Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones

Cosimo Anglano[a,*], Massimo Canonico[a], Marco Guazzone[a]

[a] *DiSIT - Computer Science Institute, University of Piemonte Orientale, Alessandria (Italy)*

Abstract

We present the forensic analysis of the artifacts generated on Android smartphones by *ChatSecure*, a *secure* Instant Messaging application that provides strong encryption for transmitted and locally-stored data to ensure the privacy of its users.

We show that ChatSecure stores local copies of both exchanged messages and files into two distinct, AES-256 encrypted databases, and we devise a technique able to decrypt them when the secret passphrase, chosen by the user as the initial step of the encryption process, is known.

Furthermore, we show how this passphrase can be identified and extracted from the volatile memory of the device, where it persists for the entire execution of ChatSecure after having been entered by the user, thus allowing one to carry out decryption even if the passphrase is not revealed by the user.

Finally, we discuss how to analyze and correlate the data stored in the databases used by ChatSecure to identify the IM accounts used by the user and his/her buddies to communicate, as well as to reconstruct the chronology and contents of the messages and files that have been exchanged among them.

For our study we devise and use an experimental methodology, based on the use of emulated devices, that provides a very high degree of reproducibility of the results, and we validate the results it yields against those obtained from real smartphones.

*Corresponding author. Address: viale T. Michel 11, 15121 Alessandria (Italy). Phone: +39 0131 360188.

Email addresses: `cosimo.anglano@uniupo.it` (Cosimo Anglano), `massimo.canonico@uniupo.it` (Massimo Canonico), `marco.guazzone@uniupo.it` (Marco Guazzone)

Keywords: Mobile forensics, ChatSecure, Android, Instant Messaging, Secure Instant Messaging

1. Introduction

Instant Messaging (*IM*) applications are very popular among smartphone users because of the level of convenience they provide in interpersonal communications. Quite sophisticated IM applications are available today for the prominent smartphone platforms (e.g., Android, iOS, and Windows Phone, to name a few) that allow users to exchange text and files in (pseudo) real time.

In addition to legitimate uses, however, IM applications are increasingly being used to carry out illicit activities (United Nations Office on Drugs and Crime, 2013). Therefore, the forensic analysis of these applications has received considerable attention in the recent past. IM-based evidence may indeed prove crucial in all those cases where an IM application has been used by the parties involved in a crime, i.e. by a perpetrator to interact with its victims, or by criminals in the attempt to escape interception when they communicate.

Generally speaking, the forensic analysis of an IM application is based on the availability of various types of artifacts (metadata and content of exchanged messages and files, log files, etc.) stored by that application on the local storage of smartphones. By locating, extracting, and analyzing such artifacts, quite often it is possible to recover a significant amount of potential evidence (Al Barghuthi and Said, 2013; Anglano, 2014; Husain and Sridhar, 2010).

This situation, however, is rapidly changing. The increasing awareness of the fact that IM communications may be intercepted when transiting over the infrastructure of the service provider, is stimulating the interest towards *secure* IM applications (Guardian Project, 2015a; Telegram Inc, 2015; Open Whispers System, 2015; Gliph Inc, 2015; Wickr Inc, 2015). These application, unlike standard ones, provide suitable privacy-preserving and user security mechanisms, such as strong encryption for transmitted and locally-stored data, secure user authentication, plausible deniability, forward secrecy, and so on. Secure IM applications pose new challenges to the forensic analyst, that has to deal with the issues posed by the privacy-preserving features of these applications.

Among these applications, one that is receiving increasing attention is *ChatSecure* (Guardian Project, 2015a), which is available both for Android and iOS. There are various reasons for its success: (a) it is open-source (so it is possible to audit its code), (b) it provides message encryption, partner authentication, deniability and perfect forward secrecy thanks to the use of the *Off-The-Record (OTR)* (Borisov et al., 2004) messaging system (which has gained an excellent reputation in the privacy-concerned user communities), (c) it encrypts locally stored data with *SQLCipher* (Zetetic LLC., 2015) and *IOCipher* (Guardian Project, 2016a), and (d) it has been ranked as one of the most secure IM applications by the *Electronic Frontiers Foundation (EFF)* (Electronic Frontier Foundation).

Given these characteristics, the interest on the forensic analysis of ChatSecure is evident, since there is no publicly known way of decrypting OTR-encrypted data once they are in transit over the network. Thus, forensic analysis of the devices used to communicate with ChatSecure may be the only option available to retrieve IM-based evidence.

To the best of our knowledge, there is no published work addressing the forensic analysis of ChatSecure on the Android platform. In this paper we fill this gap by describing which artifacts ChatSecure are stored in the local memory of the device, and how they can be decoded and correlated among them to infer information of potential investigative interest.

The original contributions of this paper can be summarized as follows:

- we show that ChatSecure stores locally copies of all the messages and files that are exchanged between the user and her contacts into two encrypted *SQLite v.3* (SQLite Consortium, 2013) databases;
- we analyze the encryption procedure used for these databases, and we develop and implement an algorithm able to decrypt them using the secret passphrase set by the user;
- we show how the passphrase can be retrieved from the volatile memory of an Android device;
- we discuss the decoding and the interpretation of all the artifacts generated by ChatSecure, and we show how they can be correlated to perform various forensic reconstructions, such as the chronology and contents of exchanged files and messages, the set of IM accounts used by the ChatSecure user, as well as the list of contacts associated with each one of them;

- we show that it is not possible to recover the information deleted by ChatSecure users because of the use of secure deletion techniques in SQLCipher and IOCipher;
- we devise and use an experimental methodology, based on the use of emulated devices, that provides a very high degree of reproducibility of the results, and we validate the results it yields against those obtained from real smartphones.

The rest of the paper is organized as follows. In Sec. 2 we review existing work, while in Sec. 3 we describe the methodology and the tools we use in our study. Then, in Sec. 4 we discuss the forensic analysis of ChatSecure and, in Sec. 5, we conclude the paper.

2. Related works

Smartphone forensics has been widely studied in the recent literature, which mostly focuses on Android and iOS forensics (Tamma and Tindall, 2015; Epifani and Stirparo, 2015), given the pervasiveness of these platforms. As a result, well known and widely accepted methodologies and techniques are available today that are able to properly deal with the extraction and analysis of evidence from smartphones. In this paper we leverage this vast body of work for extracting and analyzing the data generated by ChatSecure during its usage.

The importance of the forensic analysis of smartphone IM applications has been also acknowledged in the literature, where a significant number of papers on this topic has been published. (Anglano, 2014) discusses the forensic analysis of WhatsApp Messenger. (Husain and Sridhar, 2010) focuses on the forensic analysis of three IM applications (namely AIM, Yahoo! Messenger, and Google Talk) on the iOS platform. (Al Barghuthi and Said, 2013) presents the analysis of several IM applications on various smartphone platforms, aimed at identifying the encryption algorithms used by them. (Tso et al., 2012) discusses the analysis of iTunes backups for iOS devices aimed at identifying the artifacts left by various social network applications. (Walnycky et al., 2015) discusses the analysis of the data transmitted or stored locally by 20 popular Android IM applications.

None of these papers, however, covers the forensic analysis of ChatSecure on Android platforms, which is instead the focus of this paper. The closest work to ours is (Quarkslab SAS, 2015), that is focused on the iOS platform.

However, the results discussed there do not apply to Android, given the significant differences existing between the Android version and the iOS version of ChatSecure.

3. Analysis methodology and tools

The study described in this paper has been performed by carrying out a set of controlled experiments, each one referring to a specific usage scenario (one-to-one communication, group communication, file exchange, etc.), during which typical user interactions have taken place. After each experiment, the internal memory of the sending and receiving devices has been examined in order to identify, decode, and analyze the data generated by ChatSecure in that experiment. In all the experiments, we run ChatSecure v. 14.2.3 (the last one available on Google Play at the moment of this writing).

The data generated by ChatSecure are stored into an area of the internal device memory that is normally inaccessible to users (see Sec. 4.2). Therefore, suitable methodologies and tools need to be adopted in order to access and acquire this area. Tools like UFED (Cellebrite LTD, 2016), XRY (MSAB, 2016), and Oxygen Forensics Detective (Oxygen Forensics, Inc, 2016), among others, are able to perform this acquisition in a forensically-sound manner.

However, this approach presents some limitations, namely:

- **limited generality**: to gain confidence into the generality of the results, a suitably large number of devices and Android versions should be used for the experiments. The resulting high costs both in terms of purchase and of the time required to replicate the experiments on a large set of devices, however, practically limits the number of devices used for the experiments, thus potentially casting doubts on the generality of the results;

- **limited replicability**: a third party wanting to reproduce the results needs to use the same set of devices, operating systems versions, and forensic acquisition tools to repeat experiments This, however, may be problematic, both because of device availability and of the cost of the acquisition tools;

- **limited controllability**: smartphones are complex devices, running a multitude of applications and services, whose behavior and interactions are hard to characterize. As a consequence, it may be difficult not only to reproduce the exact conditions holding at the moment of each experiment, but also to exclude with certainty possible data cross-contaminations among different applications that use the same file system (as in Android).

To overcome the above limitations, in this work we carry out experiments using emulated mobile devices instead of physical ones. In particular, we use the *Android Mobile Device Emulator* (Google, 2016c) to create various *Android Virtual Devices* (AVDs), that are emulated smartphones behaving exactly like real physical devices that can be customized with different hardware characteristics and Android versions. The status of AVDs can be monitored by means of the *Android Device Monitor* (Google, 2016b) (ADM, in the following).

The use of emulated devices provides many advantages, and allows us to overcome the limitations discussed above. First, generality of results is benefited since it is simple and cost-effective to run experiments on a variety of different AVDs (featuring different hardware and software combinations), and to quickly extract the contents of their internal memory. Second, also replicability is greatly benefited, since a third-party can configure AVDs exactly as we did, thus reproducing the same conditions of our experiments. Finally, also controllability is enhanced: the configuration of AVDs (that include both hardware/software features, as well as a set of services and apps running in the background) is under total control of the experimenter by means of the ADM, thus allowing us (as well as a third-party replicating our experiments) to precisely determine the operational conditions holding on each AVD at the moment of the experiment.

For our experiments, we use the three AVDs configurations shown in Table 1 below, that are characterized by different Android versions, processor families, and volatile and persistent storage sizes. To carry out our analysis,

Table 1: Characteristics of the AVDs used in the experiments.

Characteristics of AVDs used for experiments			
Processor	*RAM* (MB)	*Internal storage (MB)*	*Android version*
ARM (armeabi-v7a)	512	2047	4.4 (API 19)
Intel Atom (x86)	1536	1024	5.1 (API 22)
Intel Atom (x86_64)	1536	1024	6.0 (API 23)

we run all the experiments on these AVDs and, at the end of each experiment, we extract the data generated by ChatSecure using the *pull* functionality of the *File Explorer* of the ADM, that allows one to recursively extract entire folders, or individual files. Alternatively, this task can be carried out using the *Android Debug Bridge* (Google, 2016a) (ADB, in the following) to pull data out from the AVD using a command-line interface.

In order to validate the results obtained with AVDs, we compare them against results obtained running experiments on a real device. More precisely, we run experiments on a Samsung SM-G350 Galaxy Core Plus smartphone running Android 4.4.2, and we use the *Cellebrite UFED4PC* platform (Cellebrite LTD, 2015b) to perform device memory extraction, and the UFED *Physical Analyzer* (Cellebrite LTD, 2015a) to decode its contents. In all the experiments we performed, the results collected from this smartphone were identical to those obtained from the emulated devices we considered.

In addition to the analysis of the persistent device memory, we also examine its volatile memory to verify whether encryption keys or secret passwords are stored there, and to identify and extract them. In particular, as discussed in Sec. 4.7, we use LiME (504ENSICS Labs, 2016) to dump the contents of the volatile memory of the AVDs used in the experiments, and Volatility (Volatility Foundation, 2016) to analyze these dumps. We perform memory analysis experiments only for the ARM architecture (row 1 of Table 1) since, at the moment of this writing, LiME supports this architecture only. Note that in order to work, LiME requires the device to be rooted. For AVDs, however, this is not an issue, since they are pre-configured to allow root access to the user.

We do not validate the memory analysis results against real devices, since LiME requires an Android kernel supporting dynamic module loading. To enable this functionality, the kernel source must be reconfigured and recompiled. For a real smartphone, the stock Android kernel is not sufficient, as vendors typically customize it to suit the specific hardware configuration of the device. Unfortunately, the kernel source for the Samsung SM-G350 Galaxy Core Plus smartphone we used for validation was not available to us, so we could not configure such a device to work with LiME.

Finally, the source code of ChatSecure (which is freely available from (Guardian Project, 2015b)) has been examined to verify our hypothesis about its behavior, or to understand how to decode the data it generates.

For the sake of reproducibility of the experiments we discuss in this paper, in (Anglano et al., 2016) we describe how to concretely configure and use the various tools that we rely upon to create and run an AVD, and to carry out the analysis of its persistent and volatile memory.

4. Forensic analysis of ChatSecure

ChatSecure is an IM application that allows its users to communicate securely via their existing accounts on IM providers that use the XMPP (XMPP Standards Foundation, 2015) protocol (e.g. *Google Talk* or *Jabber*).

To ensure privacy, ChatSecure provides end-to-end message encryption with OTR and encrypts with *SQLCipher* [1] and *IOCipher* [2] the SQLite databases it uses to store the information it generates. Furthermore, it can provide user untraceability by means of the TOR network (TOR Project, 2016) via the *Orbot* application (Guardian Project, 2016c).

A ChatSecure user may define several IM accounts (corresponding to one or more IM providers), and use them at the same time to communicate with a set of *buddies* (i.e., other IM accounts (s)he is in contact with).

ChatSecure provides the typical functionalities of all IM applications, namely: (a) contact management (i.e., inviting and removing contacts, accepting or denying invitations, etc.), (b) point-to-point communication, (c) group chats creation and participation, and (d) file transfer, as well as additional functionalities related to security management (toggling OTR encryption on and off, verification of the partner identity, etc.).

In this section we provide a detailed forensic analysis of ChatSecure that is aimed at identifying all the relevant artifacts it generates, interpreting them, and using them to reconstruct the activities carried out by its users.

In particular, after describing a fictitious scenario to give an investigative context to the analysis techniques described in this paper, we describe the set of artifacts generated by ChatSecure, and where they are stored on the memory of the device (Sec. 4.2). Next, we illustrate how to reconstruct the set of ChatSecure accounts utilized by the user (Sec. 4.3), as well as the set of the corresponding buddies (Sec. 4.4). Then, we move to the reconstruction of the chronology and contents of exchanged messages (Sec. 4.5) and files (Sec. 4.6). After that, we deal with the problem of decrypting the SQLite

[1]SQLCipher is open-source extension to SQLite that provides transparent 256-bit AES encryption of SQLite database files

[2]IOCipher is a library that implements encrypted virtual disks using an SQLCipher database (more details can be found in Sec. 4.7)

databases where ChatSecure artifacts are stored (Sec. 4.7) by describing both a decryption algorithm we devised, and how the passphrase allowing decryption can be retrieved from the volatile memory of the device. Finally, we conclude with Sec. 4.8, where we report our findings concerning the deletion of the data generated by ChatSecure, that show the impossibility of recovering them after they have been deleted.

4.1. Investigative scenario

To illustrate how the techniques discussed in this paper can be applied in the context of a digital investigation, we consider the fictitious investigative scenario reported below.

We consider the case where ChatSecure is found to be installed on a seized Android smartphone, and we need to answer the following set of typical investigative questions (for each one of them, we also indicate the section of the paper where we discuss how to obtain the corresponding answer):

1. How many distinct XMPP accounts did the user configure and use with ChatSecure? (Sec. 4.3)
2. Who are the ChatSecure contacts of the local user? (Sec. 4.4)
3. What messages have been exchanged with each one of the above contacts, and when did each communication occur? (Sec. 4.5)
4. Did the local user exchange any file with its contacts? If so, when did these exchanges occur? What is the content of the files that have been exchanged? (Sec. 4.6)
5. How to decrypt ChatSecure databases? (Sec. 4.7)
6. How to recover deleted data? (Sec. 4.8).

4.2. Location and format of ChatSecure artifacts

During its use, ChatSecure stores several artifacts into various files and databases that are located into the *info.guardianproject.otr.app.im* folder. This folder is located into the */data/data* directory of the Android file system, [3] that contains the subfolders shown in Fig. 1.

The data of forensic interest generated by ChatSecure are the following ones:

[3]This directory corresponds to the *user data* partition of the internal device memory and is inaccessible to standard users, unless the smartphone has been rooted.

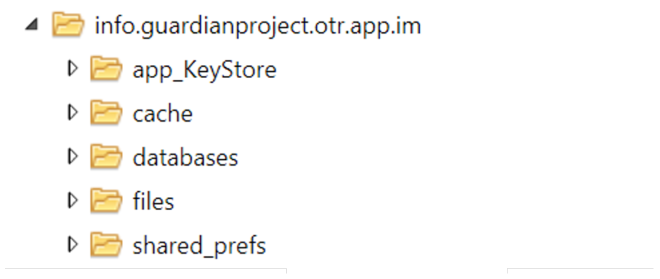


Figure 1: Structure of the main folder of ChatSecure.

- The *main database*, where ChatSecure stores the information concerning the accounts used by the ChatSecure, the list of the corresponding buddies, and local copies of the messages that have been exchanged. It consists in an SQLCipher-encrypted SQLite v.3 database, named *impsenc.db*, which is stored in the *databases* folder and contains 21 different tables. As resulting from our findings, only 11 out of these 21 tables contain information of forensic interest, namely tables *accounts*, *accountStatus*, *providers*, *providerSettings*, *contacts*, *contactList*, *presence*, *avatars*, *chats*, *messages*, and *inMemoryMessages*. The information stored in these tables, as well their structure, interpretation, and analysis, are discussed in Secs. 4.3–4.5, where we show how to use the data they store to perform various forensic reconstructions. The analysis of these tables has been performed by inspecting the source code of ChatSecure, [4] and by performing a set of controlled experiments in order to validate our findings. It is worth pointing out that the user may opt for not using encryption; this decision must be taken when ChatSecure is started for the first time after installation, and cannot be undone (in this case the database is named *imps.db*).

- The *encrypted virtual disk*: in addition to exchanged messages, ChatSecure stores locally also copies of the files that its user has exchanged with her contacts. To prevent unauthorized parties from accessing these files, ChatSecure stores them into an encrypted virtual disk, which is implemented via IOCipher. The analysis of this encrypted virtual disk is discussed in Sec. 4.6.

[4]In particular, files *Imps.java* and *ImpsProvider.java*, stored in the *src/info/guardian-project/otr/app/im/provider* directory of the ChatSecure source tree.

- The *stored secret file*: ChatSecure stores the information it needs to decrypt the main database and the virtual disk into a file, named *info.guardianproject.cacheword.prefs.xml*, which is located in folder *shared_prefs*. In Sec. 4.7 we show how this information can be decoded and used to carry out the above decryption.

For the sake of completeness, we mention also files *account.xml* (storing the information concerning the ChatSecure account) and *info.guardianproject.otr.app.im_preferences.xml* (storing ChatSecure settings and preferences), located in the *shared_prefs* folder.

4.3. Reconstructing user accounts

As mentioned before, ChatSecure allows its users to create various IM accounts, each one corresponding to a specific IM provider. From the investigative point of view, the information about all the active accounts is relevant for various reasons, including determining the identity of the providers to which additional sources of evidence (e.g., log files) can be asked, and correlating evidence with that retrieved from the devices of other ChatSecure users with whom the user has exchanged communications.

The information associated with each account (name, credentials, etc.) are stored in the main database, where they are spread across four distinct tables (namely, *accounts*, *accountStatus*, *providers*, and *providerSettings*). The structure of these tables and the meaning of their fields are reported in Tables 2–5.

Tables *accounts* and *accountStatus* jointly store the information concerning the IM accounts created by the ChatSecure user. In particular, *accounts* stores the properties of these accounts (e.g., the credentials for the authentication), while *accountStatus* stores the information concerning their *status*. These two tables are linked together by means of the foreign key of table *accountStatus* (i.e., field *account*, see Table 3).

Tables *providers* and *providerSettings*, instead, jointly store the information about the IM providers corresponding to the above IM accounts. In particular, the former table stores the information about these providers, while the latter one stores the settings of each provider (one record per setting). These two tables are joined together by means of the foreign key of table *providerSettings* (i.e., field *provider*, see Table 5).

To illustrate how to reconstruct the information concerning the ChatSecure accounts, let us consider the scenario shown in Fig. 2.

Table 2: Structure of the *accounts* table.

Table *accounts*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique record identifier
name	–	text	name of the account as chosen by the user
provider	Foreign Key	int	value of the *_id* field of the record, in table *providers*, corresponding to the provider of this IM account
username	–	text	username (on the service provider) for the account
pw	–	text	password (on the service provider) for the account
active	–	int	1 if the account is active, 0 otherwise
locked	–	int	1 if the account is locked (i.e., it is not editable), 0 otherwise
keep_signed_in	–	int	1 if ChatSecure keeps the account logged in between executions, 0 otherwise
last_login_state	–	int	either 0 or 1

Table 3: Structure of the *accountStatus* table.

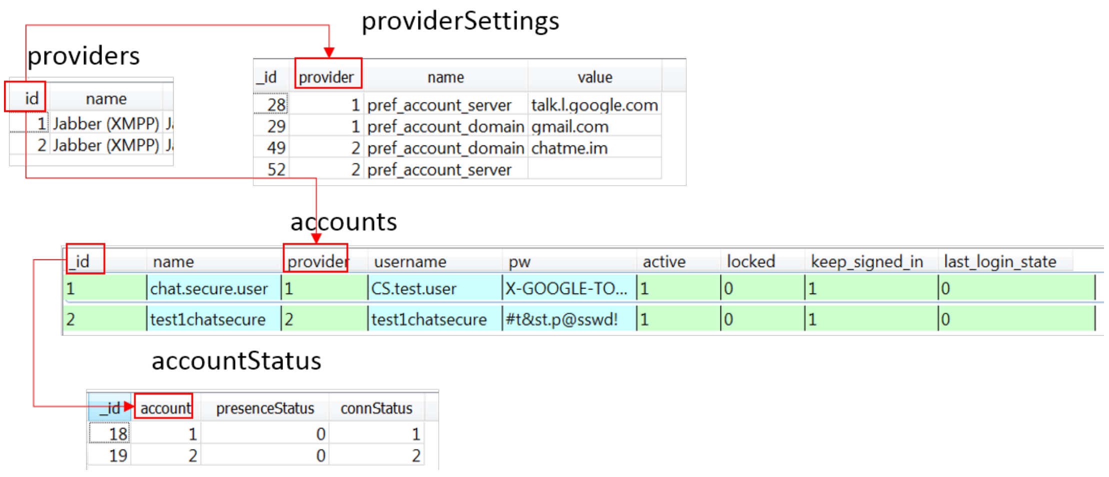
Table *accountStatus*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique record identifier
account	Foreign Key	int	value of the *_id* field of the record, in the *accounts* table, corresponding to the account this status information refers to
presenceStatus	–	int	visibility of the account to its buddies. Possible values are: 0 (offline), 1 (invisible), 2 (away), 3 (idle), 4 (do not disturb), and 5 (available)
connStatus	–	int	status of the connection to the XMPP provider. Possible values are: 0 (offline), 1 (connecting), 2 (suspended due to temporary network unavailability), and 3 (online)

Table 4: Structure of the *providers* table (fields *category* and *signup_url* have been omitted because of their lack of forensic value).

Table *providers*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique record identifier
name	–	text	name of the IM provider (e.g. GTalk, AIM, etc.)
fullname	–	text	full name of the IM provider

Table 5: Structure of the *providerSettings* table.

Table *providerSettings*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique record identifier
provider	Foreign Key	int	value of the *_id* field of the record, in the *providers* table, corresponding to the IM provider this setting refers to
name	–	text	name of the setting
value	–	text	value of the setting


providers

id	name	
1	Jabber (XMPP)	J
2	Jabber (XMPP)	J

providerSettings

_id	provider	name	value
28	1	pref_account_server	talk.l.google.com
29	1	pref_account_domain	gmail.com
49	2	pref_account_domain	chatme.im
52	2	pref_account_server	

accounts

_id	name	provider	username	pw	active	locked	keep_signed_in	last_login_state
1	chat.secure.user	1	CS.test.user	X-GOOGLE-TO...	1	0	1	0
2	test1chatsecure	2	test1chatsecure	#t&st.p@sswd!	1	0	1	0

accountStatus

_id	account	presenceStatus	connStatus
18	1	0	1
19	2	0	2

Figure 2: Reconstruction of ChatSecure user accounts.

This scenario features two distinct ChatSecure accounts, named *chat.secure.user* and *test1chatsecure* (see the records stored in table *accounts* in Fig. 2). Both these accounts are *active* (i.e., they are currently used), as indicated by the value 1 stored in fields *active*, and are never logged out of the respective IM provider during a ChatSecure session, as indicated by the value 1 stored in fields *keep_signed_in*.

To determine the IM provider associated with each account we need to join tables *accounts* and *providers*. The results of this operation show that user *chat.secure.user* is associated with IM provider no. 1 and that authenticates with it using username *CS.test.user* and password "*X-GOOGLE-TOKEN...*" (password shortened for readability purposes), while user *test1chatsecure* is associated with IM provider no. 2 and authenticates with it using username *test1chatsecure* and password "*#t&st.p@sswd!*". Note that the passwords used to authenticate with IM providers are stored in cleartext, and as such may be readily used to authenticate with the IM providers *outside the ChatSecure application* once recovered from this database.

To determine the identity of the IM providers used by the various accounts we need to join tables *providers* and *providerSettings*. The results of this operation indicate that provider no. 1 corresponds to Google's *GTalk* IM service (see record no. 28 of table *providerSettings* in Fig. 2), and that its account domain is *gmail.com* (see record no. 29 of table *providerSettings*). From this, we determine that the ChatSecure user *chat.secure.user* corresponds to the GTalk user *CS.test.user@gmail.com*. Furthermore, we also determine that provider no. 2 corresponds to the *ChatMe* IM service (it uses the *chatme.im* server), and that its account domain is *chatme.im*, meaning that ChatSecure user *test1chatsecure* corresponds to ChatMe user *test1chatsecure@chatme.im*.

Finally, to determine the status of each account we need to join tables *accounts* and *accountStatus*. From the results of this operation we determine that both users *chat.secure.user* and *test1chatsecure* are offline because of the unavailability of the network when the memory of the device was acquired (see the values stored in fields *presenceStatus* and *connStatus* in the records of table *accountStatus*).

4.4. Reconstructing contact lists

Each ChatSecure account is typically associated with a set of contacts, i.e. remote users with whom (s)he can exchange messages and files. The evidentiary value of contact information is notorious, as it allows an investigator to determine who the user was in contact with.

Each one of the contacts is associated with its *nickname* (i.e., the name used by the ChatSecure user to denote the buddy), its *username* (that identifies the contact on the corresponding IM provider), and an optional *avatar* (i.e., a picture, chosen by the corresponding user, which is downloaded by ChatSecure and displayed together with the nickname).

The information concerning user contacts is stored in the main database, and is spread across four distinct tables, namely *contacts*, *avatars*, *presence*, and *contactsList*, whose structure and meaning is reported in Tables 6–9.

In particular:

- tables *contacts* and *presence* store the information about the various contacts and on the corresponding status, respectively, and are linked together via the foreign key of table *presence* (i.e., field *contact_id*, see Table 7);

- table *avatars* stores the information about the avatars of the various contacts and, as reported in Table 8, it is linked to various tables, namely *contacts*, *providers*, and *accounts*, via the corresponding foreign keys;

- table *contactList* stores the information about how the contacts of the ChatSecure users are organized into lists, and is linked to both tables

Table 6: Structure of the *contacts* table (fields *qc* and *rejected* have been omitted because of their lack of forensic value).

Table *contacts*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique identifier of the contact
username	Secondary Key	text	username of this contact on the corresponding IM provider
nickname	–	text	name displayed by ChatSecure for this contact
provider	Foreign Key	int	value of field *_id* of the record, in table *providers*, this contact is an account of
account	Foreign Key	int	value of field *_id* of the record, in table *accounts*, corresponding to the local ChatSecure account this contact belongs to
contactList	Foreign Key	int	value of field *_id* of the record, in table *contactList*, corresponding to the contact list this contact belongs to
type	–	int	contact type: 0 (normal), 1 (temporary, not in contacts but subscribed to receive updates), 2 (temporary group chat contact), 3 (blocked), 4 (hidden), 5 (pinned)
subscriptionStatus	–	int	status update receipt from this contact: 0 (none), 1 (requested to subscribe), 2 (requested to unsubscribe)
subscriptionType	–	int	exchange of status updates with this contact: 0 (no interest in update), 1 (stop receiving updates), 2 (receive updates), 3 (contact wants updates from the user), 4 (mutual interest in receiving updates), 5 (pending invitations)
otr	–	int	status of the OTR encryption; possible values are: 0 (off), 1 (on, don't know who turned it on), 2 (on, enabled by the user), 3 (on, enabled by the contact)

Table 7: Structure of table *presence* (fields *jid_resource* and *priority* have been omitted because of their lack of forensic value).

Table *presence*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique identifier of the record
contact_id	Foreign Key	int	value of field *_id* of the record, in table *contacts* table, corresponding to the contact this presence information refers to
client_type	–	int	type of the client; possible values are: 0 (default), 1 (mobile), 2 (android)
mode	–	int	presence status of the contact; possible values are: 0 (offline), 1 (invisible), 2 (away), 3 (idle), 4 (do not disturb), and 5 (available)
status	–	text	status message of the contact

Table 8: Structure of the *avatars* table.

Table *avatars*			
Name	*Role*	*Type*	*Meaning*
_id	Primary key	int	unique avatar identifier
contact	Foreign Key	text	value of field *username* of the record, in table *contacts*, this avatar belongs to
provider	Foreign Key	int	value of field *_id* of the record, in table *providers*, corresponding to the provider this contact is user of
account	Foreign Key	int	value of field *_id* of the record, in table *accounts*, corresponding to the account the contact owning this avatar is buddy of
hash	–	text	SHA-1 hash of the picture used as avatar by this contact
data	–	blob	raw image data of the avatar

Table 9: Structure of table *contactList*.

Table *contactList*			
Name	*Role*	*Type*	*Meaning*
_id	Primary Key	int	unique identifier of the record
name	–	text	display name of this contact list
account	Foreign Key	int	value of field *_id* of the record, in table *accounts*, corresponding to the ChatSecure account this list belongs to
provider	Foreign Key	int	value of field *_id* of the record, in table *providers table*, corresponding to the provider of the ChatSecure account this contact list belongs to

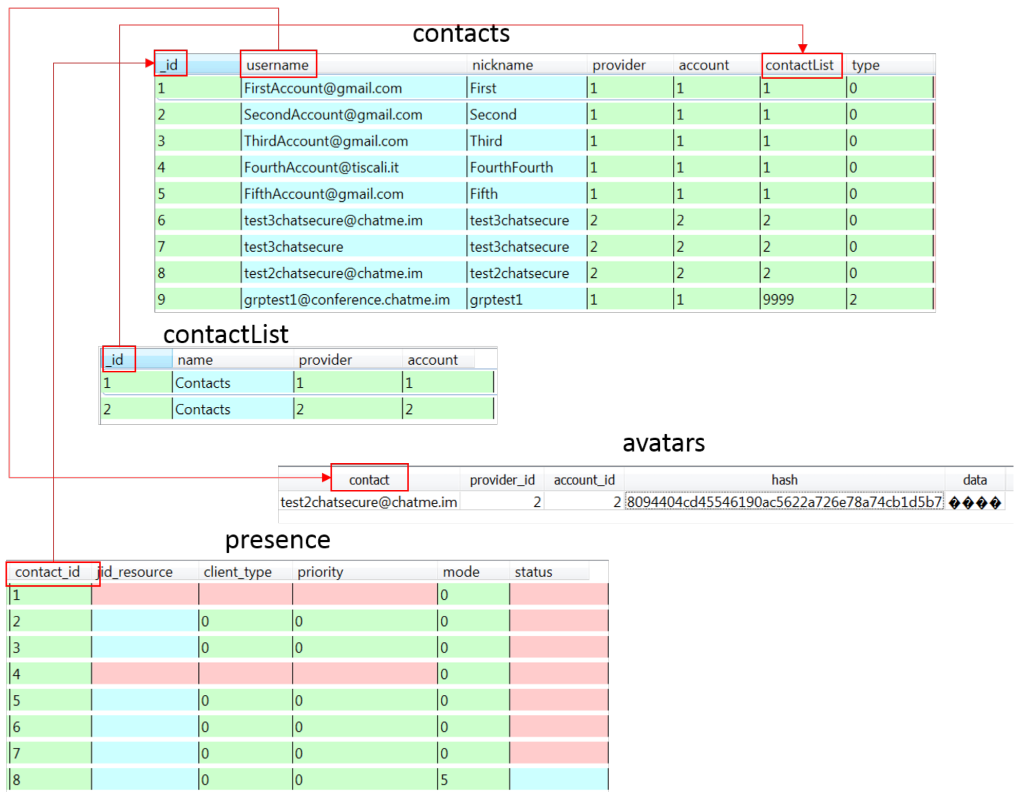
accounts and *provider* (to associate the list with the specific ChatSecure user the list belongs to) via the corresponding foreign keys (see Table 9).

To illustrate how to reconstruct the information about the contacts of a ChatSecure user, let us consider the scenario shown in Fig. 3, where 9 distinct contacts belonging to the two ChatSecure accounts shown in Fig. 2 (namely, *chat.secure.user* and *test1chatsecure*) are stored in table *contacts*.

From records no. $1, \ldots, 5$ and 9, we see that six contacts are associated with user account no. 1 (*chat.secure.user*) and provider no. 1 (*Google GTalk*), while the remaining three contacts (corresponding to records no. $6, 7$, and 8) are associated with user account no. 2 (*test1chatsecure*) and provider no. 2 (*ChatMe*). Furthermore, we also see that all these contacts are of the "normal" type (field *type*=0), with the exception of contact no. 9 (*nickname*='*grptest1*') that is a temporary contact created purposely to denote a group chat (*type*=2, see Table 6) corresponding to the remote IM account *grptest1@conference.chatme.im*.

Avatar pictures may have evidentiary value as well: they can be indeed used to link a ChatSecure contact to the real identity of the person using it (for instance, if the avatar displays the face of the user, or any location or item that can be uniquely associated with that person).

To determine the avatars associated with each contact, we join tables *contacts* and *avatars*. The results of this operation indicate that only contact no. 8 (user *test2chatsecure@chatme.im*) is associated with an avatar, whose picture is stored (as a "blob" of bytes) in field *data*; the avatar can be extracted from this field, and visualized using a standard image viewer.



_id	username	nickname	provider	account	contactList	type
1	FirstAccount@gmail.com	First	1	1	1	0
2	SecondAccount@gmail.com	Second	1	1	1	0
3	ThirdAccount@gmail.com	Third	1	1	1	0
4	FourthAccount@tiscali.it	FourthFourth	1	1	1	0
5	FifthAccount@gmail.com	Fifth	1	1	1	0
6	test3chatsecure@chatme.im	test3chatsecure	2	2	2	0
7	test3chatsecure	test3chatsecure	2	2	2	0
8	test2chatsecure@chatme.im	test2chatsecure	2	2	2	0
9	grptest1@conference.chatme.im	grptest1	1	1	9999	2

_id	name	provider	account
1	Contacts	1	1
2	Contacts	2	2

contact	provider_id	account_id	hash	data
test2chatsecure@chatme.im	2	2	8094404cd45546190ac5622a726e78a74cb1d5b7	����

contact_id	jid_resource	client_type	priority	mode	status
1				0	
2		0	0	0	
3		0	0	0	
4				0	
5		0	0	0	
6		0	0	0	
7		0	0	0	
8		0	0	5	

Figure 3: Reconstruction of ChatSecure contact lists.

Also the status information of a contact may have evidentiary value. For instance, the textual status message may provide information about the real identity of the contact, and also its presence status at the moment of the last update may provide information about the behavior of that contact.

To determine the status of each contact, as reported by the last time this information was updated locally, we have to join tables *contacts* and *presence*. From the results of this operation, we see that (a) group chat contacts (i.e., contact no. 9 in our example scenario) have no associated status, (b) the status all the other contacts is "*offline*" (the value 0 is stored in the corresponding *mode* field), with the exception of contact *test2chatsecure@chatme.im* (contact no. 8), whose status is instead "*available*" (*mode* field contains 5). We also observe that none of these contacts is associated to a textual status message (fields *status* are empty).

Finally, also contact lists may be important from the evidentiary point of view, as they allow to link each contact to the corresponding ChatSecure account used by the local user.

To reconstruct the contact lists, we have to join tables *contacts* and *contactList*. From the results of this operation, we see that these contacts are organized in two distinct lists: the first one includes contacts no. $1, \dots, 5$ and 9, and belongs to the ChatSecure account no. 1, while the second one includes the remaining contacts, and belongs to account no. 2. In both cases, the name of the list is *Contacts*.

4.5. Reconstructing the chronology and contents of chat messages

Reconstructing the time in which each message was sent or received, the content of that message, and the communication partner, is of obvious investigative importance.

Each time a message is sent or received, ChatSecure stores in the main database a record containing both its textual content and various metadata (e.g., the identifier of the corresponding buddy, and the date and time when the exchange occurred). This information is spread across two distinct tables of the main database, namely *messages* and *inMemoryMessages*, that have the same structure, [5] that is described in Table 10 together with the interpretation of its fields. The reason for which two distinct tables are used is

[5]The main database includes also another chat-related table, named *chats*, that has not been described here since the information it stores is redundant being it repeated in tables *messages* and *inMemoryMessages*.

unclear; however, the messages they contain do not overlap, so both of them needs to be analyzed to recover all the messages that have been exchanged.

Table 10: Structure of the *inMemoryMessage* and *messages* tables (fields *packet_id* and *shown_ts* have been omitted because of their lack of forensic value).

The *inMemoryMessages* and *messages* tables		
Name	*Type*	*Meaning*
_id	int	unique identifier of the message
thread_id	int	identifier of the contact this message has been exchanged with
nickname	text	used for group chat messages only to indicate the nickname chosen by the local ChatSecure user in that group chat (empty for one-to-one messages)
body	text	body of the message
date	int	the date this message has been sent or received (13-digits Unix epoch format)
type	int	type of the message: 0 (outgoing), 1 (incoming), 2 (presence became "available"), 3 (presence became "away"), 4 (presence became "busy"), 5 (presence became "unavailable"), 6 (message converted to a group chat), 7 (status message), 8 (message cannot be sent now, will be sent later), 9 (OTR is turned off), 10 (OTR is turned on), 11 (OTR turned on by the user), 12 (OTR turned on by the communicating partner), 13 (incoming encrypted), 14 (incoming encrypted and verified), 15 (outgoing encrypted), 16 (outgoing encrypted and verified)
err_code	int	error code (0 = no error)
err_msg	text	error message (if any)
is_muc	int	flag indicating whether it is a group chat message (1) or not (0)
is_delivered	int	flag indicating whether a "delivered" confirmation was received (1) or not (0)
mime_type	text	type of data exchanged (null for text message, non-null for transferred files (see Sec. 4.6)

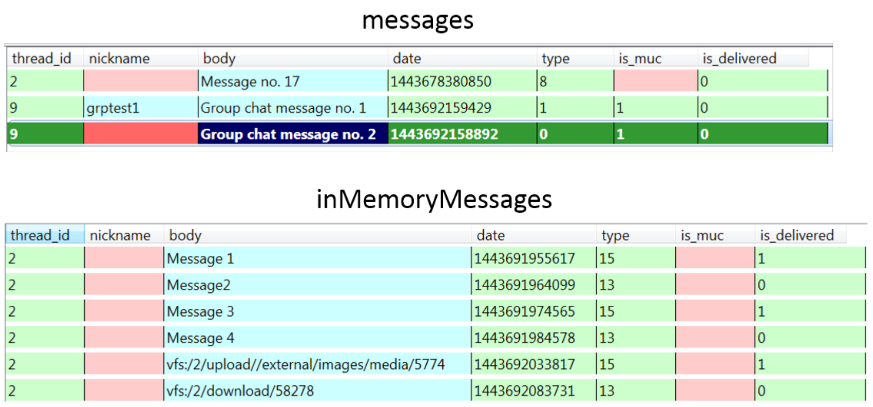
From the analysis of the meaning of the various fields (and, in particular, of the possible values of field *type*), we see that ChatSecure messages belong to two distinct categories, namely:

- *notification messages*, i.e. messages that do not carry any user-generated content, but that instead carry updates about the status contact, such as changes of his/her status (message types $2, 3, 4$ and 5), or of his/her OTR encryption status (message types $9, 10, 11$ and 12).

- *chat messages*, that carry user-generated textual content. These messages correspond to records whose *type* fields stores values in the set $\{0, 1, 13, 14, 15\}$ (see Table 10) to denote clear text outgoing (*type*=0) or incoming (*type*=1) messages, encrypted incoming message sent by an unverified (*type*=13) or a verified (*type*=14) partner, and encrypted outgoing messages sent to an unverified (*type*=15) or verified (*type*=16) partner.

The chronology of message exchanges can be reconstructed by means of the values stored in the *date* field, that store date and time of message transmission or receipt encoded as a 13-digits Unix epoch format. This holds true both for notification and chat messages, so it is possible to reconstruct not only the chronology of messages exchanged by users, but also when a notification message arrived.

To illustrate how to reconstruct the chronology of exchanged messages and the corresponding contents, let us consider the scenario depicted in Fig. 4 that shows 10 exchanged messages, which are stored in Tables *messages* (4 messages) and *inMemoryMessages* (6 messages).


messages

thread_id	nickname	body	date	type	is_muc	is_delivered
2		Message no. 17	1443678380850	8		0
9	grptest1	Group chat message no. 1	1443692159429	1	1	0
9		Group chat message no. 2	1443692158892	0	1	0

inMemoryMessages

thread_id	nickname	body	date	type	is_muc	is_delivered
2		Message 1	1443691955617	15		1
2		Message2	1443691964099	13		0
2		Message 3	1443691974565	15		1
2		Message 4	1443691984578	13		0
2		vfs:/2/upload//external/images/media/5774	1443692033817	15		1
2		vfs:/2/download/58278	1443692083731	13		0

Figure 4: Reconstruction of chronology and content of exchanged messages.

From this figure, we see that the first record in table *inMemoryMessage*, corresponds to an outgoing encrypted message (*type*=15) that was sent on Oct. 1st, 2015 at 9:32:35.617 a.m. UTC (*date*='*1443691955617*') to contact no. 2 (*thread_id*=2) (that corresponds to the contact whose nickname is *Second*, see Fig. 3); the message body was "*Message 1*", and has been successfully delivered to (i.e., visualized by) its recipient (*is_delivered*=1). The second record of *inMemoryMessages* correspond instead to an incoming encrypted message (*type*=13), that was received by the same user and carried as textual content the string "*Message 2*" on Oct. 1st, 2015 at 9:32:44.099 a.m. UTC (*date*='*1443691964099*') that has not been delivered (i.e., visualized) by the ChatSecure user.

Finally, the first message of table *messages* (whose body was "*Message no. 17*"), is an outgoing message sent to the same contact on Oct. 1st, 2015 at 05:46:20.850 a.m. UTC (*date*='*1443678380850*'), but whose transmission was delayed (*type*=8) and, as such, had not been successfully delivered to the recipient (*is_delivered*=0).

Note that ChatSecure stores in tables *messages* and *inMemoryMessages* the messages corresponding to all the local accounts, i.e. *chat.secure.user* and *test1chatsecure* in our example (see Fig. 2). However, the identity of the local account LA corresponding to a given message can be easily determined by correlating the unique contact identifier ID stored in the *thread_id* field with the record of table *contact* storing ID in its *_id* field; the value of field *account* of that record will indicate the local account LA. In this way, in the examples above we could tell that messages have been exchanged between contact no. 2 and account *chat.secure.user*, since this contact is associated with account no. 1 (see Fig. 2).

Table *messages* in Fig. 4 also stores messages exchanged in group chats, corresponding to the last two records of table *messages*, as indicated by *is_muc=1*. From these records, we see that the corresponding messages have been sent to the group chat corresponding to the contact no. *9* of table *contacts* (the chat room named *grptest@conference.chatme.im*), and carried as textual content the strings "*Group chat message no. 1*" and "*Group chat message no. 2*", respectively.

As a final observation, it is worth noticing that ChatSecure stores messages in clear text, even if they have been encrypted before transmission (see the contents of field *body* of all the records in Fig. 4): this is indeed the case of all the records stored in table *inMemoryMessages*, that correspond to encrypted messages that have been either sent (*type*=15) or received (*type*=13).

4.6. Reconstructing the chronology and contents of file exchanges

In addition to textual messages, ChatSecure allows its users to exchange also files of any type (at the moment of this writing, however, this functionality is available only for one-to-one communications and not for group chats). Determining the chronology of these exchanges, and more importantly the contents of exchanged files, may be of crucial importance in many investigations.

Each time a file is exchanged, ChatSecure creates a record that stores the same information described for chat messages, either in the *messages* or in the *inMemoryMessages* of the main database . Furthermore, it stores the content of the file into an IOCipher encrypted virtual disk to keep it inaccessible to an authorized third-party. The file transfer mechanism used by ChatSecure interfaces directly with its encrypted virtual disk, that is: (a) before being sent, files are stored on the virtual disk, from which they are fetched and sent across the network, and (b) received files are stored directly in the virtual disk.

The message records corresponding to file transfers are identified looking at the contents of their *mime_type* and *body* fields. In particular, the former field stores the MIME media type (Freed et al., 2013) of the transferred file, while the latter one stores the full path of the file in the encrypted virtual disk.

IOCipher implements the above virtual disk by using *libsqlfs* (Guardian Project, 2016b), a library that in turn implements a POSIX-style file system by means of an SQLCipher-encrypted SQLite database. This database is named *media.db*, and includes only two tables, namely *meta_data* (storing various file metadata, such as identifier, path name, and timestamps), and *value_data* (storing the actual file blocks), whose structure and interpretation is reported in Tables 11 and 12, respectively.

Table 11: Structure of table *meta_data* of the *media.db* database (fields lacking any forensic value are omitted).

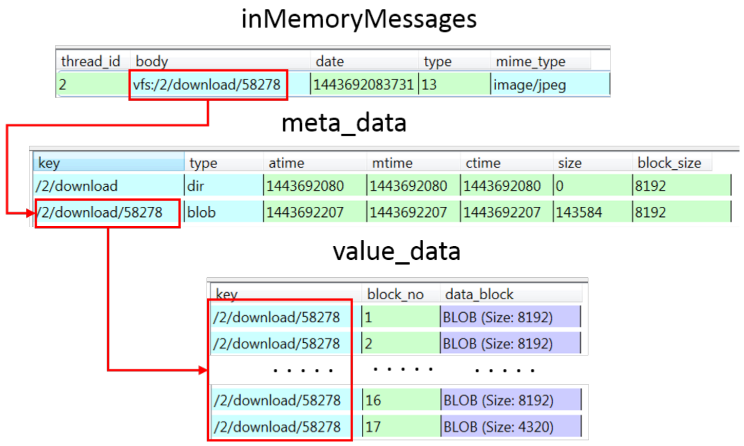
Table *meta_data*		
Name	*Type*	*Meaning*
type	text	type of the object: directory (*dir*), file (*blob*), symbolic link (*symlink*)
key	text	full path of the object in the libsqlfs file system
ctime, mtime, atime	int	file creation, last modification, and last access time, respectively (10-digits Unix epoch format)
size	int	file size (in bytes)
block_size	int	block size (in bytes)

Table 12: Structure of table *value_data* of the *media.db* database (fields lacking any forensic value are omitted).

Table *value_data*		
Name	*Type*	*Meaning*
key	text	full path of the file in the libsqlfs file system, as stored in the corresponding *meta_data* table
block_no	int	sequence number of the file block stored in this record
data_block	binary	data stored in the file block corresponding to this record

To reconstruct the chronology and the contents of the files that have been exchanged, it is necessary to analyze and correlate the records stored both in the main database *impsenc.db* and the *media.db* database implementing the encrypted virtual disk.

To illustrate how to perform this reconstruction, let us consider Fig. 5, that shows the records generated during the download of a file (for the upload case, the situation is similar).



thread_id	body	date	type	mime_type
2	vfs:/2/download/58278	1443692083731	13	image/jpeg

key	type	atime	mtime	ctime	size	block_size
/2/download	dir	1443692080	1443692080	1443692080	0	8192
/2/download/58278	blob	1443692207	1443692207	1443692207	143584	8192

key	block_no	data_block
/2/download/58278	1	BLOB (Size: 8192)
/2/download/58278	2	BLOB (Size: 8192)
. . . . .	. . . . .	. . . . .
/2/download/58278	16	BLOB (Size: 8192)
/2/download/58278	17	BLOB (Size: 4320)

Figure 5: Reconstruction of a downloaded file.

First, it is necessary to identify the records of tables *messages* and *inMemoryMessages* corresponding to file transfers by examining the values stored in the *mime_type* and *body* fields. Fig. 5 shows the record corresponding to the transfer of a JPEG image (*mime_type='image/jpeg'*) which has been stored in the encrypted virtual disk into a file whose full path name is '*/2/download/58278*' (the '*vfs:*' prefix is not part of the path name, but only indicates the use of the Android Virtual File System to interface with the libsqlfs file system). From this record, we can also determine that it was an incoming encrypted file (*type=13*), received from contact no. *2* on Oct. 1st, 2015 at 09:34:43.731 a.m. UTC (*date='1443692083731'*).

After having identified the files that have been exchanged, it is possible to retrieve the corresponding data by examining the records stored in the tables of the *media.db* database. The starting point is table *meta_data*, in which we search for a record whose *key* field stores the same path name stored in the *body* field of the corresponding *inMemoryMessages* record, i.e. '*/2/download/58278*'. From this record, we can determine that the file of interest has a size of 143584 bytes, and is stored as a sequence of blocks of 8192 bytes each.

To retrieve these blocks, all the records of table *value_data* whose *key* field stores the value '*/2/download/58278*' must be retrieved, and then the content of their *data_block* field must be extracted to be stored into a single file according to the corresponding sequence numbers.

4.7. Dealing with encryption

As mentioned before, ChatSecure relies on SQLCipher to encrypt, using the AES-256 algorithm, both the main database *impsenc.db*, as well as the *media.db* database used by IOCipher to implement the encrypted virtual disk. Therefore it is necessary to decrypt them in order to analyze their contents.

The encryption key used by SQLCipher is generated internally by ChatSecure, and is never exposed to the user. This key is instead saved in the internal memory of the device so that it can be retrieved and used by ChatSecure to decrypt the above databases. However, to make sure that an adversary cannot decrypt these databases using the saved secret key, ChatSecure uses the *CacheWord* (Guardian Project, 2015) library to encrypt it using a user-defined *secret passphrase*, and to store it into an XML file named *info.guardianproject.cacheword.prefs.xml* located in the *shared_prefs* folder (see Fig. 1). To decrypt the saved secret key, the passphrase set by the user needs to be re-entered each time ChatSecure is started.

From the above discussion it follows that to decrypt the ChatSecure databases three distinct problems must be solved, namely:

1. obtaining the *secret passphrase* chosen by the user;

2. decrypting the *secret key* stored by CacheWord;
3. decrypting the databases using the secret key.

In the rest of this section, after describing the encryption scheme adopted by ChatSecure (Sec. 4.7.1), we discuss how to decrypt the secret database encryption key (Sec. 4.7.2) and how to decrypt the databases using this key (Sec. 4.7.3). Finally, we show that the passphrase is stored in the volatile memory of the device from which it can be extracted and used in the decryption process (Sec. 4.7.4).

4.7.1. The encryption procedure

Before discussing how the secret key can be decrypted, it is necessary to illustrate the procedure used by ChatSecure to generate, encode, and store it. This procedure has been reconstructed by analyzing the source code of ChatSecure (in particular, file *WelcomeActivity.java*) and of CacheWord (in particular, files *PassphraseSecrets.java* and *PassphraseSecrectsImpl.java*), and is reported in Algorithm 1 below using pseudo-code, which is executed only when ChatSecure is used for the first time.

Algorithm 1 ChatSecure secret key generation, encryption, and storage algorithm.

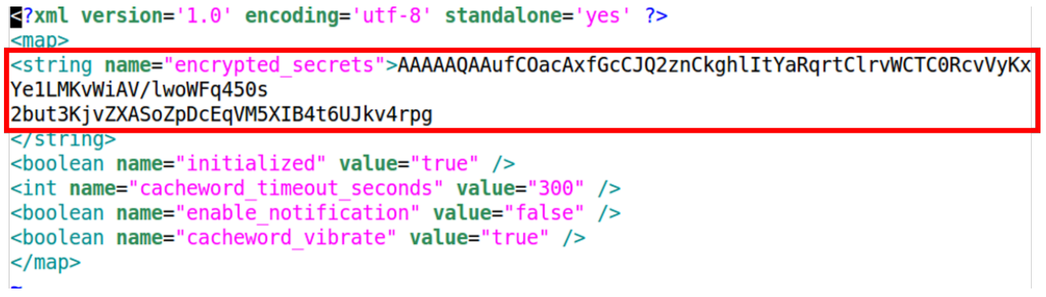
```
1: $secretKey$ = AES.generateSecretKey(256)
2: $passPhrase$ = readPassPhraseFromUser()
3: $salt$ = generateRandomSalt()
4: $IV$ = generateRandomInitializationVector()
5: $IC$ = computeIterCount()
6: $passPhraseKey$ = pbkdf2($passPhrase$, $salt$, $IV$, $IC$)
7: $encryptedSecretKey$ = AES.encrypt($secretKey$, $passPhraseKey$, $IV$)
8: $serializedSecret$ = concatenate($IC$, $salt$, $IV$, base64Encode($encryptedSecretKey$))

9: save($serializedSecret$, $info.guardianproject.cacheword.prefs.xml$)
```

As shown in Algorithm 1, a 256-bit key is generated first (line 1), and then the user is asked to provide a *passPhrase* (line 2).

Starting from this passphrase, a 256-bit *derivate key*) named *passPhraseKey* in Algorithm 1) is computed (line 6) by means of the *Password-Based Key Derivation Function 2 (PBKDF2)* (IETF Network Working Group, 2000) algorithm. This latter algorithm requires four distinct parameters, namely the passphrase and three additional values, namely a randomly-chosen 128-bit *salt* (line 3), a randomly-chosen 96-bit *initialization vector (IV)*, and a 32-bit integer *iteration counter (IC)* computed as function of the speed of the processor of the device.

Then, the derivate key *passPhraseKey* is used to encrypt the secret key used for database encryption (line 7) with AES-256, and the result is stored in the *encryptedSecretKey* variable. Finally, the values of *IC*, *salt*, and *IV* are concatenated with the Base64 encoding of *encryptedSecretKey*, and are saved (as a sequence of bytes) into the *info.guardianproject.cacheword.prefs.xml* file.

An example of the resulting *serializedSecret* is shown in Fig. 6, where it is highlighted using a square box drawn around it.



```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
<string name="encrypted_secrets">AAAAAQAAufCOacAxfGcCJQ2znCkghlItYaRqrtClrvWCTC0RcvVyKx
Ye1LMKvWiAV/lwoWFq450s
2but3KjvZXASoZpDcEqVM5XIB4t6UJkv4rpg
</string>
<boolean name="initialized" value="true" />
<int name="cacheword_timeout_seconds" value="300" />
<boolean name="enable_notification" value="false" />
<boolean name="cacheword_vibrate" value="true" />
</map>
~
```

Figure 6: The *serializedSecret* generated and saved by Algorithm 1.

4.7.2. Decrypting the SQLCipher encryption key

To decrypt the ChatSecure databases, the secret key used with SQLCipher must be known. Given that this key is unknown to the user, it must be decrypted from the *serializedSecret* stored in the *info.guardianproject.cacheword.prefs.xml* file.

Assuming that the user passphrase is known, this decryption can be carried out by means of Algorithm 2, that we devised starting from Algorithm 1.

To decrypt the secret AES key from the *serializedSecret*, first the user-generated *passphrase* is obtained in some way (either by the user or, as discussed in Sec. 4.7.4, by extracting it from the volatile memory of the device). Then, the *serializedSecret* is read from the *info.guardianproject.cacheword.prefs.xml* file (line 2), and is subsequently decomposed into its constituent elements, namely *IC*, *salt*, *IV*, and *encryptedSe-*

Algorithm 2 ChatSecure secret key decryption algorithm.

```
1: $passPhrase$ = getPassPhrase()
2: $serializedSecret$ = readFromFile($info.guardianproject.cacheword.prefs.xml$)
3: $IC$ = extractFromSequence($serializedSecret$, 0)
4: $salt$ = extractFromSequence($serializedSecret$, 32)
5: $IV$ = extractFromSequence($serializedSecret$, 160)
6: $encryptedSecretKey$ = extractFromSequence($serializedSecret$, 256)
7: $passPhraseKey$ = pbkdf2($passPhrase$, $salt$, $IV$, $iterCount$)
8: $decodedSecretKey$ = base64Decode($encryptedSecretKey$)
9: $decryptedSecretKey$ = AES.decrypt($decodedSecretKey$, $passPhraseKey$, $IV$)
```

cretKey (lines 3–6). The second parameter of function *extractFromSequence* indicates the offset (expressed in bits) from the beginning of the *serializedSecret* sequence where each element is stored (and it is computed by considering the size of each component).

To decrypt *encryptSecretKey*, the derived key *passPhraseKey* used to encrypt it (see Algorithm 1, line 7) is computed first by means of the PBKDF2 function (line 7) using the same values of *salt*, *IV*, and *IC* used to generate it in Algorithm 1 (line 7), as well as the *passPhrase*.

Then, to obtain the SQLCipher encryption key, we first Base64-decode the value stored in *encryptedSecretKey* (recall that in Algorithm 1 this key is Base64-encoded before being stored, see line 8). The result of this operation is stored in variable *decodedSecretKey* (line 8), which is finally decrypted to yield the SQLCipher key *decryptedSecretKey* (line 9).

As an example, the decryption of the *serializedSecret* shown in Fig. 6 yields the SQLCipher key '*62 9B 8D BF 3F 26 13 1B 2F B6 96 19 FD 4C F9 92 A1 D2 D0 12 96 B5 73 BA 34 59 FA FF 8A 12 CD 89*' (expressed as a sequence of bytes in hexadecimal encoding).

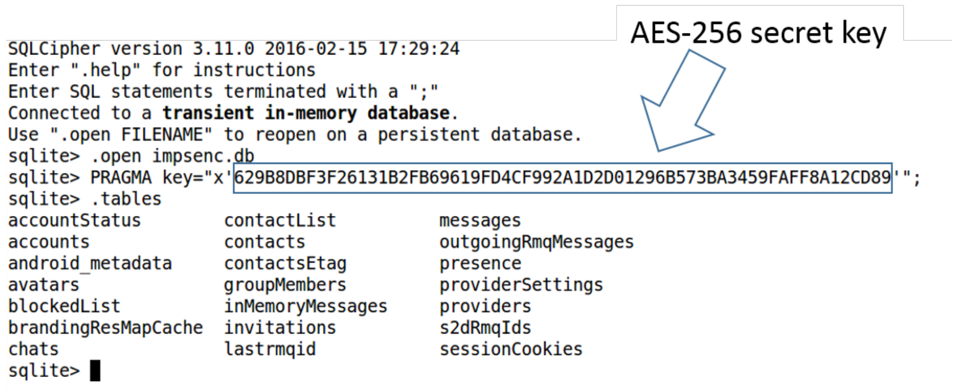
We have implemented the above decryption algorithm as an Android app that exploits parts of the CacheWord source code (in particular, file *PassphraseSecrectsImpl.java*), which is freely available upon request. The choice of implementing it for Android and not for another platform stems from the fact that the CacheWord source code does not correctly compile outside the Android development environment.

4.7.3. Decrypting ChatSecure databases

Once the encryption key used with SQLCipher has been obtained by means of Algorithm 2, the ChatSecure databases can be decrypted using any

SQLite v.3 client that supports SQLCipher.

In Fig. 7 we show how the main database *impsenc.db* can be decrypted on a Linux system by means of the SQLCipher command line tool (freely available from (Zetetic LLC., 2015)).



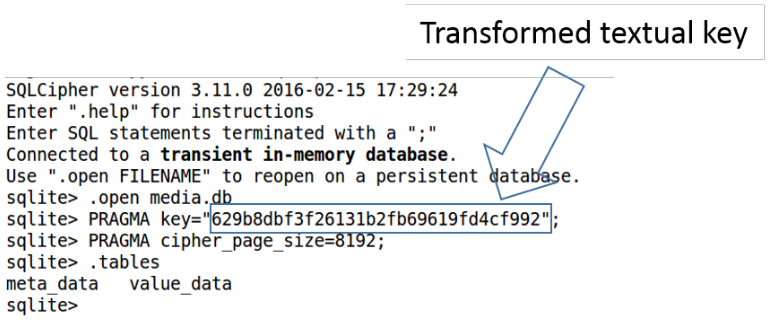
```
SQLCipher version 3.11.0 2016-02-15 17:29:24
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open impsenc.db
sqlite> PRAGMA key="x'629B8DBF3F26131B2FB69619FD4CF992A1D2D01296B573BA3459FAFF8A12CD89'";
sqlite> .tables
accountStatus        contactList          messages
accounts             contacts             outgoingRmqMessages
android_metadata     contactsEtag         presence
avatars              groupMembers         providerSettings
blockedList          inMemoryMessages     providers
brandingResMapCache  invitations          s2dRmqIds
chats                lastrmqid            sessionCookies
sqlite> █
```

Figure 7: Decrypting ChatSecure *impsenc.db* with SQLCipher.

After launching the SQLCipher client, the encrypted database is opened first by means of the *.open impsenc.db* command. Then, it is decrypted by means of the *PRAGMA key = "x'KEY_BYTES'";* command, where *KEY_BYTES* denotes the hexadecimal encoding of the sequence of bytes corresponding to encryption key. The last *.tables* command shown in Fig. 7 serves only to verify that decryption has been correctly performed, as in the case of a wrong key the PRAGMA directive fails silently.

The decryption procedure for the *media.db* database is slightly different, as shown in Fig. 8. In particular, a textual key is used in place of the 256-bit SQLCipher key used for the *impsenc.db* database (where the key was passed to the *PRAGMA key* command as a hexadecimal sequence). This textual key is obtained by first converting the 256-bit SQLCipher key into a lower-case textual string (by translating each hexadecimal digit into the corresponding ASCII character), and then by truncating it to the leftmost 32 characters. After this key has been computed, the *media.db* is decrypted by means of the *PRAGMA key="TXT_KEY"* command (where *TXT_KEY* denotes it) followed by the *PRAGMA cipher_page_size = 8192;* command [6]

[6]The value of 8192 for the page size for the encrypted database is a design choice of the libsqlfs library (see function *sqlfs_t_init* in the *sqlfs.c* file of the libsqlfs source tree).



```
SQLCipher version 3.11.0 2016-02-15 17:29:24
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
Connected to a transient in-memory database.
Use ".open FILENAME" to reopen on a persistent database.
sqlite> .open media.db
sqlite> PRAGMA key="629b8dbf3f26131b2fb69619fd4cf992";
sqlite> PRAGMA cipher_page_size=8192;
sqlite> .tables
meta_data   value_data
sqlite>
```

Figure 8: Decrypting ChatSecure *media.db* with SQLCipher

(that, instead, was not required for the *impsenc.db* database), as shown in Fig. 8.

4.7.4. Extracting the passphrase from volatile memory

As discussed before, Algorithm 2 needs the user-defined passphrase to decrypt the SQLCipher encryption key, that must be gathered in order to proceed with database decryption and analysis.

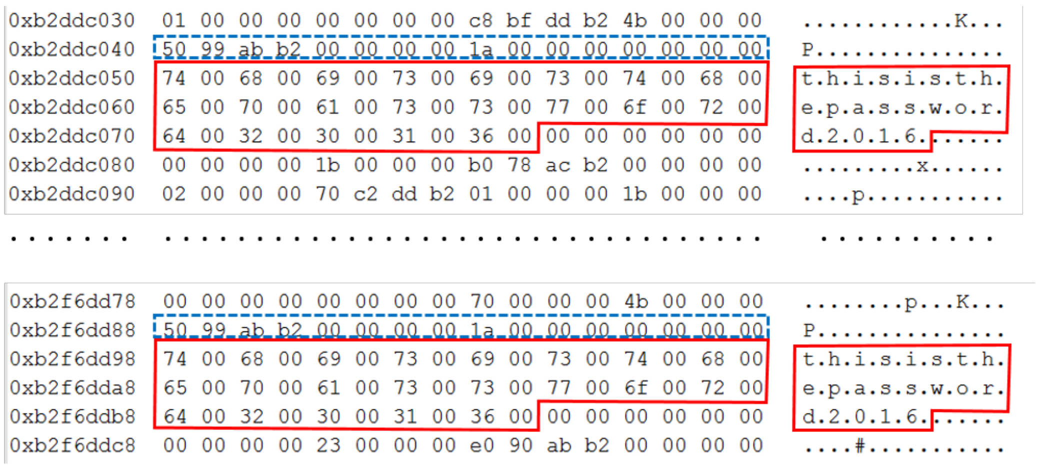
If the ChatSecure user is unwilling to reveal the passphrase, this problem becomes hard to solve, since this passphrase is never stored on the persistent memory of the device, so it cannot be retrieved from there. However, as discussed below, the passphrase persists in the volatile memory of the device after it has been inserted by the user when ChatSecure is started. Therefore, if the device is switched on and ChatSecure is running, the passphrase can be located in the volatile memory, and can be extracted from there.

In this section, we first discuss how we found out that the passphrase persists in volatile memory, and then we show how it can be identified and extracted from a dump of its contents.

To verify whether the passphrase persists in the volatile memory of the ChatSecure device, we performed experiments in which we started ChatSecure, entered the passphrase, put the application in the background, waited a given amount of time during which the app was not used; then, we extracted the contents of volatile memory of the ChatSecure process, and searched it for the passphrase that was entered. Experiments were organized in rounds, where each round included experiments in which we progressively increased the amount of time we waited before performing acquisition, up to a maximum of two hours. We ran different sets of rounds, each one corresponding to a different passphrase. Memory extraction and analysis was carried out by using the methodology described in (Anglano et al., 2016), using LiME for extraction and Volatility for analysis.

The results of our experiments can be summarized as follows:

1. the passphrase was always found in the volatile memory of the ChatSecure process, thus proving that it persists there for the entire execution of ChatSecure;
2. the passphrase is stored as a null-terminated Unicode UTF16-LE string (an example is shown in Fig. 9 for the passphrase *thisisthepassword2016*, which is highlighted by continuous-line box surrounding it);
3. the sequence of bytes encoding the passphrase is preceded by the 16-bytes signature *50 99 ab b2 00 00 00 00 1a 00 00 00 00 00 00 00* (highlighted in Fig. 9 by a dotted-line box surrounding it);
4. the passphrase and its signature appear twice in the memory space of the ChatSecure process, as exemplified in Fig. 9.



```
0xb2ddc030  01 00 00 00 00 00 00 00 c8 bf dd b2 4b 00 00 00   ............K...
0xb2ddc040  50 99 ab b2 00 00 00 00 1a 00 00 00 00 00 00 00   P...............
0xb2ddc050  74 00 68 00 69 00 73 00 69 00 73 00 74 00 68 00   t.h.i.s.i.s.t.h.
0xb2ddc060  65 00 70 00 61 00 73 00 73 00 77 00 6f 00 72 00   e.p.a.s.s.w.o.r.
0xb2ddc070  64 00 32 00 30 00 31 00 36 00 00 00 00 00 00 00   d.2.0.1.6.......
0xb2ddc080  00 00 00 00 1b 00 00 00 b0 78 ac b2 00 00 00 00   .........x......
0xb2ddc090  02 00 00 00 70 c2 dd b2 01 00 00 00 1b 00 00 00   ....p...........

.......     ................................                  ..........

0xb2f6dd78  00 00 00 00 00 00 00 00 70 00 00 00 4b 00 00 00   ........p...K...
0xb2f6dd88  50 99 ab b2 00 00 00 00 1a 00 00 00 00 00 00 00   P...............
0xb2f6dd98  74 00 68 00 69 00 73 00 69 00 73 00 74 00 68 00   t.h.i.s.i.s.t.h.
0xb2f6dda8  65 00 70 00 61 00 73 00 73 00 77 00 6f 00 72 00   e.p.a.s.s.w.o.r.
0xb2f6ddb8  64 00 32 00 30 00 31 00 36 00 00 00 00 00 00 00   d.2.0.1.6.......
0xb2f6ddc8  00 00 00 00 23 00 00 00 e0 90 ab b2 00 00 00 00   ....#...........
```

Figure 9: Passphrase in the volatile memory of the device.

Of course in real cases the passphrase is not known, so it cannot be found in memory by simply searching for it, as we instead did in our experiments. We need therefore to define a method allowing us to identify an unknown passphrase stored in volatile memory.

A natural choice would be that of using the 16-bytes signature mentioned above as a landmark indicating the position of the passphrase. Unfortunately, the results of our experiments indicate that this method yields a large number of false positives, since the above signature is present also for other UTF16-LE null-terminated strings.

However, we can leverage the fact that the passphrase appears *twice* in the volatile memory of the ChatSecure process, each time preceded by the above 16-bytes signature, to prune all the candidate strings that do not occur twice in the above memory space (they are clearly false positives). Although we cannot exclude that this procedure will filter out all the false positives, it is certainly able to greatly reduce their number.

Finally, after all the candidate passphrase have been extracted from the volatile memory region belonging to the ChatSecure process, we can find the correct one by first running Algorithm 2 for each one of them, and then by attempting to decrypt the ChatSecure databases with the secret key it returns. It is worth noticing that the database decryption procedure can be implemented by using the SQLCipher API, thus making the above method fully automatable.

4.8. Dealing with deletions

The last issue we consider is concerned with the recovery of messages and files deleted by the ChatSecure user. These deletions are performed by deleting the corresponding records from the *impsenc.db* and *media.db* databases.

It is well-known that in SQLite databases deleted records are kept in the so-called *unallocated cells*, i.e. slack space stored in the file corresponding to the database, from which they can be recovered (Jeon et al., 2012).

Unfortunately this is not the case for ChatSecure databases, since their records are deleted securely, i.e. they are overwritten upon deletion. As a matter of fact, in SQLCipher (that, as already discussed, is used both by ChatSecure and IOCipher to encrypt the *impsenc.db* and the *media.db* databases, respectively), secure deletion is enabled by default, as reported in its official documentation (sql, 2014), that states:

> "*(...) as of version 2.0.5, SQLCipher now enables SQLite's PRAGMA secure_delete=ON option. This causes the freed pages to be zeroed out on delete to hinder recovery. As before, they remain encrypted. Note that this doesn't imply that the pages are removed from the database file, just that their content is wiped when they are marked free.*"

To verify whether the above holds true in reality, we performed a set of experiments in which we deleted various messages and files from the above databases, and then we attempted to recover the corresponding records by means of specialized tools (Cellebrite LTD, 2015a; Oxygen Forensics, Inc, 2013). Our analysis did not yield any result, thus indicating that secure deletion is actually working in the current version of ChatSecure.

We have therefore to conclude that the recovery of deleted messages and files is not possible.

5. Conclusions

In this paper we have discussed the forensic analysis of ChatSecure, a secure IM application that adopts strong encryption for transmitted and locally-stored data to ensure the privacy of its users.

In particular, we have shown that ChatSecure stores local copies of both exchanged messages and files into two distinct databases, that are strongly encrypted by means of the SQLCipher library. Although the encryption mechanisms used by ChatSecure is rather complex, we have devised an algorithm able to decrypt these databases starting from the secret passphrase chosen by the user as the initial step of the encryption process.

We have also shown how this passphrase can be identified and extracted from the volatile memory of the device, where it persists – after having been entered by the user – for the entire execution of ChatSecure, thus allowing one to carry out decryption even if the passphrase is not revealed by the user.

Moreover, we have also shown how to analyze and correlate the data stored in the databases used by ChatSecure to identify the IM accounts used by the user and his/her buddies to communicate, as well as to reconstruct the chronology and contents of the messages and files that have been exchanged among them.

Finally, we have shown that the data stored in the databases cannot be recovered after having been deleted, as a consequence of the secure deletion technique adopted by SQLCipher.

The study reported in this paper has been performed by means of a methodology that is based on the use of emulated devices and therefore provides a very high degree of reproducibility of the results. The accuracy of the method has been assessed by validating the results it yields against those obtained from real smartphones. We believe that this methodology represents also a significant contribution of this paper.

References

Forensic Recovery of Deleted Data. 2014. Available at https://discuss.zetetic.net/t/forensic-recovery-of-deleted-data/20.

504ENSICS Labs . Linux memory extractor (lime). 2016. Available at http://codeload.github.com/504ensicsLabs/LiME/zip/master.

Al Barghuthi N, Said H. Social Networks IM Forensics: Encryption Analysis. Journal of Communications 2013;8(11):708–15.

Anglano C. Forensic Analysis of WhatsApp Messenger of Android Smartphones. Digital Investigation 2014;11(3):201–13. doi:10.1016/j.diin.2014.04.003.

Anglano C, Canonico M, Guazzone M. Technical Note to Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones. Technical Report TR-INF-2016-09-02-UNIPMN; University of Piemonte Orientale; 2016. Available at https://www.di.unipmn.it/TechnicalReports/TR-INF-2016-09-02-UNIPMN.pdf.

Borisov N, Goldberg I, Brewer E. Off-the-Record Communication, or, Why Not To Use PGP. In: Proc. of the 2004 ACM Workshop on Privacy in the Electronic Society (WPES). Washington, DC, USA: ACM Press; 2004. p. 77–84. doi:10.1145/1029179.1029200.

Cellebrite LTD . UFED Mobile Forensics Applications. 2015a. Available at http://www.cellebrite.com/Mobile-Forensics/Applications.

Cellebrite LTD . UFED4PC: The Software-Based Mobile Forensics Solution. 2015b. Available at http://www.cellebrite.com/Mobile-Forensics/Products/ufed-4pc.

Cellebrite LTD . Cellebrite Android Forensics. 2016. Available at http://www.cellebrite.com/mobile-forensics/capabilities/android-forensics.

Electronic Frontier Foundation . Secure Messaging Scorecard. Available at https://www.eff.org/secure-messaging-scorecard.

Epifani M, Stirparo P. Learning iOS Forensics. Packt Publishing, 2015.

Freed N, Klesin J, Hanses T. Media Type Specifications and Registration Procedures. 2013.

Gliph Inc . Gliph: Secure Group Messaging and Bitcoin Payments. 2015. Available at https://gli.ph/.

Google . Android Debug Bridge. 2016a. Available at https://developer.android.com/studio/command-line/adb.html.

Google . Android Device Monitor. 2016b. Available at https://developer.android.com/studio/profile/monitor.html.

Google . Run Apps on the Android Emulator. 2016c. Available at https://developer.android.com/studio/run/emulator.html.

Guardian Project . CacheWord: Passphrase Caching and Management. 2015. Available at https://guardianproject.info/code/cacheword/.

Guardian Project . ChatSecure: Encypted Messenger for iOS and Android. 2015a. Available at https://chatsecure.org/.

Guardian Project . ChatSecure source code. 2015b. Available at https://chatsecure.org/developers/.

Guardian Project . IOCipher: Virtual Encrypted Disks. 2016a. Available at https://guardianproject.info/code/iocipher/.

Guardian Project . libsqlfs. 2016b. Available at https://github.com/guardianproject/libsqlfs.

Guardian Project . Orbot: TOR for Android. 2016c. Available at https://guardianproject.info/apps/orbot/.

Husain MI, Sridhar R. iForensics: Forensic Analysis of Instant Messaging on Smart Phones. In: Goel S, editor. Digital Forensics and Cyber Crime. Springer Berlin Heidelberg; volume 31 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*; 2010. doi:10.1007/978-3-642-11534-9_2.

IETF Network Working Group . PCKS #5: Password-Based Cryptography Specification Version 2.0. RFC n. 2898; 2000. Available at https://tools.ietf.org/html/rfc2898.

Jeon S, Bang J, Byun K, Lee S. A recovery method of deleted records for SQLite database. Personal and Ubiquotous Computing 2012;16(6):707–15. doi:10.1007/s00779-011-0428-7.

MSAB . XRY. 2016. Available at http://www.msab.com/xry/xry-current-version.

Open Whispers System . TextSecure/Signal. 2015. Available at https://whispersystems.org/.

Oxygen Forensics, Inc . SQLite Viewer. 2013. Available at http://www.oxygen-forensic.com/en/features/analyst/data-viewers/sqlite-viewer.

Oxygen Forensics, Inc . Oxygen Forensics Detective. 2016. Available at http://www.oxygen-forensic.com/en/products/oxygen-forensic-detective.

Quarkslab SAS . ChatSecure security assessment. Technical Report 14-03-022; 2015. Available at http://blog.quarkslab.com/resources/2015-06-25_chatsecure/14-03-022_ChatSecure-sec-assessment.pdf.

SQLite Consortium . SQLite Home Page. 2013. Available at http://www.sqlite.org.

Tamma R, Tindall D. Learning Android Forensics. Packt Publishing, 2015.

Telegram Inc . Telegram: a New Era of Messaging. 2015. Available at https://telegram.org/.

TOR Project . The TOR Project: Anonymity Online. 2016. Available at https://www.torproject.org/.

Tso YC, Wang SJ, Huang CT, Wang WJ. iPhone Social Networking for Evidence Investigations Using iTunes Forensics. In: Proc. of the 6th International Conference on Ubiquitous Information Management and Communication. New York, NY, USA: ACM; ICUIMC '12; 2012. p. 1–7. doi:10.1145/2184751.2184827.

United Nations Office on Drugs and Crime . Comprehensive Study on Cybercrime. Technical Report; United Nations; 2013.

Volatility Foundation . An advanced memory forensics framework. 2016. Available at http://volatilityfoundation.org/.

Walnycky D, Baggili I, A.Marrington , Moore J, Breitinger F. Network and device forensic analysis of Android social-messaging applications. Digital Investigation 2015;14, Supplement 1:S77–84. doi:10.1016/j.diin.2015.05.009; proc. of 15th Annual DFRWS Conference.

Wickr Inc . Wickr: the Most Trusted Messenger in the World. 2015. Available at https://www.wickr.com/.

XMPP Standards Foundation . XMPP Technologies Overview. 2015. Available at http://xmpp.org/about-xmpp/technology-overview.

Zetetic LLC. . SQLCipher. 2015. Available at https://www.zetetic.net/sqlcipher.