexperienced developers adopt insecure communication in their productions. In order to in-depth analysis, we classify network protocols into two categories according to their security characteristic.

Secure Network Protocol HTTPS [4] is the use of SSL or TLS as a sublayer under regular HTTP application layering. It encrypts and decrypts user data requests as well as the data that are returned by the Web server. It is more secure than other common application layer protocols. Along with the increase of security requirements, the applications on iOS platform are all forced to use HTTPS, since HTTPS is more secure to transport data after encrypting. Also, HTTPS Everywhere [3], a famous tool for extension of browsers, suggests that all communications are encrypted for HTTPS with various web servers, making network communications more secure. Therefore, by using HTTPS in Android apps, it will prevent sensitive data from leaking in untrusted network.

Insecure Communication Some companies are unwilling to share their information with others, while they build an individual proprietary protocols to communicate among their own products. It is more convenient and easy-deployment on Android platform. Therefore, there are numerous benign Android apps using proprietary protocols as a principal way of network communications. However, this method is also a preference for malware developers since most AVs or malware-detection tools monitor port 80 or 443 which are used by conventional protocols (e.g., HTTP, HTTPS).

Android framework provides APIs for allowing developers to apply proprietary protocol. In TCP, server, in the Java code, declares a *ServerSocket* object and binds its IP address and port number. Then the server calls *accept* method to wait clients connections. As a client, app who wishes to communicate with the server needs to create a Socket object to record the IP address and port of the server, and then sends streams by writing data into an *Outputstream* object. While, in UDP, an app, as a client, uses class *InetAddress* to build a socket with the server. While, in the native code, clients just need to put data into a char array and send it by function *send* or *sendto*. In order to guarantee the security of sensitive data, developers need to encrypt data by cryptographic algorithms to prevent attackers from stealing critical data or preforming a MITM attack.

2.3 Threat Module

The security of both HTTP and proprietary protocol requires the guarantee that developers encrypt sensitive data by using cryptographic algorithms. Even with experienced developers, there are also existing misuse of cryptography in Apps. Therefore, they are weak for remote network-based attackers.

Our threat model assumes attackers can capture the network traffic and reverse analyze Apps. Users sensitive data have faced three attack vectors. In general, we classify a network connection as vulnerable if: (1) HTTP is a protocol which takes plain text to communicate with servers. It will be easy to perform MIMT attacks. (2) Apps submit data in plain-text by proprietary protocols. (3) Developers encrypt sensitive data, but exit the misuse of cryptography (e.g., hard-coded key, plain-text key exchange).

3 Analysis of Insecure Communication

How network communication guarantee the security of apps in Android is a common question to security researchers. In order to investigate this question, we study 435 apps from Google Play and MyApp Android market.

Given an app, our preliminary step takes an off-the-shelf installed app as input, and launches it relying on Google Monkey [5], a popular tool for automatically executing and simulating user interaction to trigger various events of apps. After that, we collect informations about the network usage of the app. The command *netstat* is used to monitor all kinds of network information, such as network usage, interface statistics, route table and so on. We use this command to record each server's IP address and port numbers during Android runtime. However, it is limited to only preform *netstat*. In order to record the app's individual network usage completely, we also monitor these four system files: */proc/net/tcp(6)* and */proc/net/udp(6)*. All of the four files are readable with any permissions. These four files is used to record all connections status for TCP or UDP. We use app's uid to filter each network trace. For the future analysis, we also capture the network traffic for each app.

A snippet of command *netstat* results and a example of content read by these four system files are as follows:

```
Proto Local Address     Foreign Address    PID/Program name
tcp   192.168.3.2:53253 121.195.187.60:80  8799/sogou.mobile.e
tcp   192.168.3.2:34122 111.202.114.74:443 2408/com.yuntongxun
tcp   192.168.3.2:56185 203.208.43.98:80   1040/android.proces
tcp   192.168.3.2:52999 121.195.187.54:80  8799/sogou.mobile.e
tcp   192.168.3.2:45646 121.195.187.54:80  7082/com.sohu.input
tcp   192.168.3.2:40621 121.195.187.54:80  7082/com.sohu.input

local_address rem_address   st    uid  timeout inode
0100007F:13AD 00000000:0000 0A      0        0 20957
0203A8C0:D96B EE95387B:1F95 01  10666        0 73783
```

According to the first snippet, we recode *proto*, *foreign address* and *program name* to further analysis. Meanwhile, from the field *rem_address* and *uid*, we can combine with the results of *netstat* to select candidate data packets.

Table 1. The Types of Protocols Used By Sample Apps

Type	The Number of Apps
HTTP	203
HTTPS	69
Proprietary Protocol	158
Total	435

In this work, we built a dataset of apps. We select 435 apps which were randomly downloaded from Google Play and MyApp Android Market, and each of them had been downloaded at least millions of times. There were three apps that had no ability to communicate with network, and two apps crashed by some reasons. We finally successfully analyzed 430 apps. The results of our analysis are listed in Table 1.

The results demonstrate that commonly used protocols in network communication fall into three categories — HTTP, HTTPS, Proprietary Protocol. The former two have relatively sophisticated mechanisms for maintaining security, and developers usually use them to transport information about UI or web pages. The latter is a unique feature for many online-privacy-related apps to send sensitive data to servers.

It is easy to know that apps will use both HTTP and HTTPS. We consider that this situation is insecure. In this paper, we define that only HTTPS is secure protocol and HTTP and proprietary protocol are insecure. The definition of secure and insecure protocols is not arbitrary, and it is necessary and realistic for our study. First of all, all the sending data are plaintext which are encapsulated into HTTP. It is easy to be MITM attacked. For example, in an untrust WiFi channel, a game app only uses HTTP to transmit data, attackers can tamper with the runtime data. If the packages contain any payment information, it may fool users and make game companies profit loss. For another instance, an app GET a picture from advertisement server by HTTP, attackers can tamper with network traffic and load false advertising to cheat users on purpose. And if developers encrypt sensitive data, there are also numerous misuses of cryptography in the processing of encryption. Proprietary protocols are in the same situation.

According to the result, only 69 (15.9%) apps use HTTPS as their transmission method. 158 (36.7%) of the apps use other protocols (e.g., proprietary protocol, TCP, UDP) to transport sensitive data to servers. 203 apps utilize HTTP as their primary way of network communication. The fact clearly shows that users' sensitive data are insecure and easily expose to attackers.

Table 2. Network Port used by Different Protocols

Protocol Type	Port Number
HTTP	80; 8000; 8007; 8015; 8080; 8088; 8888
HTTPS	43; 4433; 8020
Proprietary Protocol	5000-6000; 8000-9000; 12000-12100

We also summarize the port numbers used in different protocols, which is listed in Table 2. Most of them use HTTP protocol as their main methods to communicate with server. These connections transport resource and sensitive data, such as users' authentication informations, geographic coordinates and even some script codes. Table 2 shows these types of connections that exploit the port numbers ranged from 8000 to 8080, and the 80 port. Lots of server developers want to use some similar port to prevent the port occupation or distribute servers' load. While, major apps uses the 43, 443, 8020 to implement HTTPS, and transfer the data which cities users are near by. Table 2 also shows that there are no particular rules on port numbers, since, it depends on developers' hobbies. Therefore our strategy is that we classify each type of connections by their port numbers.

3.1 Analysis of Proprietary Protocol

In order to analyze the vulnerability of the usage of insecure communications and assess their prevalence in existing apps, we implement RawDroid to scalably and accurately examine apps from Google Play and MyApp Android markets. It consists of Inline Hook to record necessary runtime information, and analysis of potential weakness. In the following paragraphs, we present key technical details applied in our analysis framework.

We randomly select 60 apps from the proprietary protocol category to analyze whether it exists any potential threats, and the result is listed in Table 3. We find that most apps choose to send sensitive data in the form of plain-text directly by TCP or UDP without any protections, which are insecure communications. Few of them use custom algorithms to protect data, but still expose many threats to users. In order to investigate this situation in real world, we make in-depth study as follows.

In Fig. 1, we show the overall workflow of the system, which involves several steps to determine potential threats in Android apps. Given a target app to analyze, the first step is to dynamic determine whether apps use insecure communications, and if so, we capture data packets and network usage status in the meantime. In the second step, we select data packets of all candidate apps, and use traffic analysis module to detect the apps whether contains sensitive data in plaint-text. After this step, we classify apps which use encrypted proprietary protocols and re-run them to record method traces of network API and specified parameters. Finally, in order to determine whether these connections are indeed vulnerable, we combine program analysis technique to detect and manually verify the security of proprietary protocols.

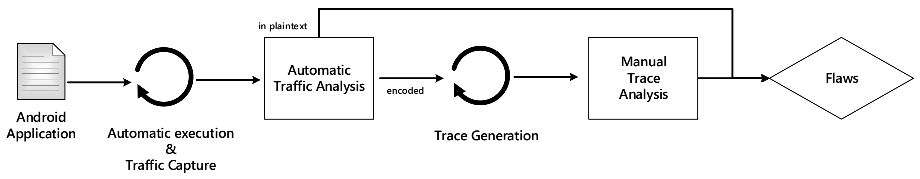

Fig. 1. Analyse Apps with RawDroid

It is very natural to consider why we use dynamic analysis rather than static analysis for detecting the security of apps' network usage. To our knowledge, many apps will get IP addresses and port numbers not only by hard-code, but also with dynamically post the requests of clients. Static analytical methods have no idea to completely identify which servers are available to the clients. Moreover, static analysis often spends a lot of time and memory that beyond our control. In order to conveniently obtain the network statuses of apps as complete as possible, we chose to use dynamic analysis for our initial analysis.

Traffic Analysis This analysis mechanism mainly identifies whether Android apps use plant text by un-common port numbers. We implement this mechanism in Python. Before analysis, we first use *adb* pull out the data packets named by app's package name from SD card of device. While the aforementioned techniques greatly select out candidate connections, this is not enough to precisely analyze and contains too mush useless informations. To our knowledge, we discover that many developers use port 8080 instead of the port 80 to implement HTTP protocol. This phenomenon is also found in HTTPS protocol (e.g., 4433). Based on the above reasons, we also take these port numbers as out filter conditions.

Secondly, we estimate the candidate suspectable connections according to the semantic contents of data streams. According to the port numbers, we first extract out each TCP stream which is resulted from the first module. Then we preliminarily filter out all the application layer protocols, and remain TCP or UDP. In the next, we analyze the semantic content and determine whether the apps have sent plaintext data by un-common port numbers. As our manual analysis found, apps send data which start with some continuous visible characters, such as the package name, or packages data into a json form. Depending on these key characteristics, we determine un-protection channels which transmit sensitive data in plain-text. We predefine some key words, such as username, password, and use them to detect whether there are sensitive data in the network traffic. Dealing with the apps which use encrypted proprietary protocols, we also record their runtime trace, and manually analyze the misuse of cryptography in apps.

Trace Generation To our knowledge, simply recording IP addresses and port numbers of remote servers provides limited help. In this phase, we need to log method traces to reduce the complexity of the analysis. We re-run the candidate apps to use Inline Hook technique to hook key methods and record method traces and every parameters we are interested in. With these runtime values, we can simply locate where these key functions are called. Meanwhile, we capture data packets of all candidate apps during runtime.

In order to implement the method of logging method traces, we use inline hook technology to hook key network APIs. Android app can access to the Internet via several different APIs include Jave and native code. In Java, class *Socket* and *InetSocketAddress* both provide interfaces that allow endpoints of a network connection interacting with servers. A new network connection can be created by calling the constructor *Socket(String dstName, int dstPort)* in TCP or *InetSocketAddress(String hostName, int port, boolean needResolved)* in UDP. The parameter *dstName* and *hostName* expects a string giving the IP address or domain name of the destination and the parameter *dstPort* and *port* represent a port number on which the destination is to be contacted. We use *Xposed* [9] framework to hook these APIs in Java, log runtime values and record call strack. In the native code, the API *send* and *sendto* models a native network connection. In this case, we use *adbi* [8] framework to query informations of remote server by *getpeername* and log the *buff* domain.

In this module, the main challenge is that it is difficult to obtain specific runtime values and traces the data flow in Java. Just hooking this API may causes a lot of noisy information. Therefore, in order to reduce the difficulty of analysis, we use the data packets which is produced by the first step instead of hooking.

Meanwhile, in the native code, recording where methods are called is also a challenge. We try to use *strace* [7], a tool used in tracing system calls and signals, to address this problem. However, *strace* produces a large number of noisy information to interference our analysis. Also, we use the class *CallStack* or the function *Backtrace*, and, not surprisingly, they do not meet our requirements. In fact, in order to address this challenge, we hook the API *send* and *sendto* and record their return address in memory. According to this address, We read the file *proc/⟨pid⟩/maps* and get the call points of *send* and *sendto.*

A snippet of trace and specified value of key network APIs in Java is as follows:

```
InetSocketAddress IP: 192.168.1.20 Port: 8888
com.example.testxposed.XModule.printStackTrace(XModule.java:157)
...
java.net.ProxySelectorImpl.lookupProxy(ProxySelectorImpl.java:101)
java.net.ProxySelectorImpl.selectOneProxy(ProxySelectorImpl.java:55)
java.net.ProxySelectorImpl.select(ProxySelectorImpl.java:32)
...
com.noah.king.framework.f.d.a(Unknown Source)
com.noah.ifa.app.standard.service.RotateService.b(Unknown Source)
com.noah.ifa.app.standard.service.RotateService.c(Unknown Source)
com.noah.ifa.app.standard.service.b.run(Unknown Source)
java.util.Timer$TimerImpl.run(Timer.java:284)
```

The first line is the hooked network API, and runtime values. The following information is the method trace. By locating the particular Java class name, such as *com.noah.ifa.app.standard.service.RotateService*, we can find what function the class is.

Proprietary Protocol Analysis For those encrypted data streams, it is a huge challenge to detect which algorithm is used in the app. Given the exisiting works, CDRep [14] has implemented similar function that automatically detects misuses of cryptographic APIs. The CDRep modules rules of misuses and matches them with byte code. However it is not available in our case, since most proprietary protocols have no common and similar rules to match. Some of them use standard cryptographic algorithm (e.g., AES, DES), and some just use their own individual method of encryption. Therefore we manually analyze the specific algorithm in apps both in Java level and native code. In order to do so, we take the log files as input, to search for interesting connections, and find their method call traces. With these auxiliary informations, we are able to quickly locate the functions where data streams are encrypted, and find potential vulnerabilities.

Limitations Although RawDroid helps to analyze the security of data transmission, it still has to depend on human efforts to validate potential vulnerabilities. However, one goal of RawDroid is to contribute to researchers quickly locate the cryptographic functions. In particular, our work can simply unite with static or dynamic analysis tools (such as TaintDroid, FlowDroid) to raise precision and understand the flow of sensitive data sent through proprietary protocol.

RawDroid may have false negatives, since we cannot trigger all connections in runtime. Moreover, apps will dynamical request the information of servers, and our filters do not distinguish this kind of network traffic. They could introduce little noisy information.

4 Evaluation

In this section, we evaluate our systems ability and efficiently identify suspicious connections on a large set of apps from Google Play and MyApp Android market, including entertaining, online shopping, and social network communicating. We use LG Nexus 4 running Android 4.4.3 to evaluate RawDroid. We also root this phone, because RawDroid deploy Xposed and Adbi to record runtime information, which requires root assessment. Then, we review the results of our analysis and why it happens. At last, we propose some advices to enhance situation.

In order to verify the results and draw a conclusion, for our study, we randomly select 60 apps from the proprietary protocol category. Table 3 presents the results of our analysis. We take the network traffic of each app as input, and find that most (54) of them only use TCP/UDP to transmit plain text with on deeper protection. More in-depth discussions are outlined in the following. We are also noted that there are only six apps that use cryptographic algorithms to protect data security, but still existing the misuse of cryptography. However, none of them use sound encryption mechanisms. The fact also reveals that a large number of developers have no security experience and users sensitive data is insecure to expose to the attackers. In order to consider what we found, we categorize the results into 2 types such as Plain Text and Proprietary Protocol.

Table 3. Result of Proprietary Protocol Analysis

Potential Weakness	The Number of Apps
Plain-text	54
Hard-coded Key	3
Plain-text Key Exchange	3

4.1 Plain-Text

As Table 3 shown, there are many (54) Android apps that just use TCP protocol to carry information without any encryption. As we all know, TCP provides reliable, ordered, and error-checked delivery of a stream between apps running on hosts communicating over an IP network, however it does not support safe channel to transport data. In consequence, it is an untrusted channel and easily exposed to attackers. According to the content of data streams, we summarize 3 categories as following.

Sensitive Data. As our analytical results presented, this plaint text channel send users' name, password, cell-phone number, geographic coordinates, device information, session token, even the history of shopping and price of commodities. In some public network environment, attackers just sniffer router and capture packets with network analysis tools, and they can also get a view of users.

Controlling Data. Game apps often employs some bites to control game process, such as the level of VIP, virtual product trades, health point of enemy, and heartbeat packets. With analyzing the meaning of each bites and modifying necessary values, game companies suffer great losses.

Pushing Message. Pushing message is a common phenomenon in the existing apps, which is offered to end users with notification and information from remote server to apps. Some of messages are significant since developer-run severs may send update message of versions and even some sensitive data. Most pushing message, as third-party libraries, are employed into apps, such as Igexin, JPush and MiPush in China. According to our discovery, only a small part of these pushing libraries use security channel to protect their messages. Igexin [1], one of the most popular pushing company, utilizes customize algorithm to encrypt messages and the key is sent with cipher message in plaint text.

4.2 The Misuse of Cryptography in Proprietary Protocol

After we manually analyze 60 apps, there are 6 applications that use proprietary protocols to transport data. 5 of them contain same kind of third-party libraries, which is provided by *Instant Messaging Cloud* companies. This kind of *IM* implements an unique protocol that transports informations of authentication and chatting message. In order to improve users' security awareness and understanding, we study how they prevent their content against hackers in detail, which is described in Section 4.3. The rest of apps, a financial app, use Caesar cipher to encrypt name and price of stock. According to the encryption methods, we also summarize 2 types of misuse of cryptography as follows.

Hard-coded Key. As our manual analysis resulted, apps use standard cryptographic algorithms with hard-coded key. In order to reduce costs of programming, developers are unwilling to design a key exchange mechanism to guarantee the security of communications. With general considerations, hard-coded key is easily exposed to attackers after reversing the binary.

Transfer key in plaintext. *com.jedigames.guaji.xiaomi* is a game app which can communicate with other gamers during playing. It sends pain-text key in the beginning of game and then the encrypted chatting messages follows. Attackers can just capture the data packets and decrypt the specific content of messages.

In summary, the situation of proprietary protocols is common in Android apps. Some apps attempt to send sensitive data intentionally, to avoid traffic monitor. What is more, some developers, with no security experience, build softwares for scalability and business convenience. However, unsecurity channel exposes users' data to pubilc. While some encrypted protocol, without safe mechanism of key distribution, uses hard-coded or plaint text keys which can be easily found. To avoid this unsafe vulnerability, we provide recommendation to developers that must make use of a sound standard protocol to transport their data such as HTTPS. Even with HTTP, each important data must be encrypted by sound and safe algorithm. More importantly, a security key exchange mechanism, such as *Diffie-Hellman*, is necessary to be applied into protection.

4.3 The Misuse of Cryptography in Proprietary Protocol

In this section, we conduct the misuse of cryptography on two selected third-party libraries from Instant Messaging Cloud provider. With the popularity of Android platform for mobile Internet industry, apps which add communication functions, use *IM* to improve user experience. We find that there are at least 30 apps, each of which has been downloaded for 2 million times from game and sociality categories, that use these two libraries. And more than 30 thousand subscribers chose this libraries in Android or iOS platform. According to analysis results, most of them use proprietary protocols with custom algorithm, thence, we investigate the point that whether there are security risks. These case study both show that, although most of these third-party libraries use a proprietary encryption protocol to interact with the servers, and declare their own protocols robust and security, cryptography misuse is not uncommon. Finally, we also describe a attack model for each case.

GotyeIM *com.open_demo* is a sample app for GotyeIM, one of the most popular IM provider in China, which can download from its official website. Its original intention is that provides developers a demo to quickly adapt, and it contains all needed IM libraries. GotyeIM first sends its authentication information to server. When it succeeded, the server sends a 32-byte pain-text key A to its Android client. What is more, in native, app *xor* a hard-coded key B with the key A to product a real key. After pre-process, app sends the key A to chatting server and encrypts user's chatting messages, login informations, session tokens and history records with *3DES-ECB*. It is worth mentioning that this app sends request data with the key A and a constant string "AES" as a fake algorithm tip.

Attack Model In this threat model, we assume that victims and attackers are both in a same public environment, such as cafe, book store. We also assume that victims trust the proprietary protocol in GotyeIM, and are willing to use it to chat sensitive data with others. The objective of attackers is to hijack app's sessions and even preform a MITM attack. To prepare for such attacks, attackers need to capture victims' data packets. After that, attackers use the key A which can be found in the beginning of the connections, *xor* the hard-coded key B to product the real key. At last, attackers use the real key to decrypt messages with *3DES-ECB*.

RongLian *com.yuntongxun.ecdemo* is a sample Android app for another popular IM provider called RongLian which has cooperation with hundreds of companies. We download the demo from its website and analyze it manually. The result shows that this third-party library encrypts user's chatting history and messages to chat server with hard-coded key. The app defines a function called AES, however it implements as a simple encryption method that encrypts data based on a permutation table. The most noteworthy is that its decryption function can be searched by Google [2]. We try this Google's version to encrypt and decrypt app's data, not surprisingly, we easily obtain user's private data. Finally, the library encapsulates all data into Protocol Buffers, a method of serializing structured data without data encryption by Google, and sends it in TCP.

Attack Model Our attack model assumes that attackers control the router to capture users' network flow by some methods, for example, ARP attack, a fake AP, etc. The goal of attackers is to extract chatting messages from data streams and send it to a server. We also assume that the apps which contain RongLian libraries is installed by the user. To achieve this, attackers need to extract chatting message field from Google Protocol Buffers [10], and then use the decryption codes which can be searched from Google [2] to decrypt the ciphertext with hard-coded key.

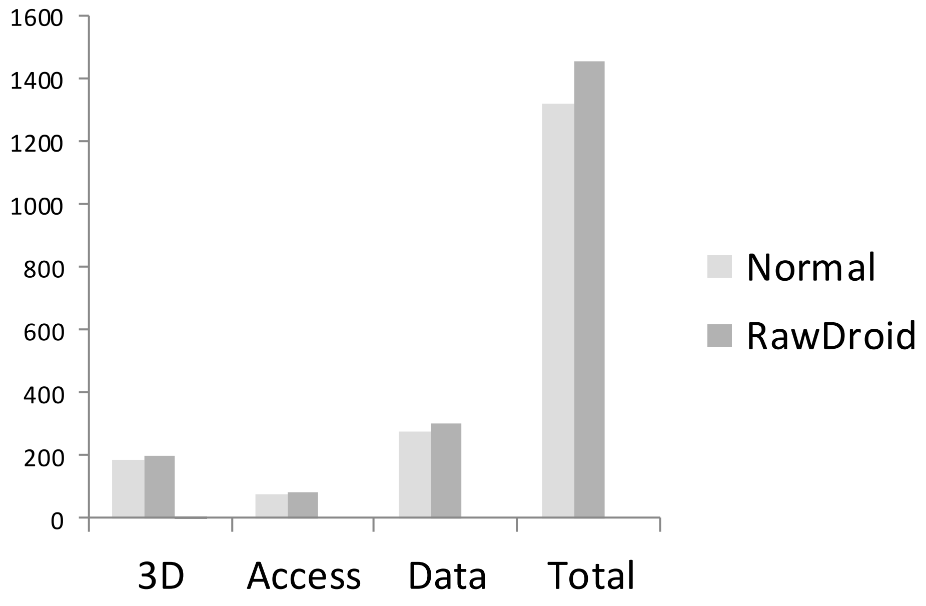

Fig. 2. 0xbechmark Result

4.4 Performance

Notice that native code is a common way to be used for performing heavy computation and return results back to Java, while Java code is used more in building UI or implementing logics of general functions. Therefore, it is necessary to test

RawDroids performance to evaluate the running states of RawDroid in real-world Android devices.

In order to evaluate RawDroids impact on performance in Android, we use 0xbechmark [6], a comprehensive benchmark tool, to perform measurements on the Nexus 4 device. We depict our results in Fig. 2. The y axle is the run time for each category and the unit is millisecond. We discuss the related data of 3D, access and I/O time. Notice that in all categories, except Data, the gap of run time overhead is not noticeable. RawDroid preforms less than 5% on average. However, overhead becomes significant in Data, since most apps frequently use raw sockets to network communication, we need to hook each connection and log some specific parameters. RawDroid preforms additional 10% overhead on this work. Overall, our benchmark experiments demonstrate that RawDroid makes a minimum impact on system performance.

5 Related Work

To the best of our knowledge, there is no in-depth discussion of the security about proprietary protocol and raw socket usage. Most relevant works are static and dynamic analysis for detecting leakage of sensitive data or focusing on security of HTTP(S) in Android platform. In this section, we describe these related works.

Privacy Leakage A series of works in the security of privacy disclosures on Android platform have been proposed. Both static, including Flowdroid [15], TainDroid [16], and dynamic techniques, including ContentScope [17] and CHEX [18], are used to detect the privacy disclosure. In particular, Flowdroid [15] checks whether Android apps leak private data from source to sink by static taint analysis. TainDroid [16] dynamically taints sensitive information to trace where privacy can be leaked. It alerts users when privacy is sent to the network. ContentScope [17] examines a large number of apps and analyzes whether apps expose content provider to other apps. CHEX [18] presents a method to statically discover sensitive flow. Also, in Securacy [24], Ferreira D et al. develop an app to help users fully know what their apps do. However, it considers all un-regular port as unsafe, which is a arbitrary decision.

The main idea in detecting privacy leakage is that uses taint analysis technique to discover sensitive information flow from custom source to sink. However, Flowdroid [15] costs too mush memory to analyze a app. Meanwhile, it is not feasibility that TainDroid [16] modifies a system to support its function. Therefore, some researchers propose similar tools with natural language processing technique such as Describeme [19], Supor [20] and Uipicker [21].

Security of HTTP(S) For the past few years, there has been less works related to the security of HTTP(s) in Android platform. For the security in HTTP, most works discuss how Android apps protect user authentication. In Appcracker [13], Cai F et al. presents Appcracker and checks the vulnerabilities of user authentication in a public network. In [22], Liu Hui et al. proposes a methodology to evaluate potentially vulnerable of user authentication in Android apps. Meanwhile, in HTTPS, Fahl S et al. Mallodroid [12] presents a tool to detect the state of SSL in Android apps and draw a conclusion about misuse of SSL. Also, Sounthiraraj et al. Hunter [23] combines both static and dynamic analysis, and focuses on analyzing the prevalence of vulnerability in Android apps against MITM attacks.

6 Conclusion

We conduct the first step to investigate the usage and security of network protocol in the most popular Android apps. Our result shows that 36.7% apps use insecure communications. We design and implement a tool, RawDroid, to determine whether apps contain these vulnerabilities. Most of these vulnerabilities are caused by sending plain-text through HTTP or TCP. Also, some apps contain third-party libraries which misuse cryptographic algorithms to build proprietary protocols. We envision that our study can raise the attention of these vulnerabilities for security researchers and app developers.

References

1. Igexin `http://www.wooyun.org/bugs/wooyun-2010-0185354`
2. CodeSearch `https://searchcode.com/codesearch/view/13566752/`
3. HTTPS Everywhere `https://www.eff.org/https-everywhere/`
4. HTTPS `https://en.wikipedia.org/wiki/HTTPS`
5. Google Monkey `http://developer.android.com/guide/developing/tools/monkey.html.`
6. 0xbenchmark `0xbenchmark.appspot.com`
7. Strace `https://en.wikipedia.org/wiki/Strace`
8. The Android Dynamic Binary Instrumentation Toolkit `https://github.com/crmulliner/adbi`
9. Xposed `http://repo.xposed.info/`
10. Google Protocol Buffers `https://developers.google.com/protocol-buffers/`
11. M. Marlinspike. More Tricks For Defeating SSL In Practice. In Black Hat USA, 2009
12. Fahl S, Harbach M, Muders T, et al. Why Eve and Mallory love Android: An analysis of Android SSL (in) security[C]//Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012: 50-61
13. Cai F, Chen H, Wu Y, et al. Appcracker: Widespread vulnerabilities in user and session authentication in mobile apps[J]. MoST 2015, 2014.
14. Ma, Siqi, et al. "CDRep: Automatic Repair of Cryptographic Misuses in Android Applications." Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM, 2016.
15. Arzt, Steven, et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." ACM SIGPLAN Notices 49.6 (2014): 259-269.
16. Enck, William, et al. "TaintDroid: an information-flow tracking system for real-time privacy monitoring on smartphones." ACM Transactions on Computer Systems (TOCS) 32.2 (2014): 5.

17. Jiang, Yajin Zhou Xuxian. "Detecting passive content leaks and pollution in android applications." Proceedings of the 20th Network and Distributed System Security Symposium (NDSS). 2013.
18. Lu, Long, et al. "Chex: statically vetting android apps for component hijacking vulnerabilities." Proceedings of the 2012 ACM conference on Computer and communications security. ACM, 2012.
19. Zhang, Mu, et al. "Towards automatic generation of security-centric descriptions for Android apps." Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security. ACM, 2015.
20. Huang, Jianjun, et al. "Supor: Precise and scalable sensitive user input detection for android apps." 24th USENIX Security Symposium (USENIX Security 15). 2015.
21. Nan, Yuhong, et al. "Uipicker: User-input privacy identification in mobile applications." 24th USENIX Security Symposium (USENIX Security 15). 2015.
22. Liu Hui, Zhang Yuanyuan, Li Juanru, Wang Hui, Gu Dawu. Open Sesame! Web Authentication Cracking via Mobile app Analysis. in 18th Asia Pacific Web Conference (APWEB 2016). Suzhou, China. Sept 23-25, 2016
23. Sounthiraraj, David, et al. "Smv-hunter: Large scale, automated detection of ssl/tls man-in-the-middle vulnerabilities in android apps." In Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS14. 2014.
24. Ferreira D, Kostakos V, Beresford A R, et al. Securacy: an empirical investigation of Android applications' network usage, privacy and security[C]//Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. ACM, 2015: 11.