*A Systems Approach to Cyber Security*
*A. Roychoudhury and Y. Liu (Eds.)*


*doi:10.3233/978-1-61499-744-3-3*

# Chat-App Decryption Key Extraction Through Information Flow Analysis

Zhongmin DAI, SUFATRIO, Tong-Wei CHUA, Dinesh Kumar BALAKRISHNAN, Vrizlynn L. L. THING

*Institute for Infocomm Research, Singapore*
*Email: {daiz, sufatrio, twchua, dineshb, vriz}@i2r.a-star.edu.sg*

**Abstract.** Recent years have seen a pervasive usage of mobile-based instant messaging apps, which are popularly known as chat apps. On users' mobile devices, chat logs are usually stored encrypted. This paper is concerned with discovering the decryption key of chat-log database files as they are used by popular chat apps like WhatsApp and WeChat. We propose a systematic and generalized information-flow based approach to recovering the decryption key by taking advantage of both static and dynamic analyses. We show that, despite the employed code obfuscation techniques, we can perform the key discovery process on relevant code portions. Furthermore, to the best of our knowledge, we are the first to detail the employed string de-obfuscation, encrypted database file structure, and decryption-key formulation of the latest WhatsApp with crypt12 database. We also demonstrate how our key-extraction techniques can decrypt encrypted WhatsApp and WeChat database files that originate from a target device. Additionally, we show how we can construct a version of WhatsApp or WeChat that simulates the key generation processes of a remote target device, and recover the keys. Lastly, we analyze why our technique can work on widely-popular chat apps, and mention measures that can be adopted by chat-app developers to better protect the privacy of billions of their users.

**Keywords.** Mobile security, privacy protection, Android, mobile apps, chat apps

## 1. Introduction

The unprecedented proliferation of mobile devices in recent years has led to a pervasive usage of mobile-based instant messaging applications, which are also popularly known as *mobile chat apps*. Such apps allow mobile users to instantly exchange text messages and media files to each other on either an 1-to-1 or user-group basis. On user devices, chat logs are stored in database files, and are usually encrypted. Decryption of the chat-log files thus represents a very serious threat to the privacy of mobile users.

This paper is concerned with discovering the decryption key (password) of chat-log database files as they are used by two widely-popular chat apps, namely WhatsApp [1] and WeChat [2]. It proposes a systematic and generalized approach to discovering the decryption key of both apps by inspecting the flow of sensitive key-related information, particularly towards the employed cryptographic libraries. Using the two popular apps as real-world examples, we show that, despite the employed code obfuscation techniques, we can still discover the actual decryption keys in use and gain understanding of the key

construction process. Furthermore, to the best of our knowledge, we are the first to detail the employed string de-obfuscation, encrypted database file structure, and decryption-key formulation of WhatsApp app that uses the latest (as of October 2016) `crypt12` database file.

To realize our information-flow analysis, we make use of a combination of both dynamic and static analyses. In our work, which is implemented on Android, we employ both the Dynamic Dalvik Instrumentation (DDI) [3,4] and Android Dynamic Binary Instrumentation (ADBI) [5,6] dynamic analysis toolkits in order to observe database-file access operations and to recover the decryption keys in use. We also utilize static analysis tools, including `Apktool` [7], jadx Dalvik-to-Java decompiler [8], and taint trackers [9,10], to assist us in understanding the code structure, component interactions, and decryption-key information flow within the target chat apps.

While there exist prior articles that explain how decryption keys of some specific older versions of WhatsApp and WeChat are derived [11,12], our work takes a generalized approach to observing key-related information flow of the chat apps. Hence, while existing scripts target only very specific older or current versions of chat apps, our proposed technique can apply to different chat-app versions as long as they retain their same design and practice of employing cryptographic libraries and passing the key information into the libraries.

Our experiments done on a rooted device show how our analysis of WhatsApp and WeChat allows us to discover the decryption keys and all cipher-operation parameters. Subsequently, we can extend the key-extraction techniques by showing how we can construct a WhatsApp and WeChat app version that can simulate other devices' key generation process. As a result, we will be able to decrypt encrypted WhatsApp and WeChat log database files that originate from a remote target device, thus discovering the target mobile user's chat activity information.

Given the huge user base of both analyzed chat apps, our decryption-key discovery results are therefore very important. Hence, we also provide an in-depth analysis of why our proposed technique can still discover the keys of highly-popular chat apps. Based on our root-cause analysis results, we propose concrete counter measures that can be adopted by chat-app developers to improve their future app versions and better protect the privacy of billions of their users.

In summary, our work in this paper makes the following contributions:

- We propose a systematic and generalized approach to discovering the decryption key of chat apps by inspecting the flow of key-related information particularly towards their employed third-party cryptographic libraries.
- Using a set of experiments on a rooted device, we demonstrate how we obtain the decryption key and cipher-operation parameters of both WhatsApp and WeChat.
- To the best of our knowledge, we are the first to detail the employed string de-obfuscation, encrypted database file structure, and decryption-key formulation of the latest crypt12 WhatsApp database file.
- We elaborate how we can derive a chat-app version that can simulate other devices' key generation process in order to decrypt encrypted log files that originate from a remote target device.
- Lastly, we provide an in-depth analysis on why our technique can work on highly-popular apps, and suggests counter measures that can be adopted by chat-app developers to prevent potential attacks.

The remainder of this paper is organized as follows. Section 2 gives some background and mentions related work. Section 3 elaborates our proposed chat-app analysis approach. Section 4 reports our experiments on WhatsApp and WeChat. Section 5 explains how we can derive a chat-app version that simulates other devices' key generation process. Section 6 suggests how chat-app developers can improve their future apps, and also discusses ethical considerations of our work. Finally, Section 7 concludes this paper.

## 2. Background and Related Work

### *2.1. Background*

We give some background on WeChat and WhatsApp apps that we analyzed. We also briefly describe the dynamic and static analysis tools employed in analyzing the apps.

#### *2.1.1. Popular Chat Apps and Their Database Files*

WeChat [2] from Tencent is among the most popular chat apps. It is originally known as Weixin, and has a huge user base particularly in Asia [13]. The latest company report mentions that the combined monthly active users (MAU) of WeChat and Weixin in 2015 reached 697 million, which is an increase of 39% from the previous year.

In Android, WeChat private data is stored under the `/data/data/com.tencent.mm` folder. Two sub-folders of interest are the `MicroMsg` and the `shared_prefs`. Other than several configuration files, the `MicroMsg` folder contains a folder that has a name resembling a random alphanumeric string, e.g. `b98a831a089cef3031385d56a0f51927`. This folder contains two encrypted database files, namely `enFavourite.db` and `EnMircroMsg.db`. They are both encrypted using SQLCipher [14], which is an open-source encryption extension to SQLite. SQLCipher provides a transparent database encryption interface to SQLite by wrapping SQLite method invocations so that encryption and decryption can be carried out before the data is written into the database or after it is read from the database, respectively. The `shared_prefs` folder, meanwhile, contains `.xml` files that store key-value pairs of WeChat settings.

WhatsApp [1], which is now owned by Facebook Inc., is currently the most popular chat app [15]. In February 2016, WhatsApp reported that it had 1 billion users [16]. For Android devices, WhatsApp stores its log within two unencrypted SQLite database files `msgstore.db` and `wa.db` under the protected `/data/data/com.whatsapp/databases` folder. Additionally, WhatsApp keeps an encrypted backup of the `msgstore.db`, which is wrapped by a header and trailer containing several pieces of account information, on the phone's SD card. This file is stored with the `.crypt`$\langle$*version_no*$\rangle$ filename extension, with `crypt12` being the most recent one as of October 2016. The file is utilized when the mobile user transfers his/her chat log over to a new phone while retaining the existing WhatsApp phone number [17].

Similar to WeChat, there exist articles in the literature that explain how to obtain the key of the encrypted database file of specific WhatsApp version. For instance, [11] derives the key for the now-defunct WhatsApp `crypt7` database file. Our work reported in this paper, instead, takes a generalized approach to discovering the decryption key based on an information-flow analysis of the targeted chat apps.

### 2.1.2. Dynamic Analysis using DDI/ADBI Toolkits

We perform our dynamic analysis of chat apps on Android by using the Dynamic Dalvik Instrumentation (DDI) and Android Dynamic Binary Instrumentation (ADBI) toolkits. The DDI toolkit [3,4] is employed to monitor the Dalvik code component of a target Android app. It performs an in-line hooking technique of Dalvik-method entry points, and diverts the invocation of a target Dalvik method into a correspondingly added JNI-based native method. The added native method will ultimately invoke the original Dalvik method, thus enabling itself as a code-instrumentation point for the original Dalvik method. The DDI toolkit also supports the loading of additional Dalvik classes into a process, thus allowing the instrumentation code to be partially written in Java.

The DDI toolkit operates on top of the ADBI toolkit [6,5], which implements the hijacking utility of the ARM binary code. Due to its binary-level hooking feature, the ADBI can hook the native code of an Android app. In our chat-app dynamic analysis, we simply print out the parameters passed into the monitored methods, which can then be viewed using the `adb logcat` command.

Using the DDI/ADBI toolkits, performing a dynamic monitoring and instrumentation of Android apps does not require any modification of the target apps, thus keeping their signature intact. As such, the conducted analysis avoids any possible issues with apps that perform self-integrity checks. This approach is thus more robust than app-rewriting based monitoring approaches that perform Dalvik code injection. Furthermore, the ability of monitoring the native code component of target apps represents a strong feature, which is lacking in many other Android dynamic analysis tools. The DDI/ADBI toolkits are previously utilized by Mulliner et al. [18] to modify the behavior of target apps related to their in-app billing transactions with Google Play. One downside of the DDI/ADBI toolkits is that they require a rooted device to run the analysis.

### 2.1.3. Static Analysis of Chat Apps

To inspect our target chat apps and their information flows, we utilize a few static analysis tools, including `Apktool` [7], jadx decompiler [8], and IDA Pro [19]. The `Apktool`, which incorporates a Dalvik disassembler called `baksmali`, is employed to extract a chat app's APK file and generate the smali code representation of the app's Dalvik code. The jadx Dalvik-to-Java decompiler is alternatively used to recover the Java source code of an app. One benefit of obtaining smali code, which corresponds to the assembly of a Dalvik bytecode, is that the code can be modified and reassembled to produce a modified app. In contrast, a decompiler usually cannot fully recover an app's Java source code, which could be due to missing high-level (e.g. type) information or any applied preventive obfuscation transformations [20]. The recovered Java source code is, nonetheless, easier to inspect than smali code due to the shown high-level programming constructs. We utilize the IDA Pro to analyze the native code component of the target apps.

To specifically inspect dataflow connection between two operations within our target apps, we use FlowDroid [9] and DroidSafe [10]. The two tools can help identify data flow from a specified source and sink method of a target app. We can thus utilize the tools to inspect potential data flow between two operations of interest. Analyzing large complex apps like WhatsApp and WeChat, however, take the two tools a long time to complete and require a machine with an ample amount of RAM. In our experiments, we managed to get some analysis results from FlowDroid after running it with several

optimization flags [21]. It is known, however, that FlowDroid may induce false negatives as well as false positives [9,22]. We employed FlowDroid in our our experiments to help point out any potential connections between operations of interest, which then provided inputs to our manual analysis of the recovered code.

### *2.2. Related Work*

There exist articles in the literature that describe how we can derive the decryption key of specific WhatsApp and WeChat versions. The work [11] lists the steps to obtain the key for WhatsApp's now-defunct `crypt7` database file. Meanwhile, [12] describes how WeChat currently forms its key by taking the first seven characters of the MD5 hash value of the WeChat user ID and the phone IMEI number. Instead of targeting specific chat-app versions, our work in this paper takes a systematic and generalized information-flow based approach to recovering the key by applying both static and dynamic analyses. Hence, unlike [12,11], our proposed approach can apply to future chat-app versions provided that they still retain their existing basic design and practice of employing third-party cryptographic libraries and passing the key information into the libraries.

There are a number of existing works that dynamically analyze Android apps to inspect the behavior of the apps. AppTrace [23] uses dynamic analysis of Android apps to identify any malware execution. The work, however, does not aim to find the decryption key of target app's database files, which is the goal of our work. ConDroid [24] considers the problem of locating critical, interesting or dangerous code, and enforcing its execution for observability. It combines a static analysis with concolic execution in order to observe an execution path that leads to a code section target. While the work enables the potential observability of a target app's code sections, it does not specifically aim to discover the target app's operations that hold or deal with decryption key of chat apps. Moreover, the technique applies only to Android app bytecode, and cannot deal with any included native code.

## 3. Decryption-Key Discovery using Information Flow Analysis and Execution Monitoring

We now explain in this section the attack models that we assume on both target chat apps, and the approach that we take in our decryption-key discovery process.

### *3.1. Attack Models*

While both WeChat and WhatsApp use encryption to protect their log database files, the encryption is used for different purposes and is executed differently. Hence, we consider different attack models for the two chat apps as follows.

#### *3.1.1. WeChat Attack Model*

WeChat encrypts its database files using SQLCipher, and stores them under the protected `/data/data/com.tencent.mm` folder. This encryption measure thus represents an extra layer of protection since, under normal circumstances on unrooted devices, all files

under the folder are accessible only to WeChat app. This measure is therefore very useful given the fact that the many Android users intentionally root their devices.

For WeChat, we thus consider the following two attack scenarios:

- ($WC_1$) *A locally-acquired, unlocked rootable device scenario*: Here, we assume that we have physically acquired an unlockable device with a WeChat app installed. Also, we assume that the device is *rootable*, either because it is already rooted by its owner, or is vulnerable to a root exploit. As a result, we can access the acquired device, including all WeChat related files, and perform a dynamic instrumentation and monitoring to be elaborated later in Section 4.1.
- ($WC_2$) *A remote, trojanized and rootable device scenario*: This scenario assumes a remote target device with an Internet connection that, along with a WeChat installed, has our trojan app running. Similar to the scenario $WC_1$, we also assume that the device is rootable. This allows the trojan app to transfer via the Internet the following information after its execution privilege elevation: WeChat database files, shared preference file, and device information (e.g. IMEI number). For this scenario, we need to first find out what methods to monitor as explained in Section 4.1, and subsequently run a password-generating app as shown in Section 5.

As can be observed, the two considered scenarios above do assume a rootable target device. One may thus argue that, given the prerequisite root privilege, other attack techniques can alternatively be employed to reveal the targeted decryption key. In our attack technique, the requirement for a rootable target device is put to solely extract WeChat protected files on the target device. Using this information, our attack can then run independently on our own device, and does not require any further interactions with the target device, which may be intrusive and could be observable by the mobile user.[1]

### *3.1.2. WhatsApp Attack Model*

Under its protected `/data/data/com.whatsapp/databases` folder, WhatsApp stores its database files as well as a key file `/data/data/com.whatsapp/files/key` in clear. As mentioned earlier, WhatsApp also applies encryption to generate a backup database file, which is normally stored on the device's SD card, for a chat-log transfer purpose. Our goal is thus to decrypt the encrypted backup database file without having access to any of WhatsApp protected files.

We consider a WhatsApp attack model that enables us to obtain the following pieces of required information:

- Encrypted WhatsApp backup database file stored on the SD card.
- Google account name that is used by mobile user, which can be obtained by invoking the Android API call `android.accounts.AccountManager.getAccountsByType("com.google")` [25].

Three following attack scenarios are therefore possible:

- ($WA_1$) *A locally-acquired, unlockable device scenario*: where we acquire an unlockable device, with a WhatsApp app installed and its SD card accessible. Since the device is unlockable, we can thus install a simple (helper) app that reveals the required Google account name.

[1] Notice that, for WhatsApp, our attack does not require a rootable target device as explained in Section 3.1.2.

- ($WA_2$) *An acquired SD card with a guessable Google account-name scenario*: where we acquire the SD card containing an encrypted WhatsApp database file, and we can guess the used Google account name.
- ($WA_3$) *A remote, trojanized device scenario*: where we target an Internet-connected remote device installed with both WhatsApp and our trojan app. This scenario allows the trojan app to send over both the required database file and Google account name.

Unlike the two WeChat scenarios $WC_1$ and $WC_2$, all the three WhatsApp scenarios $WA_1$–$WA_3$ above do not assume a rootable target device. To achieve our objective of decrypting the database file under these three scenarios, we need to first perform the key-discovery technique on our rooted device as elaborated in Section 4.2, and subsequently run a password-generating app as expounded in Section 5.

### *3.2. Chat-App Execution Monitoring and Key Inspection*

By using the DDI/ADBI toolkits, we are able to monitor the passed arguments to methods that perform database decryption operations, both at the the Dalvik and native code levels. The DDI/ADBI toolkits additionally allow us to inspect the operations at the `libc` level. Hence, we are also able to see all `libc`'s database-file related operations, together with the name of the files. For our experiments, we customized the ADBI toolkit to provide better memory-map support. We increased the size of the `char raw[]` array, as suggested in [26], from the default 80,000 to 800,000.

The main challenge faced in dynamically analyzing the target chat apps is determining the pertinent app methods that receive the decryption key strings as their arguments. There are a high number of methods in large Android apps like WhatsApp and WeChat. Hence, observing all methods is simply impractical. We thus need to shortlist the target methods to monitor by applying static analysis on the chat apps.

Another problem in dynamically analyzing the target apps is ensuring that decryption-key related operations are indeed executed by the apps. Fortunately, WeChat opens the decrypted files right after the user logs on. WhatsApp uses the key when it reads the encrypted duplicate database file as the user moves his/her chat log over to a new phone. Additionally, the key is used when the user initiates a WhatsApp menu of performing a chat-log backup, which we monitored in our experimentation.

### *3.3. Static Analysis of Key-Related Information Flow*

We have discussed the challenge of determining the app methods that receive the decryption key strings as their arguments earlier. To shortlist the target methods to monitor, we perform the following static analyses of the chat apps at both the Dalvik and native code levels, together with several applied heuristics.

First, we generate a list of all native methods that are invoked by the Dalvik code of a target chat app. For this, we use the `Apktool` to recover the smali code representation of the app's Dalvik code. In smali code, a native method is declared with `.method` followed by method access identifiers, such as `public`/`private` and `static`, and the identifier `native`. An example is the declaration of `.method private static native nativeResetCancel(IZ)V`. We can thus list all invoked native methods by searching a

regular expression of "`method .* native`". To further shortlist our target methods, we consider methods that accept strings or array of bytes in their parameters.

Second, we analyze the information flow within the app's native code by tracking the method interaction using the IDA Pro. Starting from the native methods shortlisted in the previous step, we employ the IDA Pro to observe the interaction among the native methods. Again, we can opt to consider only methods that accept strings or array of bytes in their parameters.

Third, we can also check if the native code methods that perform cryptographic operations are actually adopted or customized from a third-party library. Information of the library will thus be very helpful in understanding the cryptographic operations in use, especially if the library is an open-source software. While a customization is performed and code obfuscation may be applied, the core data structures and cryptographic methods typically do not differ much.

Lastly, we apply static analysis of the Dalvik code by using the static taint tracking analysis tools, such as FlowDroid. In this way, we can observe dataflow relationship between two operations of interest. The recovered Java code derived by a Dalvik-to-Java decompiler also provides useful information on how the Dalvik code component of a target app works.

## 4. Experimental Results

We applied the dynamic and static analysis techniques elaborated in Section 3 to WeChat and WhatsApp, and report the experimental results on a rooted device in this section.

### *4.1. WeChat Key Recovery*

We analyzed WeChat version 6.3.22, whose APK was built on July 11, 2016. To trigger the WeChat's decryption process, we need to first log out any existing user session, and then log in again.

WeChat customizes the SQLCipher code base [27] instead of simply using it. When inspecting the smali code of WeChat's Dalvik code, we could not find any reference to the standard `net.sqlcipher.database` SQLCipher package [28]. Instead, we found out a number of SQLite-related classes under the `com.tencent.kingkong` package. One such class is `SQLiteConnection`, which is related to database file opening. We thus hooked and monitored the `SQLiteConnection.<init>(...)` method, and extract its passed string parameters. In this way, we were able to retrieve the decryption key.

Furthermore, the path of the accessed database file can also be extracted from the `SQLiteDatabaseConfiguration` object that is passed as an argument to the hooked method above. The accessed WeChat database files, which are identified as parameter `path` below, and their respective decryption key, which are identified as parameter `mPassword`, are recovered as follows:

```
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/EnMicroMsg.db
SQLiteConnection.<init>:: mPassword=fe34ed7
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/enFavorite.db
SQLiteConnection.<init>:: mPassword=fe34ed7
```

```
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   ee1da3ae2100e09165c2e52382cfe79f/EnResDown.db
SQLiteConnection.<init>:: mPassword=7a6cc74
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/SnsMicroMsg.db
SQLiteConnection.<init>:: mPassword=<NULL>
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/SnsMicroMsg.db
SQLiteConnection.<init>:: mPassword=<NULL>
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/IndexMicroMsg.db
SQLiteConnection.<init>:: mPassword=<NULL>
SQLiteConnection.<init>:: path=/data/data/com.tencent.mm/MicroMsg/
   b98a831a089cef3031385d56a0f51927/CommonOneMicroMsg.db
SQLiteConnection.<init>:: mPassword=<NULL>
```

We additionally hooked a few native methods of WeChat to obtain several cipher parameters used by WeChat [29]. While it is possible to hook the relevant Java methods in order to obtain the cipher parameters, we found that we can obtain the parameters more easily by hooking a native SQLCipher method that is adopted by WeChat. For this purpose, we took advantage of the SQLCipher code base [27]. Although WeChat customizes SQLCipher, it seems to apply little modification on the core SQLCipher data structure and operations. We thus hooked the `libkkdb.sqlcipher_codec_ctx_set_pass(...)` method, and inspected its accessed database filename, decryption key, and `codex_ctx` object. The data structure of the `codex_ctx` object can be discovered by analyzing the relevant SQLCipher source file of `crypto_impl.c`.

We were able to retrieve all the parameters of SQLCipher database file decryption by inspecting the following recovered parameter values on our monitoring device:

```
sqlcipher_codec_ctx_set_cipher :: cipher_name=aes-256-cbc, return code=0
sqlcipher_codec_ctx_set_kdf_iter :: kdf_iter=4000, return code=0
sqlcipher_codec_ctx_set_pass :: key=fe34ed7, return code=0
sqlcipher_codec_ctx_set_pass :: codec_ctx->page_sz=1024
sqlcipher_codec_ctx_set_pass :: filename=/data/data/com.tencent.mm/
   MicroMsg/b98a831a089cef3031385d56a0f51927/EnMicroMsg.db
sqlcipher_codec_ctx_set_use_hmac :: use=0, return code=0

sqlcipher_codec_ctx_set_cipher :: cipher_name=aes-256-cbc, return code=0
sqlcipher_codec_ctx_set_kdf_iter :: kdf_iter=4000, return code=0
sqlcipher_codec_ctx_set_pass :: key=fe34ed7, return code=0
sqlcipher_codec_ctx_set_pass :: codec_ctx->page_sz=1024
sqlcipher_codec_ctx_set_pass :: filename=/data/data/com.tencent.mm/
   MicroMsg/b98a831a089cef3031385d56a0f51927/enFavorite.db
sqlcipher_codec_ctx_set_use_hmac :: use=0, return code=0

sqlcipher_codec_ctx_set_cipher :: cipher_name=aes-256-cbc, return code=0
sqlcipher_codec_ctx_set_kdf_iter :: kdf_iter=4000, return code=0
sqlcipher_codec_ctx_set_pass :: key=7a6cc74, return code=0
sqlcipher_codec_ctx_set_pass :: codec_ctx->page_sz=1024
sqlcipher_codec_ctx_set_pass :: filename=/data/data/com.tencent.mm/
   MicroMsg/ee1da3ae2100e09165c2e52382cfe79f/EnResDown.db
sqlcipher_codec_ctx_set_use_hmac :: use=0, return code=0
```

From the shown results above, we can conclude that WeChat performs its cipher operations using "aes-256-cbc" with a page size of 1,024 and the KDF iteration value [30] of 4,000. No HMAC construction is used for the operations. We can also see that the recovered keys confirm those that were obtained using the Dalvik-based method monitoring. We have validated the correctness of the recovered keys and parameter values by running Linux's `sqlcipher` command to open the encrypted database files and produce the clear database files.

### *4.2. WhatsApp Key Recovery and Database File Analysis*

We analyzed WhatsApp version 2.16.133, whose APK was built on June 18, 2016. Its encrypted database uses `.crypt12` extension, which is still being used as of October 2016. While discussions and decryption scripts for older `crypt7` and `crypt8` files are publicly available, there are still no available details of how the more recent crypt files can be decrypted. Similar to the decryption of the older WhatsApp database versions [11], our goal is to discover both the key and the initialization vector (IV) of the decryption operation.

In our analysis, we found out that string literals of WhatsApp code are obfuscated. In smali code representation of WhatsApp Dalvik bytecode, a string literal declaration operation looks like in the following (see also [31] for Dalvik bytecode's instruction set):

```
const-string/jumbo v0, "i|6C\r|f0C[i|6\u001e\u001dc2".
```

We thus first performed string de-obfuscation on WhatsApp bytecode classes in order to help us locate target methods related to cryptographic operations more effectively. Based on our analysis of the Java source code recovered, we discovered that each string within a class is obfuscated simply by XOR-ing it with a repeated 5-byte obfuscation string that is generated for the class.

Using our string de-obfuscation script, we could pinpoint the `com.whatsapp.util.b` class, which seems to perform encryption/decryption operations. The class contains a number of private instance fields of `javax.crypto.Cipher` class type [32]. Using dynamic analysis, we discovered all decryption parameters by monitoring "`static Cipher getInstance(String)`" and "`void init(int, Key, AlgorithmParameterSpec)`" methods [32]. The former reveals the employed "AES/GCM/NoPadding" cipher transformation, while the latter shows the key and IV used. The values recovered on our device are as follows, with the last 30 characters of the key shown as '`*`':

```
file open :: path=/data/data/com.whatsapp/databases/msgstore.db, flags=0,
   mode=0, fd=57
file open :: path=/data/data/com.whatsapp/files/key, flags=0, mode=0, fd=92
Ljavax/crypto/Cipher :: mode=1
Ljavax/crypto/Cipher :: transformation=AES/GCM/NoPadding
size of byte array is 32
Ljavax/crypto/Cipher :: key=A0D0EB3BE2A06495CD9FEDBC93EFDCEBF0************
  ******************
size of byte array is 16
Ljavax/crypto/Cipher :: IV=E6BAC22AC9712FE5A7292E852A5C3092
```

Unlike WeChat, WhatsApp manages its encrypted database file without relying on any third-party libraries, such as SQLCipher. Since its encrypted database file is meant for user chat-log transfer, WhatsApp thus also embeds additional sensitive information

into the file in order to facilitate its decryption and subsequent transfer. Understanding the structure and content of the database file is therefore important.

Our analysis on the `.crypt12` database file found that WhatsApp puts a 67-byte header before and 20-byte trailer after the database ciphertext (more information on this header and trailer below). We have successfully validated the obtained key and IV by running our decryption code on the header- and trailer-stripped database ciphertext of `.crypt12` file. The following code snippet shows how our decryption code employs the recovered cipher parameters:

```
Cipher cipher = Cipher.getInstance("AES/GCM/NoPadding")
SecretKeySpec keySpec = new SecretKeySpec(key, "AES")
GCMParameterSpec ivSpec = new GCMParameterSpec(128, IV)
cipher.init(Cipher.DECRYPT_MODE, keySpec, ivSpec)
compressedPlainBytes = cipher.doFinal(cipherBytes)

// Decompress the obtained compressedPlainBytes
decompresser = new Inflater(false)
decompresser.setInput(compressedPlainBytes)
clearDB = new FileOutputStream(clearDBFilename)
buffer = new byte[1024]
while(!decompresser.finished()){
    count = decompresser.inflate(buffer)
    clearDB.write(buffer, 0, count)
}
```

Lastly, we would like to understand the content of the encrypted database header and trailer, and how the decryption key is formed based on the contained information during a log transfer. WhatsApp does not store the decryption key within the header of its `crypt12` file. Instead, it needs to contact a WhatsApp server and retrieve the decryption key by supplying several pieces of information extracted from the header and trailer.

Our conducted analysis found out the following pieces of information embedded within the `crypt12`'s 67-byte header:

- *cipher-header preamble* (2 bytes): `0x0`, `0x1`;
- *key version* (1 byte): `0x2`;
- *server salt* (32 bytes): the stored pseudo-random salt from WhatsApp server;
- *Google account-name salt* (16 bytes): the salt used to produce *Google account-name hash*, which is to be sent to WhatsApp server (more on this below);
- *IV* (16 bytes): the initialization vector value.

Likewise, we found the following pieces of information within the 20-byte trailer, which are used by WhatsApp app solely to ensure the integrity of the database file:

- *MD5 hash value* (16 bytes): the MD5 hash value of the encrypted database file;
- *Phone no's suffix* (4 bytes): which is derived from the last few digits of the device's phone number.

When a mobile user transfers his/her encrypted WhatsApp chat log into a new device, the decryption key needs to be retrieved from WhatsApp server. WhatsApp client app on the new device sends both the *server salt* and the *Google account-name hash* to WhatsApp server. The former is extracted from the database file header. The latter is derived by applying the SHA-256 hash function on the mo-

bile user's *Google account name*, which is obtainable by invoking the Android API `android.accounts.AccountManager.getAccountsByType("com.google")` [25], and the *Google account-name salt* from the file header.

Once WhatsApp app receives the decryption key at the log transfer process, it will then store the key, and other necessary information, within a file named `/data/data/com.whatsapp/files/key`. In the WhatsApp version that we analyzed, this `key` file is 158-byte long, and contiguously stores *Java serialization data header* (27 bytes), *cipher-header preamble* (2 bytes), *key version* (1 byte), *server salt* (32 bytes), *Google account-name salt* (16 bytes), *Google account-name hash* (32 bytes), *IV* (16 bytes), and the retrieved decryption key (32 bytes). Subsequent cryptographic operations on the encrypted database file simply utilize the locally stored key.

## 5. Key Generation for Remote Target Devices

WeChat and WhatsApp decryption keys are formed based on a few pieces of device-specific information, such as the phone IMEI number, and/or values that are locally stored on the device [12,11]. When we consider attack scenarios involving a non locally-accessible target device, such as scenarios $WC_2$, $WA_2$ and $WA_3$, we thus need to simulate the target device's key generation process. Knowing what methods to recover the key as expounded in Section 4, unfortunately, is still insufficient. We additionally need to modify the chat-app behavior on our monitoring device by utilizing the information sent by the planted trojan app or guessed information. This can be done by using the following two techniques.

The first technique rewrites the target chat app. It replaces all Dalvik-level Android API invocations that obtain the device-specific information with direct value assignment operations. `Apktool` [7] can be used to generate the text-based smali code representation of the chat apps, and to subsequently derive the APK file of the altered chat app.

The second technique utilizes the dynamic instrumentation feature of the DDI/ADBI toolkits. It has a benefit over the first one in that it can intercept and instrument both Java and native methods. This feature was previously employed by [18]. The work shows how the behavior of target apps can be modified so that, instead of performing an in-app billing transaction with Google Play, the instrumented code intercepts the transaction and sends back a forged in-app payment confirmation.

In addition to app behavior modification, we need to copy all relevant local files of the target device, which are assumed to be transferred by our trojan app, into our monitoring device. Our monitoring techniques as explained in Section 4 will then be able to reveal the decryption keys of WeChat and WhatsApp running on the target device.

### *5.1. WeChat Behavior Modification*

A successful WeChat password generation of a remote device can be achieved by performing the following two steps, which are previously also pointed out by [12]:

- Modify WeChat behavior by replacing the value returned by the Android API call `android.telephony.TelephonyManager.getDeviceId()` with the IMEI number of the target device.

- Copy the shared preference files within the folder `/data/data/com.whatsapp/shared_prefs/` with those from the target device.

Using this technique, we can then reveal the targeted decryption key. The technique, in fact, will continue to work despite possible changes of the employed hash function (i.e. MD5) in future WeChat versions, as long as the key-derivation process depends upon only the IMEI number and locally stored files. This represents an advantage of our employed dynamic analysis over the static analysis used by prior work [12].

### *5.2. WhatsApp Behavior Modification*

For WhatsApp password generation of a non locally-acquired device, we need to perform the following two steps:

- Copy the target device's database file to the monitoring device's SD card.
- Modify WhatsApp to return the *Google account name* of mobile user in the target device. In this way, the altered app will retrieve the key from WhatsApp server using the *Google account-name hash* of the target device as explained earlier.

Notice that the key request issued by the monitoring device to WhatsApp server will succeed if the server does not store and verify the association between the used Google account name and phone number of a mobile device. In our experiments, we decided not to probe WhatsApp server and ascertain this provisioning. Instead, we opt to forewarn the security community of this potential security weakness pertinent to a chat-app design that allows for a log transfer using an encrypted database. We further discuss the applicability and ethical considerations of WhatsApp password generation later in Section 6.3.

## 6. Discussions

### *6.1. Weakness Root Causes*

Given the potential impact of our decryption-key recovery on the analyzed chat apps, we conducted an analysis of why our technique is possible. The following are our key observations on why our analysis technique can still attack widely-popular chat apps.

- The decryption-key construction is sometimes performed within the Dalvik code of the chat apps. As Dalvik bytecode can be easily reverse engineered and analyzed, the key construction steps can thus be inspected easier.
- The formed key is passed as a parameter in clear among the app methods. As such, dynamic monitoring can easily reveal the passed key and other accompanying pieces of sensitive information in their clear final form.
- The inclusion or customization of third-party libraries, particularly those related to database encryption/decryption, still largely maintains the key data structures and main operational methods of the libraries. Inspection of the libraries, especially the open-source ones, will therefore make code analysis and dynamic monitoring of the chat apps become significantly easier.
- The decryption-key formulation of the analyzed chat apps depends on deterministic device-dependent information, such as IMEI number as in WeChat or Google account name as in WhatsApp. An attack technique like ours can therefore simulate a key generation process of a remote target device as explained in Section 5.

Additionally, for chat apps that allow for a log transfer to a different device like WhatsApp, the following represent insecure or imprudent practices that can contribute to a successful decryption of the encrypted database by an attacker:

- Lack of a record keeping and verification by the chat-app server on the association between a mobile device's phone number and user account information.
- Lack of privacy protection that prevents the reading of information stored within the encrypted file's header.

### *6.2. Recommended Counter Measures*

The following are counter measures that can be adopted by chat-app developers to strengthen their future app versions:

- The key-construction step can be done within the last native method performing/invoking the decryption operation. This way, dynamic analysis can thus inspect only the flow of individual pieces of information that finally constitute the key into the native method.
- Inclusion or customization of a third-party library needs to substantially alter the key data structures and main operational methods related to the decryption key. Obfuscation techniques, particularly the layout and data obfuscation techniques, can be very useful for this.
- Control-flow obfuscation can also be applied to the Dalvik code so that static taint-tracking analysis would find it more difficult to inspect the flow of device-specific information that is used for key construction.
- The Dalvik code may also spuriously flow the relevant sensitive device-specific information in order to complicate the information-flow analysis of the chat apps.
- Chat-app developers may also explore the possibility of incorporating a random value in formulating a decryption key. This measure will therefore hinder attacks that simulate the key-generation process of a remote device, such as ours. There exists, however, an issue of securely storing such a random value. If the value is stored locally on a mobile device, its confidentiality thus needs to be protected. It could be encrypted, for example, using the mobile user's account password.

For a chat-app model that provisions a log transfer to a different device, and whose remote server issues the decryption key upon request by a device, the following counter measures can additionally be exercised:

- The chat-app server needs to keep track of the association between a mobile device's phone number and user account information. The server returns the requested key only if the two pieces of information supplied in a query match.
- The database file header can be encrypted using a user-chosen password. A chat-log transfer will proceed correctly only if the user enters the correct password.

### *6.3. Ethical Considerations and Applicability of Our Attacks*

We discuss below ethical considerations of our techniques and obtained results. We additionally review the applicability conditions of our techniques, which also relates to the security implications of the two analyzed chat apps.

We believe that our results sufficiently highlight potential security issues with the design of current popular chat apps. We have not contacted the developers of WeChat and WhatsApp to inform them of our findings for the following reasons. Techniques to decrypt the current version of WeChat and older versions of WhatsApp are already known [12,11]. Additionally, while our technique expounds a systematic and generalized approach to recovering a target chat app's key and decrypting its log, some attack prerequisites (as summarized below) do exist. Instead, we choose to share our results in this publication so as to reach a wider audience within the security community. Furthermore, we also suggest several counter measures that can be applied to prevent the attacks. This is in line with a practice accepted by the security community to responsibly disseminate potential security weaknesses, and suggest measures to address them.

When good security precautions are fully exercised by mobile users, our techniques do not directly apply. In our experiments on WeChat reported in Section 4.1, we were able to obtain its encrypted database files since the target device was rooted. Likewise, WhatsApp users must physically safeguard the SD card of their devices. They also should keep their Google account names secret, and make them difficult to guess. Lastly, the key retrieval from WhatsApp server works only if it insecurely issues an inquired key belonging to other number. Nevertheless, our attacks clearly highlight potential threats to mobile user privacy that could work on rooted or exploitable devices in conjunction with planted trojan apps, which are not uncommon.

Given the importance of our findings to chat-app security, we hope that our techniques and results presented in this paper can help the security community become aware of the privacy threats. We also hope that our paper can provide useful insights on how the security community can move forward together in securing future chat apps.

## 7. Conclusion

We have presented our information-flow based approach to discovering the decryption key of log database files from two highly popular chat apps, namely WhatsApp and WeChat. We have shown that, despite the employed code-obfuscation technique, we can discover the keys and obtain useful information on the key construction process. We have additionally detailed the employed string de-obfuscation, encrypted database file structure, and decryption-key formulation of the latest WhatsApp version. Moreover, we have additionally elaborated how we can construct a chat-app version that can simulate other devices' key generation process. Given our generalized approach to discovering and generating the decryption key, the proposed technique can still apply to different chat-app versions as long as they still retain the same design and practice of employing third-party cryptographic libraries and passing the key information into the libraries. Lastly, we have provided an in-depth analysis of why our technique can work on widely-popular chat apps, and listed measures that can be adopted by chat-app developers to improve their future app versions and better protect the privacy of billions of their users.

## Acknowledgment

This material is based on research work supported by the Singapore National Research Foundation under NCR Award No. NRF2014NCR-NCR001-034.

## References


[1] WhatsApp Messenger, https://play.google.com/store/apps/details?id=com.whatsapp.
[2] WeChat, https://play.google.com/store/apps/details?id=com.tencent.mm.
[3] C. Mulliner, Android DDI: Dynamic Dalvik Instrumentation, 30th Chaos Communication Congress, http://www.mulliner.org/android/feed/mulliner_ddi_30c3.pdf, 2013.
[4] C. Mulliner, ddi – Dynamic Dalvik Instrumentation Toolkit, https://github.com/crmulliner/ddi.
[5] C. Mulliner, Binary Instrumentation on Android, SummerCon, 2012, http://www.mulliner.org/android/feed/binaryinstrumentationandroid_mulliner_summercon12.pdf.
[6] C. Mulliner, adbi – The Android Dynamic Binary Instrumentation Toolkit, https://github.com/crmulliner/adbi.
[7] Apktool, http://ibotpeaches.github.io/Apktool.
[8] jadx – Dex to Java decompiler, https://github.com/skylot/jadx.
[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps, 35th Conference on Programming Language Design and Implementation (PLDI), 2014.
[10] M. I. Gordon, D. Kim, J. Perkins, L. Gilham, N. Nguyen, and M. Rinard, Information Flow Analysis of Android Applications in DroidSafe, Network and Distributed System Security (NDSS), 2015.
[11] M. Ibrahim, How to Decrypt WhatsApp crypt7 Database, Digital Internals, http://www.digitalinternals.com/security/decrypt-whatsapp-crypt7-database-messages/307, May, 2014.
[12] F. M. Darus, How to decrypt WeChat EnMicroMsg.db database?, Forensic Focus, http://articles.forensicfocus.com/2014/10/01/decrypt-wechat-enmicromsgdb-database, 2014.
[13] Wikipedia, WeChat, https://en.wikipedia.org/wiki/WeChat.
[14] SQLCipher, https://www.zetetic.net/sqlcipher.
[15] Wikipedia, WhatsApp, https://en.wikipedia.org/wiki/WhatsApp.
[16] WhatsApp Blog, One billion, https://blog.whatsapp.com/616/One-billion, February, 2016.
[17] WhatsApp, Frequently Asked Questions – How do I move my chat history over to my new Android phone?, https://www.whatsapp.com/faq/en/android/20902622.
[18] C. Mulliner, W. Robertson, and E. Kirda, VirtualSwindle: An automated attack against in-app billing on Android, 9th ACM Symposium on Information, Computer and Communications Security (ASIACCS '14), 2014.
[19] IDA Pro, https://www.hex-rays.com/products/ida.
[20] C. Collberg, C. Thomborson, and D. Low, A taxonomy of obfuscating transformations, Technical Report, 148, University of Auckland, Auckland, New Zealand, 1997.
[21] soot-infoflow-android wiki, https://github.com/secure-software-engineering/soot-infoflow-android/wiki.
[22] E. Khalaj, R. Vanciu, and M. Abi-Antoun, Comparative evaluation of static analyses that find security vulnerabilities, Technical Report, Wayne State University, Detroit, MI, 2014.
[23] L. Qiu, Z. Zhang, Z. Shen, and G. Sun, AppTrace: Dynamic trace on Android devices, International Conference on Communications (ICC), 2015.
[24] J. Schutte, R. Fedler, and D. Titze, ConDroid: Targeted dynamic analysis of Android applications, 29th International Conference on Advanced Information Networking and Applications (AINA), 2015.
[25] Android Open Source Project, AccountManager, http://developer.android.com/reference/android/accounts/AccountManager.html.
[26] adbi – Issues, Too many memory mapping, https://github.com/crmulliner/adbi/issues/5.
[27] SQLCipher, sqlcipher/sqlcipher, https://github.com/sqlcipher/sqlcipher/tree/910fd70a3fdeeaa937e0d26940f924dbefe7ba77/src.
[28] SQLCipher, SQLCipher for Android application integration, https://www.zetetic.net/sqlcipher/sqlcipher-for-android.
[29] SQLCipher, SQLCipher API, https://www.zetetic.net/sqlcipher/sqlcipher-api.
[30] Wikipedia, PBKDF2, https://en.wikipedia.org/wiki/PBKDF2.
[31] Android Open Source Project, Dalvik bytecode, https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.
[32] Oracle, javax.crypto – Class Cipher, https://docs.oracle.com/javase/7/docs/api/javax/crypto/Cipher.html.