Forensic Analysis of Telegram Messenger on Android Smartphones

Please cite this paper as:
Cosimo Anglano, Massimo Canonico, Marco Guazzone,
"Forensic Analysis of Telegram Messenger on Android Smartphones,"
Digital Investigation, Volume 23, December 2017, Pages 31–49.
DOI:10.1016/j.diin.2017.09.002
Publisher: https://doi.org/10.1016/j.diin.2017.09.002

Forensic Analysis of Telegram Messenger on Android Smartphones

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

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

Abstract

In this paper we present a methodology for the forensic analysis of the artifacts generated on Android smartphones by *Telegram Messenger*, the official client for the Telegram instant messaging platform, which provides various forms of secure individual and group communication, by means of which both textual and non-textual messages can be exchanged among users, as well as voice calls.

Our methodology is based on the design of a set of experiments suitable to elicit the generation of artifacts and their retention on the device storage, and on the use of virtualized smartphones to ensure the generality of the results and the full repeatability of the experiments, so that our findings can be reproduced and validated by a third-party.

In this paper we show that, by using the proposed methodology, we are able (a) to identify all the artifacts generated by Telegram Messenger, (b) to decode and interpret each one of them, and (c) to correlate them in order to infer various types of information that cannot be obtained by considering each one of them in isolation.

As a result, in this paper we show how to reconstruct the list of contacts, the chronology and contents of the messages that have been exchanged by users, as well as the contents of files that have been sent or received. Furthermore, we show how to determine significant properties of the various chats,

*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)

groups, and channels in which the user has been involved (e.g., the identifier of the creator, the date of creation, the date of joining, etc.). Finally, we show how to reconstruct the log of the voice calls made or received by the user.

Although in this paper we focus on Telegram Messenger, our methodology can be applied to the forensic analysis of any application running on the Android platform.

Keywords: Mobile forensics, Telegram Messenger, Telegram, Android, Instant Messaging

---

1. Introduction

Instant Messaging (IM) platforms are nowadays very popular among smartphone users, because they provide very convenient ways to share both textual and non-textual contents. In addition to legitimate users, IM services are very popular also among criminals (United Nations Office on Drugs and Crime, 2013), that use them for their communications as they make it harder than traditional communication means to link the real identity of a person to an account (s)he uses, and more and more often they use end-to-end encryption to escape interception. The forensic analysis of these applications is therefore of crucial importance from the investigative standpoint (Wu et al., 2017; Zhang et al., 2016; Zhou et al., 2015; Anglano, 2014; Anglano et al., 2016; Mehrotra and Mehtre, 2013; Walnycky et al., 2015).

This is particularly true for *Telegram*, a very popular IM platform (in Feb. 2016, the Telegram Messenger LLP company reported that there were 100,000,000 active users per month, with 350,000 new users signing up per day (Telegram Messenger LLP, 2016)), providing secure one-to-one, one-to-many, and many-to-many communication services, as well as self-destructing chats, that is reportedly used for various criminal activities, ranging from cybercrime (C. Budd, 2016) to those engaged by various terrorist organizations (J. Warrick, 2016). The ability of accessing the contents of communications carried out by means of Telegram may thus assume crucial importance in many investigations.

While it is already known that *Telegram Messenger* (its official client) saves on the internal memory of the device a significant amount of unencrypted data (Gregorio et al., 2017; Satrya et al., 2016a,b), to the best of our knowledge there is no published work, addressing the forensic analysis

of Telegram Messenger on Android, that provides a methodology to obtain – from the above data – the complete reconstruction of all the user activities and, at the same time, to allow an independent party to validate these results.

Furthermore, while it is true that most prominent mobile forensic platforms (e.g., (Cellebrite LTD., 2015b; Micro Systemation, 2016; Oxygen Forensics, Inc., 2013a; Compelson Labs, 2017)) are able to decode the various data stored by Telegram Messenger, they do not provide any explanation of how this decoding is performed, nor they provide any guidance on how to correlate different pieces of evidence to completely reconstruct user activities. Thus, it is impossible to assess the completeness and the correctness of the results generated by them.

In this paper we fill this gap by presenting a methodology for the forensic analysis of applications running on Android, and we apply it to the Telegram Messenger (the focus on the Android version of Telegram Messenger maximizes the investigative impact of our work, as 85% of Telegram users use an Android smartphone (The Telegram Team, 2017)). We show that, thanks to the use of this methodology, we are able to fully reconstruct all the user activities by (a) identifying all the artifacts that carry relevant investigative information, (b) describing how they can be decoded in order to extract that information, and (c) showing how they can be correlated in order to infer information of potential investigative interest that cannot be obtained by considering individual artifacts in isolation.

Our methodology is based on the exploitation of virtualized smartphones in place of physical ones. The use of virtualized devices brings various benefits, the most important of which are the generality and the reproducibility of the results, while we establish their accuracy by comparing them with those obtained by using a physical smartphone.

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

- we present a forensic analysis methodology for applications running on Android, which is based on the use of virtualized smartphones and provides a very high degree of reproducibility of the results;

- we use this methodology to perform a thorough and reproducible analysis of Telegram Messenger, and we validate the results we obtained against those obtained from real smartphones. In particular:

– we identify all the forensically-relevant artifacts stored by Telegram Messenger on Android smartphones;
– we determine the structure and format of these artifacts, and we implement the corresponding decoding procedures in a Java program, that we use for our analysis;
– we map the data stored by Telegram Messenger to the user actions that generated it;
– using the above mapping, we show how to recover the account used with Telegram Messenger, and how to reconstruct (a) the contact list of the user, (b) the chronology and contents of both textual and non-textual messages, and (c) the log of the voice calls done or received by the user.

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 Telegram Messenger, performed by applying the above methodology 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 Telegram Messenger during its usage.

The importance of the forensic analysis of IM applications on Android smartphones has been also acknowledged in the literature.

(Wu et al., 2017; Zhang et al., 2016; Zhou et al., 2015) focus on *WeChat*, (Anglano, 2014) on *WhatsApp*, (Anglano et al., 2016) on *ChatSecure*, and (Mehrotra and Mehtre, 2013) on *Wickr*.

(Walnycky et al., 2015) discusses the analysis of the data transmitted or stored locally by 20 popular Android IM applications, while (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.

(Azfar et al., 2016) proposes instead a taxonomy outlining the artifacts of forensic interests generated by various communication apps. Other papers (Ovens and Morison, 2016; Husain and Sridhar, 2010; Tso et al., 2012) have instead focused on the forensic analysis of IM applications on iOS devices.

None of the above papers, however, focuses on Telegram Messenger, which is instead the focus of (Gregorio et al., 2017), where a methodology for its forensic analysis on the Windows Phone platform is presented. Our work differs from this one in two important regards: (a) our methodology is more general (indeed, the methodology discussed in (Gregorio et al., 2017) can be considered a sub-case of ours), and (b) the structure and interpretation of the artifacts generated by Telegram Messenger on Android are significantly different from those generated on Windows Phone devices.

The analysis of Telegram Messenger on Android has been also partly addressed in (Satrya et al., 2016a,b) that, however, focus on the identification of the location and format of the artifacts generated by Telegram Messenger, but not on their interpretation and correlation. In particular, in the above papers only the raw data generated by Telegram Messenger are shown (e.g., the raw contents of various tables in the main database of the app), but these data are not interpreted and are not tied to specific actions performed by the user.

Analogous considerations hold for (Susanka, 2017) that, although focusing on vulnerabilities of Telegram Messenger on Android, describes also how to decode the data stored in two of the tables of the main database (that, however, contains may other tables storing forensically-relevant information).

In contrast, in our work we provide a much deeper analysis of the artifacts generated by Telegram Messenger, in which we show how to analyze them to reconstruct the various actions carried out by users, that include contact management, textual and non-textual message exchanges, and voice communications.

3. The analysis methodology

The methodology we propose for the forensic analysis of Android applications is based on the controlled execution of a set of experiments, using one or more Android devices, and on the inspection and analysis of the internal memory (both persistent and volatile) of these devices.

Given that the goal of any forensic analysis is to allow the analyst to obtain the digital evidence generated by the application under consideration, the methodology used to carry out it must exhibit the following properties:

- *completeness*: the identification of all the data generated by the application under analysis. To obtain completeness, suitable experiments stressing all the relevant functionalities of the application need to be carried out;
- *repeatability*: the possibility for a third-party to replicate the experiments under the same operational conditions, and to obtain the same results. To achieve repeatability, it must be possible for a third-party to use the same set of devices, operating systems versions, and forensic acquisition tools to repeat experiments under the same operational conditions;
- *generality*: the results hold for many (possibly all) Android smartphones and versions. To achieve generality, the experiments should be repeated on as many smartphones and Android versions as possible.

In our methodology, we achieve completeness by designing suitable experiments, by executing them in a systematic way, and by resorting to source code analysis (when possible) to gather additional insights into the behavior of the application and/or in the way it encodes the data it stores locally. To achieve generality and repeatability, we resort to virtualized mobile devices instead of physical ones (more precisely, we use the *Android Mobile Device Emulator* (Google, 2016b), see Sec. 4.5.1 for more details). Virtualized smartphones, indeed, make simple and cost-effective running experiments on a variety of different virtual devices (featuring different hardware and software combinations), thus yielding generality. Furthermore, they allow a third-party to use virtualized devices identical to those we used in our experiments, as well as to control their operational conditions, so that the same conditions holding at the moment of our experiments can be replicated on them. In this way, repeatability is ensured.

More precisely, our methodology is organized into a workflow, depicted in Fig. 1, that encompasses a sequence of distinct phases. The methodology starts with two distinct and independent activities, that can be carried out simultaneously.

The first one ("*Analysis of application functionalities*") consists in the analysis of the functionalities provided by the application, and is aimed at

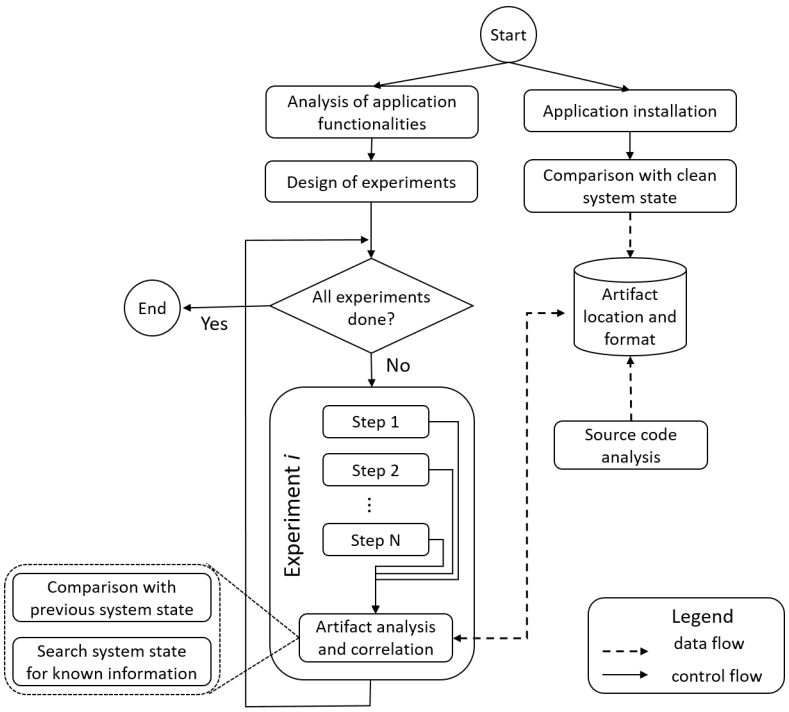

Figure 1: Workflow of the analysis methodology for mobile applications.

the identification of those functionalities whose use may generate information that are relevant from an investigative standpoint. These functionalities are provided as input to the subsequent phase ("*Design of experiments*"), where we define a set of experiments that emulate a user who exercises them, so as to elicit the application to generate and store data on the memory of the device.

The second one consists instead in locating – on the device memory – where the application stores the data it generates during the installation process, with the aim of analyze them to determine their type, their encoding, and the information they represent. To this end, the application is first installed ("*Application installation*"), and then the data stored in the internal memory of the smartphone during the installation process are identified. This is achieved by extracting the file systems, located on the internal memory of the device, before and after the installation of the application, and by finding the differences among them ("*Comparison with clean system state*"). Such extraction can be performed by either dumping the contents of the file systems (virtualized devices grant root access), or by directly accessing the files typically used by virtualization platforms to implement persistent memory.

As shown in Fig. 1, "*Source code analysis*" may be employed (provided that the source code of the application is available) to assist the analyst in determining the format of the data stored in these files, as well as the procedure to decode and interpret them. Source code analysis is also used in later stages of the methodology to better understand how the application works, as well as to corroborate or refute hypotheses about its behavior.

Once the initial stages of the analysis have been completed, the methodology contemplates the carrying out of the experiments, that are performed one after the other until the last experiment has been completed.

Each experiment may consist of several consecutive steps, and at the end of each one, the contents of the device internal memory (both persistent and volatile) are extracted and compared with those extracted at the end of the previous step ("*Comparison with previous system state*"), in order to determine whether new data have been generated during the step, and where these data are located. Furthermore, both allocated and undeleted files, as well as unallocated space, may be searched for known information generated by the application, such as specific values or patterns that are defined in the experiment ("*Search system state for known information*"). If requested by the analysis needs, such a search may be also performed on

volatile memory, that may be collected and analyzed by means of suitable tools (e.g., (504ENSICS Labs., 2016; Volatility Foundation, 2016)). Source code analysis may be also employed to assist in the determination of the format and meaning of these data.

After the data generated in the last step have been identified, they are added to the list of known artifacts ("*Artifact location and format*"), and are analyzed and correlated among them (if needed) to map them to the user actions carried out in that step. Finally, at the end of each experiments, the artifacts created in its steps may be correlated to carry out more complete reconstructions of user actions.

4. Forensic analysis of Telegram Messenger

In this section, we apply the methodology discussed in Sec. 3 to the forensic analysis of *Telegram Messenger*, the official client of Telegram that has been released directly by Telegram Messenger LLP (the company that owns Telegram).[1] While various third-party Telegram clients are available for the Android platform (e.g., *Plus Messenger*, *Anyways*, *Supergram*, *Telegram+*, *Telegram Plus*, etc.), in this paper we focus on Telegram Messenger because, at the moment of this writing, it is indeed the most used Telegram client (it features more that 100 million installations, unlike the other ones that cumulatively feature no more than a few million downloads). Furthermore, all these third-party clients use the same code base of Telegram Messenger, so the results discussed in this paper apply also to them.

We start in Sec. 4.1 with the analysis of application functionalities, whose results provide the basis for the design of the experiments we carry out in our study, that are discussed in Sec. 4.2. Then, in Sec. 4.3 we describe the location and format of the artifacts generated by Telegram Messenger, as resulting from the experiments we carried out (although this is the outcome of the whole experimental process, we anticipate them to provide a better understanding of the material that follows). Next, in Sec. 4.4 we describe how we exploited source code analysis to gain a better understanding of how Telegram Messenger encodes most of the relevant information it stores on the smartphone. Finally, in Sec. 4.5 we describe the results of our analysis,

[1] At the moment of this writing, Telegram clients exist for both mobile (i.e., *Android, iOS, Windows Phone*, and *Firefox OS*) and desktop systems (i.e., *Windows, Linux*, and *Mac OS*), as well as for web-based access (Telegram Messengers LLP, 2017).

obtained through the execution of the various experiments. In particular, we discuss (a) how to determine which account has been used on the mobile device to access Telegram (Sec. 4.5.2), (b) how to reconstruct the list of contacts (Sec. 4.5.3), (c) how to reconstruct the chronology and contents of exchanged messages (Sec. 4.5.4), as well as of the voice calls log (Sec. 4.5.5) and, finally, we report our (negative) findings concerning the issues of the recovery of deleted data (Sec. 4.5.6).

4.1. Analysis of application functionalities

As discussed in Sec. 3, the first step of the methodology consists in analyzing the functionalities of the application, in order to identify those that are relevant from the investigative standpoint.

Telegram is an IM platform that enables its users to exchange messages, using various communication schemes (namely, one-to-one, one-to-many, and many-to-many communications), as well as to carry out voice calls, using various privacy-preserving techniques. Telegram supports the exchange of both textual messages (whose content is plain text) and non-textual ones (used to exchange data of any type, including contact information, geo-coordinates, and files of any type).

Telegram Messenger is a feature-rich application that provides users with access to all the functionalities provided by the Telegram platform. Of course, not all these functionalities are of investigative interest, so we analyze them and we identify those that bring valuable information in various investigative scenarios, namely:

- *The Telegram user account*: within the Telegram system, each user is uniquely identified by means of its *Telegram ID* (*TID*), an integer value (chosen by Telegram) which is associated with a phone number. Furthermore, each user may optionally set a *Telegram username*, that allows other users to find him/her via the search functionality provided by Telegram, and a *profile photo*.

  All these information may have a significant investigative value. The TID uniquely identifies the user within the Telegram system, while the phone number may be instead used to tie the virtual identity of the user to its real one. The profile photo may instead provide additional hints about the identity or the location of the user, for instance if it displays the face of the user, or any location or item that can be uniquely associated with that person.

- *Contacts*: in Telegram, each user is associated with a list of *contacts*, i.e., other Telegram users with whom (s)he may communicate. For each contact, Telegram Messenger stores his/her TID, phone number, and profile photo.

  The investigative importance of the information about contacts is clear, as it allows an investigator to determine whom the user was in contact with, and to possibly determine the real identity of each contact by using – as already discussed for the user account information – his/her TID, phone number, and profile photo.

- *Message exchanges*: Telegram provides its users with the possibility of carrying out one-to-one, one-to-many, and many-to-many communication by using, respectively, *chats*, *channels*, and *groups*, where users can exchange both textual and non-textual messages. Telegram Messenger organizes the messages exchanged by the user into *dialogs*, each one corresponding to a specific chat, group or channel.

  The ability of reconstructing the chronology and contents of exchanged messages is of obvious investigative importance, as it allows the investigator to determine with whom the user communicated, when these communications occurred, and what it was exchanged.

  Furthermore, the identification of the properties of each dialog in which the user was involved with (i.e., its type, its creator, its date of creation, its administrators, etc.) may provide valuable evidence in various investigative scenarios. For instance, the choice of using a secret chat (a form of chat where messages are encrypted end-to-end and they self-destroy after a user-defined amount of time) instead of a regular one may indicate the intention of the users to totally hide the fact they are communicating. Analogously, the creation and administration of a private group or channel (i.e., that cannot be found by the search function of the Telegram platform) on which illegal material is shared, or unlawful communications are broadcast, may provide evidence that the user was involved in criminal activities (e.g., terrorist propaganda or diffusion of child pornography material).

- *Voice calls*: starting with v. 3.18, Telegram provides its users with voice calls that, as secret chats, relies on user-to-user communication channels and end-to-end encryption. The ability of reconstructing the

chronology of voice calls (i.e., when a call has been performed, with whom, and for how long) is of evident investigative value.

4.2. Design of experiments

As discussed in Sec. 3, after the relevant functionalities have been identified, the next step of the methodology consists in the design of a set of experiments aimed at reconstructing the actions that a user may carry out by exploiting them.

On the basis of the functionalities of Telegram Messenger, identified in the previous section, we define the following sets of experiments:

1. *experiments concerning the user account*, listed in Table 1, whose goal is to determine the Telegram account that has been used on the smartphone;

Table 1: Experiments concerning the Telegram account of the local user. A and B denote the Telegram users involved in the experiments.

User account experiments	
Description	**Steps**
Configuration of the user account	1. A connects to Telegram providing its credentials 2. B connects to Telegram providing its credentials

2. *experiments concerning contacts*, listed in Table 2, whose goal is to reconstruct the contact list of the user, as well as the management operations carried out on it by the user;
3. *experiments concerning dialogs*, listed in Table 3, whose goals are the reconstruction of the dialogs in which the user has been involved with and the determination, for each one of them, of (a) its type (regular/secret chat, private/public (super)group or channel), (b) the role of the user (creator/administrator/member), and (c) its creation date (for dialogs created by the user) or the date in which the user joined it;
4. *experiments concerning message exchanges*, listed in Table 4, whose goal is the reconstruction of the chronology and contents of the textual and non-textual messages exchanged within each dialog;
5. *experiments concerning phone calls*, listed in Table 5, whose goal is the reconstruction of the chronology of the phone calls made and received by the user.

Table 2: Experiments concerning user contacts. A and B denote the Telegram users involved in the experiments.

User contacts experiments	
Description	**Steps**
Invite a contact	1. A invites B to join Telegram
Add/remove contacts	1. A adds B from within Telegram Messenger 2. A deletes B from within Telegram Messenger
Add/remove contacts	1. A adds B in the phonebook 2. A removes B from the phonebook
Add/remove contacts	1. A adds B from within Telegram Messenger 2. A removes B from the phonebook
Add/remove contacts	1. A adds B in the phonebook 2. A removes B from within Telegram Messenger
Block a contact	1. A blocks B 2. A unblocks B

Table 3: Experiments concerning dialogs. A and B denote the Telegram users involved in the experiments.

Dialogs experiments	
Description	**Steps**
Regular chat	1. A creates a regular chat with B by sending a message to him/her 2. A and B exchange messages (as reported in Table 4) 3. A deletes the chat
Secret chat	1. A creates a secret chat and adds B 2. B opens the message notifying he joined the secret chat 3. A and B exchange messages (as indicated in Table 4) 4. A deletes the chat
Public (private) group	1. A creates a new public (private) group 2. A adds B to the group 3. A and B exchange textual and non-textual messages over the group (as reported in Table 4) 4. A deletes the group
Public supergroup	1. A joins the public supergroup named "*Telegram Party*" 2. A sends several messages to the supergroup 3. A receives several replies from the supergroup 4. A leaves the supergroup
Public (private) channel	1. A creates a new public (private) channel 2. A adds B to the channel 3. A and B exchange textual and non-textual messages over the channel (as reported in Table 4) 4. A deletes the channel

Table 4: Experiments involving message exchanges. A, B, and C denote the Telegram users involved in the experiments.

One-to-one message exchange	
Description	**Steps**
Textual message sending, both users are online	1. A and B exchange several messages 2. A and B delete the messages one after the other
Textual message sending, receiver is offline	1. A sends a message to B when B is offline 2. B goes online and replies to A
Textual message sending, sender is offline	1. A goes offline 2. A sends a message to B 3. A goes online
Textual message sending, receiver is blocked	1. A blocks B 2. A sends a message to B 3. A unblocks B
Textual message forwarding	1. A forwards to C the message sent to B 2. B forwards to C the message received from A
Non-textual message sending	1. A sends a picture file to B 2. A sends a video file to B 3. A sends an audio file to B 4. A sends a PDF file to B 5. A sends a contact to B 6. A sends geo-location information to B 7. A and B delete the messages one after the other

Table 5: Experiments concerning phone calls. A and B denote the Telegram users involved in the experiments.

Phone calls experiments	
Description	**Steps**
Successful call	1. A calls B 2. B answers 3. A and B have a conversation 4. A hangs up
Unanswered call	1. A calls B 2. B does not answer 3. A rings B until the call is dropped by Telegram
Refused call	1. A calls B 2. B refuses the call

6. *deleted information*, whose goal is to determine whether it is possible to recover deleted information. The experiments concerning deletion are spread across Tables 2, 3, and 4.

4.3. Location and format of Telegram artifacts

In this section we anticipate the results we collected at the end of the various experiments, concerning the location and format of the artifacts generated by Telegram Messenger that, as discussed in Sec. 3, is one of the main output of the proposed methodology. While this output is the outcome of the whole experimental process, and not of the initial stages of the methodology, we believe that anticipating it here provides a better understanding of the interpretation of the data generated during these experiments, and that will be discussed in detail in the subsequent sections.

During its use, Telegram Messenger stores various data in the internal memory of the device, and in particular in the folders of the *user partition* shown in Fig. 2. These folders are located in the /data directory (which is the usual mount point of the user partition in the Android file system), and in particular in folders data (which is inaccessible to standard users) and media (whose access is instead unrestricted).

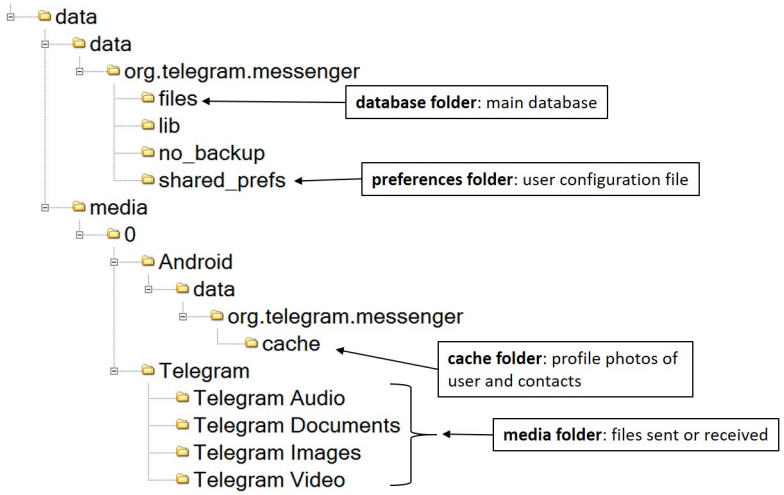

Figure 2: Structure of the folders where Telegram stores its artifacts.

As reported in Fig. 2, the results of our analysis show that there are four main artifacts that provide forensically-relevant data, namely:

- the *main database*, an SQLite database named cache4.db which is located in the *database folder*, and that contains many important information, namely the list of contacts of the user, the chronology of exchanged messages, the contents of textual messages, the location on the device memory where the contents of non-textual messages are stored, and the log of the voice calls;

- the *user configuration file*, an XML file named *userconfing.xml* [2] located in the *preferences folder*, that stores the details of the Telegram account used on the device;

- the *profile photos* of the user and of his/her contacts, stored in the *cache folder*, that may reveal information about the real identity of user contacts;

- the copies of the files sent or received by the user via Telegram Messenger, which are stored in the *media folder*.

As will be detailed in Sec. 4.5, Telegram Messenger stores most of the data it generates into complex data structures (Telegram Messenger LLP, 2017c), that we name *Telegram Data Structures* (*TDSs*). TDSs are stored in a *binary serialized* form (Telegram Messenger LLP, 2017a), whereby their fields are stored as a sequence of bytes where each one of them appears in a specific position. Therefore, to retrieve the information stored in a TDS, it is first necessary to *deserialize* it, i.e., to extract its various fields from the corresponding binary sequence, and then to decode each one of them.

To carry out the above steps, the structure and the serialization scheme of the TDSs must be known. However, at the moment of this writing, the serialization procedure is known only for a limited set of TDSs. Source code analysis, which represents the subsequent step of our methodology, has been used to identify all the different TDSs used by Telegram Messenger, as well as to reconstruct the structure and serialization scheme for the undocumented ones, or for those that have been changed since they have been documented.

4.4. Source code analysis

As discussed in Sec. 3, source code analysis should be – whenever possible – used to gain a deeper understanding of the behavior of the application under

[2] Please note that the name is misspelled by Telegram.

analysis, as well as to determine how it encodes and stores its relevant data, so that suitable decoding and interpretation procedures can be devised.

In the Telegram Messenger this stage can be performed, thanks to the availability of its source code (which can be downloaded from (DrKLO, 2017)). The main outcome of this stage has been the understanding of the structure of the TDSs used by Telegram Messenger and the consequent development of a procedure to deserialize them from the binary format in which they are stored, that we subsequently implemented as a Java program that we used to decode the results gathered in our experiments.

The structure of these TDSs is specified in the so-called *Telegram Language* (*TL*) (Telegram Messenger LLP, 2017b), that defines them as *composite types*, i.e., data structures that are made up by a set of *attributes*, each one consisting in turn either in a *base types* (which are primitive data types, like int and string) or in a composite type. Each composite type is uniquely identified by its *constructor*, which is represented by a 32-bit integer number.

The official Telegram documentation (Telegram Messenger LLP, 2017a) provides the rules to deserialize base types. For instance, the int type can be deserialized by reading from its serialized form a sequence of 4 bytes in little-endian order and interpreting their concatenation as a 32-bit signed two's complement integer. However, for composite types the above documentation is, at the time of this writing, outdated and incomplete.

The analysis of source code allowed us to determine that the deserialization of a composite type consists in (a) determining the type to be deserialized, and (b) recursively deserializing its attributes. The first step is performed by reading from the binary serialized form of the composite type a 32-bit header representing the type's constructor. Since each constructor uniquely identifies a type, once the constructor is read the logical structure of the remaining binary sequence is known. Thus, the second step simply applies the same deserialization procedure to each type's attribute (except for attributes of a base type, which do not have a constructor).

For the sake of readability, we do not to provide here the full details of the deserialization procedure for all the TDSs used by Telegram Messenger. However, this procedure can be easily deduced from the inspection of the source code where these TDSs are defined. In particular, each TDS is defined in a specific Java class, as reported in Table 6, that belongs to the `org.telegram.tgnet` package (which is located under the TMessagesProj/src/main/java/org/telegram/tgnet of the source code).

To deserialize a given TDS, we note that the corresponding Java class

Table 6: Mappings between TDSs and Telegram Messenger's Java classes for deserialization.

TDS	Java class
Base types (e.g., int and string)	`org.telegram.tgnet.NativeByteBuffer`
Chat	`org.telegram.tgnet.TLRPC.Chat`
documentAttributeAudio	`org.telegram.tgnet.TLRPC.TL_documentAttributeAudio`
documentAttributeFileName	`org.telegram.tgnet.TLRPC.TL_documentAttributeFilename`
documentAttributeVideo	`org.telegram.tgnet.TLRPC.TL_documentAttributeVideo`
EncryptedChat	`org.telegram.tgnet.TLRPC.EncryptedChat`
FileLocation	`org.telegram.tgnet.TLRPC.FileLocation`
GeoPoint	`org.telegram.tgnet.TLRPC.GeoPoint`
Message	`org.telegram.tgnet.TLRPC.Message`
messageActionPhoneCall	`org.telegram.tgnet.TLRPC.TL_messageActionPhoneCall`
MessageMedia	`org.telegram.tgnet.TLRPC.MessageMedia`
messageMediaContact	`org.telegram.tgnet.TLRPC.TL_messageMediaContact`
messageMediaDocument	`org.telegram.tgnet.TLRPC.TL_messageMediaDocument`
messageMediaGeo	`org.telegram.tgnet.TLRPC.TL_messageMediaGeo`
messageMediaPhoto	`org.telegram.tgnet.TLRPC.TL_messageMediaPhoto`
messageMediaVenue	`org.telegram.tgnet.TLRPC.TL_messageMediaVenue`
Peer	`org.telegram.tgnet.TLRPC.Peer`
PhotoSize	`org.telegram.tgnet.TLRPC.PhotoSize`
User	`org.telegram.tgnet.TLRPC.User`
UserProfilePhoto	`org.telegram.tgnet.TLRPC.UserProfilePhoto`

contains a `TLdeserialize()` method, that is able to deserialize the composite type corresponding to that TDS, as well as its subtypes. This method reads the type's constructor and, according to the value read, calls the right `readParams()` method to deserialize the right subtype.

4.5. Analysis results

As illustrated in Sec. 3, the core of our methodology is the carrying out of the experiments designed in the previous stage of the methodology. In the Telegram Messenger case, these experiments have been described in Sec. 4.2.

In this section we illustrate how to decode, interpret, and correlate the data generated by these experiments in order to reconstruct the actions performed by the user during his/her interactions with Telegram Messenger.

We start with the description of the various settings holding for the experiments (Sec. 4.5.1), and how we collected the data generating during their execution. Then, we show how to identify the Telegram account used on the smartphone under examination (Sec. 4.5.2) and, subsequently, how to reconstruct the contact list and the operations the user did on it (Sec. 4.5.3). Next, we show how to reconstruct the chronology and contents of textual and non-textual messages exchanged over the various dialogs, as well as of the properties of these dialogs (Sec. 4.5.4), and then we discuss how to recon-

struct the voice call logs (Sec. 4.5.5). Finally, we report our negative findings about the possibility of recovering information that have been deleted by the user (Sec. 4.5.6).

4.5.1. Experimental settings

In our experiments we used various releases of Telegram Messenger, that include those ranging from v. 3.15 to v. 3.18. As a matter of fact, the application is updated quite frequently, and in the time frame spanning our study, we observed several updates. Hence, we repeated the experiments with all these versions, as they were released. In this way, we could observe that the results of the experiments were the same regardless of the specific version of Telegram Messenger used to carry out them since, despite the updates, the location, format, amount, and interpretation of the data they store remained unchanged.

Furthermore, in our study, we use – as the mobile virtualization platform – the *Android Mobile Device Emulator* (Google, 2016b), that allows one to create virtual smartphones (named *Android Virtual Devices*, or AVD for brevity) behaving exactly like real physical devices, and that can be customized with different hardware characteristics and Android versions. In particular, in this study we use the three AVDs configurations shown in Table 7 below, that are characterized by different Android versions, processor families, and volatile and persistent storage sizes. To extract the file systems

Table 7: 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), 6.0 (API 23)
Intel Atom (x86_64)	1536	1024	6.0 (API 23), 7.1 (API 25)

located in the internal memory of the AVD, we use the *pull* functionality of the *File Explorer*, a component of the *Android Device Monitor* (Google, 2016a). Alternatively, it is possible to directly analyze the files used by the emulator to implement the various partitions of the internal memory (each partition corresponds to an image file in *QCOW2* format (Marc McLoughlin, 2008)).

The interested reader may refer to (Anglano et al., 2017) for the complete

description on how to concretely configure and use the various tools to create, run, and analyze an AVD.

To validate the results obtained with AVDs, we compare a sample of them, generated in a subset of selected experiments, with those obtained by running the same experiments on a real smartphone. More precisely, we run a subset of the experiments also 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 tests we performed, the results collected from this smartphone were identical to those obtained from the virtualized smartphones we considered.

4.5.2. Identification of the Telegram account

To determine the Telegram account used on the smartphone under examination, we carry out the experiments reported in Table 1, whose results show that this information is stored in the user configuration file, and in particular into one of its attributes, which is named user. This attribute stores a TDS of type User, whose structure is reported in Table 8, that in turn uses other TDSs of type UserProfilePhoto and FileLocation (also shown in the same table). The structure of these TDSs is reported in the official documentation. [3]

As can be seen from the above table, the information concerning the TID (field id), first name (field first_name), last name (field last_name), Telegram username (field username), and phone number (field phone) of the user are stored as simple values (of suitable type) into the first five fields of the User TDS. [4]

The information concerning the user profile photo is instead stored in field photo, where it is encoded into a TDS of type UserProfilePhoto, that in turn contains the identifier of the photo (field photo_id), as well as the information concerning the names of two JPEG files (Telegram Messenger converts all profile photos in the JPEG format) storing a smaller version (field photo_small) and a larger version (field photo_big) of that photo.

These files, which are both stored in the cache folder (see Fig. 2) are named as *V_L.jpg*, where V and L denote the values stored in fields volume_id

[3] See https://core.telegram.org/type/User, https://core.telegram.org/type/UserProfilePhoto, and https://core.telegram.org/type/FileLocation.

[4] The name assignment rule is described in https://core.telegram.org/constructor/userContact.

Table 8: Structure of the various TDSs used to encode the user account information.

Structure of TDS User		
Field	**Type**	**Meaning**
id	int	User TID
first_name	string	first name the user assigned to himself
last_name	string	last name the user assigned to himself
username	string	Telegram username
phone	string	phone number
photo	TDS UserProfilePhoto	profile photo
self	bool	flag indicating whether the contact refers to the user itself (*True*) or not (*False*)
contact	bool	flag indicating whether the contact is in the contact list of the user (*True*) or not (*False*)
mutual_contact	bool	flag indicating whether the user is in the contact list of the contact (*True*) or not (*False*)
Structure of TDS UserProfilePhoto		
Field	**Type**	**Meaning**
photo_id	long	identifier of the photo
photo_small	TDS FileLocation	location of the file on the local storage, corresponding to the small profile photo thumbnail
photo_big	TDS FileLocation	location of the file on the local storage, corresponding to the big profile photo thumbnail
Structure of TDS FileLocation		
Field	**Type**	**Meaning**
dc_id	int	number of the data center holding the file
volume_id	long	identifier of the server volume
local_id	int	identifier of the file

and local_id of the corresponding TDS FileLocation.

An example is reported in Fig. 3, that shows the serialized User TDS stored in the user field of the user configuration file, and its fields after it has been deserialized and decoded as reported in Table 8.

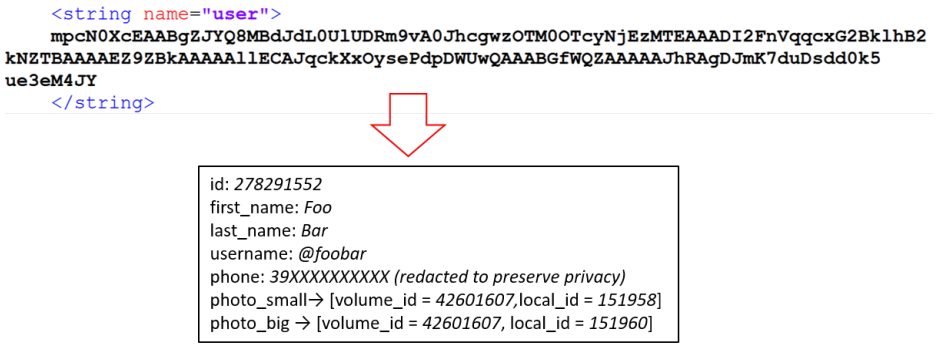

Figure 3: Decoded user information extracted from the user configuration file.

As can be seen from this figure, the TID of the user is *278291552*, while its Telegram username is '*@foobar*'. This user has set his first and last name to *Foo* and *Bar*, respectively, and has also set a profile photo. In particular, the smaller-size profile photo is stored in file *42601607_151958.jpg*, while the larger-size one is stored in file *42601607_151960.jpg*.

4.5.3. Reconstruction of operations on contacts

To reconstruct both the contact list and the operations carried out on it by the user, we perform the experiments reported in Table 2. From the analysis of the results we collected, we devise methods (a) to reconstruct the contact list starting from the data stored in the main database (Sec. 4.5.3.1), (b) to identify those contacts that have been added to the list *explicitly* by the user, and which ones have been instead automatically imported from the phonebook of the device by Telegram Messenger (Sec. 4.5.3.2), and (c) to identify *blocked* contacts, i.e., contacts that are no longer allowed to send messages to the user (Sec. 4.5.3.3).

4.5.3.1. Reconstruction of the contact list.

Telegram Messenger splits the information concerning each contact across two tables of the main database, namely contacts and users, whose structure is described in Table 9. In particular, the former table stores the TIDs of the actual contacts, while the latter one stores the information about all the

Telegram users that are known to the local user (e.g., those that have posted a message in a common group) even if they do not belong to the contact list.

Table 9: Structure of tables contacts and users.

Table contacts		
Field	**Type**	**Meaning**
uid	int	TID of the contact (connects the record with the corresponding one in table users)
mutual	int	indicates whether the local user is also in the contact list of the contact (1) or not (0)
Table users		
Field	**Type**	**Meaning**
uid	int	TID of the user
name	string	name of the user as given by the local user (i.e., it is not the Telegram username chosen by that user)
status	int	date and time, encoded as a Unix epoch timestamp, of the last status change of the user
data	TDS User (see Table 8)	information about the user

As shown in Table 9, each record in table contacts stores the TID of the corresponding contact (field uid), and a boolean value (field mutual) indicating whether the local user is also in the contact list of that contact. The corresponding record in table users is linked via its uid field, that also stores the same TID value. This latter record stores also the name given by the local user to the contact (field name), the time and date of its last status change (field status), and all the contact details (field data), encoded into a TDS of type User, whose structure is described in Table 8.

The contact list of the user can be reconstructed by selecting the records in table users that have a corresponding record in table contacts, as done by the following SQL query:

```
SELECT * FROM users WHERE uid IN (SELECT uid FROM contacts)
```

An example is reported in Fig. 4, that shows how a record in table contacts is linked to the corresponding record in table users, together with other two records of the latter table that do not correspond to a contact.

In particular, the only contact in this contact list corresponds to the user with TID=*278291552*, whose detailed information is shown in the box connected to the corresponding record of table users. The other two records in table users correspond instead to Telegram users that, albeit "known" to the local user, do not belong to its contact list. In particular, the first one

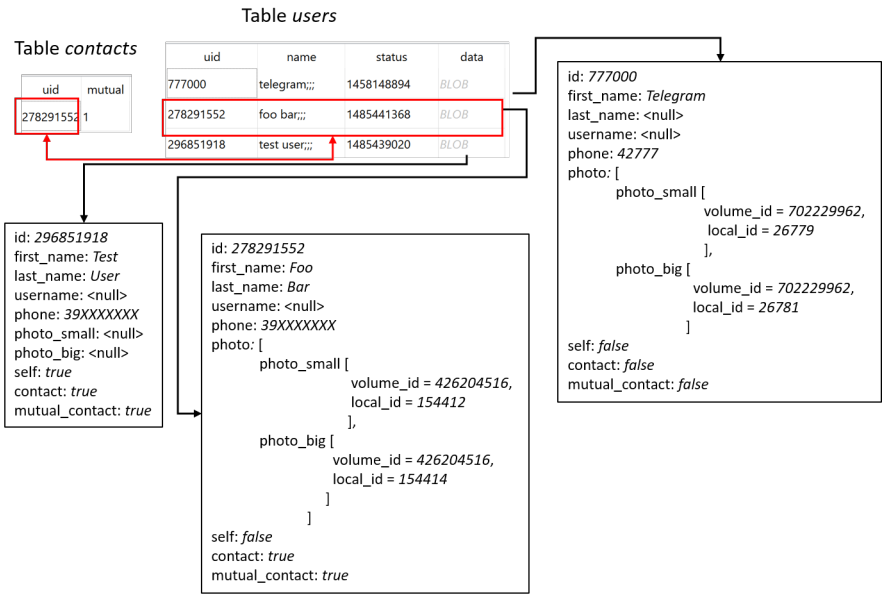

Figure 4: Reconstruction of a contact list.

(TID=*777000*) corresponds to the fictitious user which is used by the Telegram platform to broadcast service messages on a public channel, while the second one (TID=*296851918*) corresponds to the local user (field self=*true*).

4.5.3.2. Identifying contacts added explicitly.

In addition to the explicit insertion or deletion of a contact (where these operations are carried out by the user), Telegram Messenger is able to synchronize the phonebook of the device with the contact list. In particular, it automatically adds to the contact list any Telegram user whose phone number is stored in the phonebook of the device. Furthermore, it automatically removes from the contact list any Telegram user whose phone number is removed from the phonebook.

Being able to tell which contacts have been added deliberately by the user may be important in some investigative scenarios. The results of our analysis, however, show that Telegram Messenger does not store locally any information allowing one to distinguish between the explicitly-added and automatically-added contacts. Nevertheless, we find that explicitly-added contacts can be identified by analyzing the database corresponding to the phonebook of the device, which is implemented as an SQLite database,

named contacts2.db and stored in folder com.android.providers.contacts (Tamma and Tindall, 2015).

In particular, when a contact is added (either automatically or explicitly) to the contact list, Telegram Messenger writes a record also in the raw_contacts table of the phonebook database. Hence, contacts imported from the phonebook will correspond to *two* distinct records in this table: one already present in the phonebook, and another one added by Telegram Messenger. Conversely, contacts added by the user were not already present in the phonebook, and therefore will correspond to only one record in table raw_contacts.

An example of a contact that has been imported by Telegram Messenger is reported in Fig. 5, that shows the two records of table raw_contacts corresponding to the same Telegram user. In this figure, we see that table

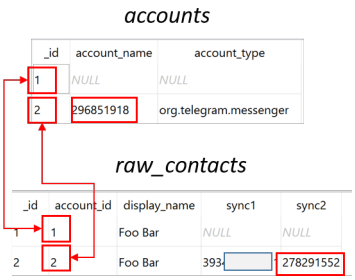

Figure 5: Records created in table raw_contacts when the contact is automatically imported by Telegram Messenger (phone numbers have been obscured to preserve privacy).

accounts (also belonging to the phonebook database) stores a record corresponding to the local Telegram user: it is the record that in fields account_name and account_type store the TID of the local user and the string '*org.telegram.messenger*', respectively. This record is linked, as shown in the figure, with the record added by Telegram Messenger to table raw_contacts, that stores the phone number and TID of the contact in fields sync1 and sync2, respectively.

4.5.3.3. Identifying blocked contacts.

Telegram Messenger allows a user to *block* a contact, which from that moment on cannot send messages to him/her anymore, and cannot receive his/her status updates. When the user blocks a contact, Telegram Messenger adds a record to table blocked_users of the main database, which contains only one field (named uid), that stores the TID of the blocked contacts.

Therefore, the set of blocked users may be reconstructed by selecting those records in table contacts whose TIDs are stored also in table blocked_users, as done by the following SQL query:

```
SELECT * FROM contacts WHERE uid IN (SELECT uid FROM
    blocked_users)
```

The results of our analysis show also that when a contact is unblocked (after having been blocked), the corresponding record in table blocked_users is deleted. Therefore, it is not possible to tell whether a currently unblocked contact has been blocked in the past, or how many times a currently blocked contact has been blocked and unblocked in the past.

4.5.4. Reconstruction of message exchanges

To reconstruct the communication activities carried out by the user, we perform the experiments reported in Tables 3 and 4. In this section we describe the methods we devised to reconstruct these activities. After a brief discussion of the main characteristics of the various dialogs types provided by Telegram Messenger (Sec. 4.5.4.1), we show how to identify the dialogs stored in the main database (Sec. 4.5.4.2), how to determine their properties (Sec. 4.5.4.3), and how to reconstruct the chronology and contents of the messages exchanged in each one of them (Sec. 4.5.4.4).

4.5.4.1. Dialog types.

As already mentioned, Telegram Messenger provides various types of dialogs enabling its users to exchange both textual (i.e., that carry UTF-8-encoded text) and non-textual (i.e., used to exchange files of any type, as well as contact information and geo-location information) messages, namely:

- One-to-one *regular chats* (also known as *cloud chats* in the Telegram jargon): messages are relayed by the Telegram system, where a copy of them is kept so that they can be synchronized on all the devices the user utilizes to access Telegram. Server-client encryption is used to encrypt messages when in transit between the sender and the Telegram system, and then from the latter one to the final recipient. For each pair of users, Telegram allows only a single regular chat.

- One-to-one *secret chats*: messages are exchanged directly between the devices of the users, i.e., their messages are never sent to nor stored on the Telegram system. End-to-end message encryption is used to

encrypt messages before transmission. Unlike regular chats, the same pair of users may create and use simultaneously as many secret chats as they desire. A secret chat is created by one of the users, and is automatically joined by the other one, after the creation has been carried out, the first time that (s)he connects to Telegram.

- One-to-many *channels*: a channel broadcasts, to all its subscribed users, the messages published by its creator or by one of its administrators (no one else is allowed to send messages on the channel). As for regular chats, messages are relayed by the Telegram system, where a copy of them is kept, and server-client encryption is used. A channel may be either *public* or *private*: in the former case, their *Telegram username* is published on the Telegram system and anyone can join them, while in the latter case they do not have a username and only invited users may join.

- Many-to-many *groups* and *supergroups*: a group broadcasts to all its members the messages sent by anyone of them. Telegram provides two distinct types of groups: *standard groups*, that can have up to 200 members, and *supergroups*, that can have from 201 to 5,000 members. As for channels, a (super)group may be either public or private.

Telegram Messenger organizes the messages exchanged by the user into *dialogs*, each one corresponding to a specific chat, (super)group or channel. At any given time, the user may be involved with as many dialogs as (s)he wants. The information concerning the various dialogs the user is involved with, and the messages exchanged within them, is stored into various tables of the main database.

4.5.4.2. Identification of dialogs.
Telegram Messenger stores into table dialogs (see Table 10) the information concerning the dialogs the user has been involved with. Each record stores the *Dialog Identifier* (*DID*) in the field did, an integer number that uniquely identifies the corresponding dialog, and other information concerning that dialog, namely the time and date of the last operation carried out in the dialog (field date), the number of messages that still have to be read (field unread_count), whether the dialog has been *pinned* (i.e., it always appears on top of the dialog list), and other less relevant information. To identify the

Table 10: Structure of table dialogs.

Table dialogs		
Field	**Type**	**Meaning**
did	int	Dialog ID (DID) of the dialog
date	int	time and date of the last operation carried out in the conversation (message sending, reception, deletion, etc.), expressed in Unix epoch time format
unread_count	int	number of messages that have been received, but still have to be read
inbox_max	int	highest Message ID (MID) among received messages
outbox_max	int	highest MID among sent messages
last_mid	int	MID of the last message that has been sent or received in the dialog (max between inbox_max and outbox_max)
pinned	int	flag indicating whether the dialog has been *pinned*, i.e., locked in the app so that it always appears on top of the dialogs list (1), or not (0)

messages exchanged within a given dialog, we anticipate that Telegram Messenger stores into one of its tables (i.e., messages) a record for each message that has been sent or received, and that this record stores – into one of its fields named uid – the DID of the dialog it belongs to (the complete discussion of table messages, and how the data it stores are used in the forensic analysis, is postponed to Sec. 4.5.4.4).

Therefore, the messages exchanged in a dialog whose DID is x can be identified by selecting those records in table messages whose uid field contains the value x, as done by the following SQL statement:

```
SELECT * FROM messages WHERE uid=x
```

An example is reported in Fig. 6, that shows two dialogs, corresponding to DIDs *77000* and *278291552*, respectively, and the related messages.

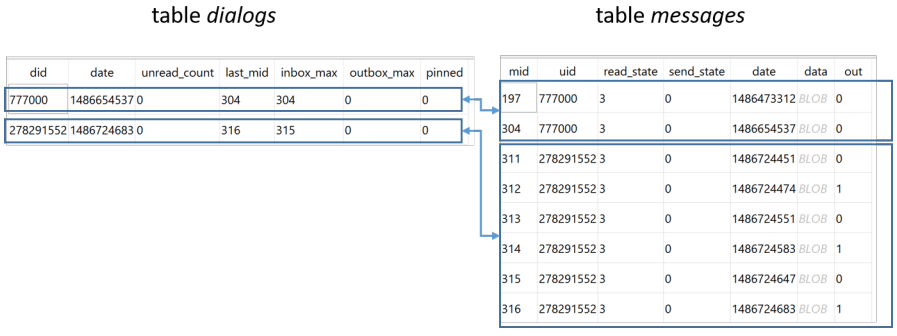

Figure 6: Contents of the dialogs table, and association with the corresponding records in the messages table. The fields of table messages are described in Table 13.

4.5.4.3. Determination of dialog properties.
As mentioned before, each dialog is characterized by various properties, that include its type (i.e., the specific communication mechanism it uses, namely regular or secret chat, public or private (super)group or channel), the role of the user (creator, administrator, or simple member), the user who created it, and the date of creation or of joining.

Telegram encodes the type of a dialog into its DID value, which is chosen according to specific rules, while its properties are stored in additional tables of the main database, as discussed below.

Identification of regular chats
In Telegram, each regular chat is identified by using as DID of the corresponding dialog the TID of the correspondent user of that chat. For this reason, as mentioned before, only a single regular chat is allowed between any pair of users.

Hence, regular chats can be identified by selecting the records of table dialogs whose field did contains one of the values stored in field uid of records in table users, i.e., any Telegram user the local user has exchanged messages with, as done by the following SQL statement:

```
SELECT * FROM dialogs WHERE did IN (SELECT uid FROM users)
```

For instance, both the dialogs shown in Fig. 6 correspond to regular chats between the local user and users identified by TIDs 777000 and 278291552, respectively.

A regular chat is automatically created when the first message is exchanged between the involved users. Hence, the creator of the chat is the user who sent the first message, and the date of creation coincides with that of the sending of this message (see Sec. 4.5.4.4).

Identification of secret chats
For each secret chat, Telegram Messenger stores a record in table enc_chats (see Table 11). Hence, these chats can be identified by looking at the records in the above table.

As can be seen from Table 11, table enc_chats stores various information concerning each secret chat, including its identifier (field uid), its name (field name), and the TID of the chat partner (field user).

Furthermore, other relevant information are encoded in the TDS stored in field data, whose structure is also shown in Table 11, and in particular: (a) the TID of the user who created the chat (field admin_id), (b) the creation date (field date), and (c) the TID of the user who joined the chat after creation (field participant_id).

To identify the messages exchanged in a given secret chat, it is necessary to identify the corresponding record in table dialogs, so that its did value can be used to retrieve the records of table messages that are linked to it. The results of our analysis show that, for these records, we have that dialogs.did = enc_chats.uid $\ll 32$, where $\ll$ denotes the left-shift bitwise operator. Hence, dialogs corresponding to secret chats can be identified accordingly, as done by the following SQL statement:

```
SELECT * FROM dialogs WHERE did IN (SELECT uid<<32 FROM
    enc_chats)
```

As an example, consider the scenario shown in Fig. 7, where we show a record in table enc_chats and the corresponding record in table dialogs (note that $1952104764 \ll 32 = 8384226119745798144$). By decoding the fields of

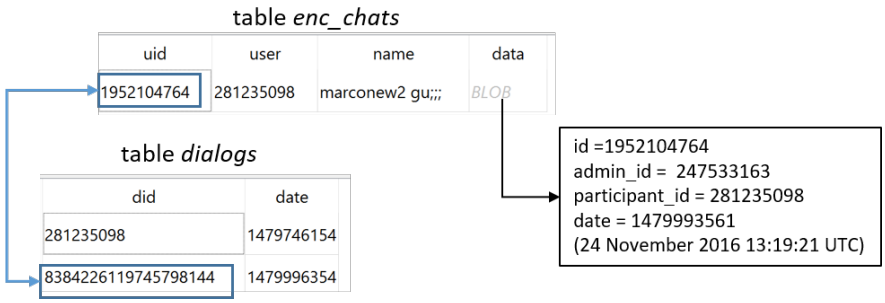

Figure 7: Determining the properties of a secret chat and the corresponding dialog.

the enc_chats record as reported in Table 11, we see that (a) the secret chat has been created by the user whose TID is *247533163*, (b) it has been created on Nov 24, 2016, and (c) it has been joined later by the user whose TID is *281235098*.

Identification of groups, supergroups, and channels

For each group, supergroup, or channel, Telegram Messenger stores a record in table chats (see Table 12), in addition to a record in table dialogs.

The records of these tables that correspond to the same dialog are linked together via the values stored in fields uid and did, respectively. In particular, we have that dialogs.did = -chats.uid. Therefore, the dialogs corresponding to a (super)group or channel can be identified accordingly, as done by the following SQL statement:

```
SELECT * FROM dialogs WHERE did IN (SELECT -uid FROM chats)
```

As can be seen from Table 12, table chats stores the identifier of the (super)group chat (field uid) and its name (field name). Additional information is instead encoded in the TDS of type Chat stored in field data (also reported in Table 12).

The data stored in this TDS allow one to determine many properties of the dialog. In particular, the type of the dialog can be determined by correlating the values stored in fields version and megagroup as follows:

- if version=1, then the dialog is a group;
- if version=0 and megagroup=*True*, then the dialog is a supergroup;
- if version=0 and megagroup=*False*, then the dialog is a channel.

Its public or private nature can be instead determined by the value stored in field username, which is set to its Telegram username for public dialogs, and to the *null* value for private ones.

Dialogs created or administered by the user can be identified by means of the value stored in fields creator and administrator, respectively, while their date of creation is stored in field date. This latter field contains instead the date of joining, for those dialogs that were not created by the user. The number of members of the dialog is instead stored in field participants_count, while its optional title and profile photo are stored in fields title and photo, respectively. Finally, active dialogs (i.e., dialogs the user is still participating in) are those where left=*False*.

To illustrate, let us consider the scenario shown in Fig. 8, representing an excerpt of table chats that contains four records, each one corresponding to a different (super)group/channel, and in particular:

- Record no. 1 corresponds to a private (username = *null*) group (version=1), which has been created by the local user (creator=*True*) on April 9, 2017 at 11:42:15 UTC (date=1491738135), and whose title is '*GroupTest*'. The user is still member of the group (left=*False*), that has two participants (participants_count = 2).

Table 11: Structure of table enc_chats and of the TDS EncryptedChat it contains.

Structure of table enc_chats		
Field	**Type**	**Meaning**
uid	int	identifier of the secret chat
user	int	TID of the secret chat partner
name	string	name given to the secret chat by the user that created it
data	TDS EncryptedChat	stores additional information about the secret chat
Structure of TDS EncryptedChat		
Field	**Type**	**Meaning**
id	int	identifier of the secret chat
admin_id	int	TID of the user who created the secret chat
date	int	creation date and time (in Unix epoch format) of the secret chat
participant_id	int	TID of the user who joined the secret chat after creation

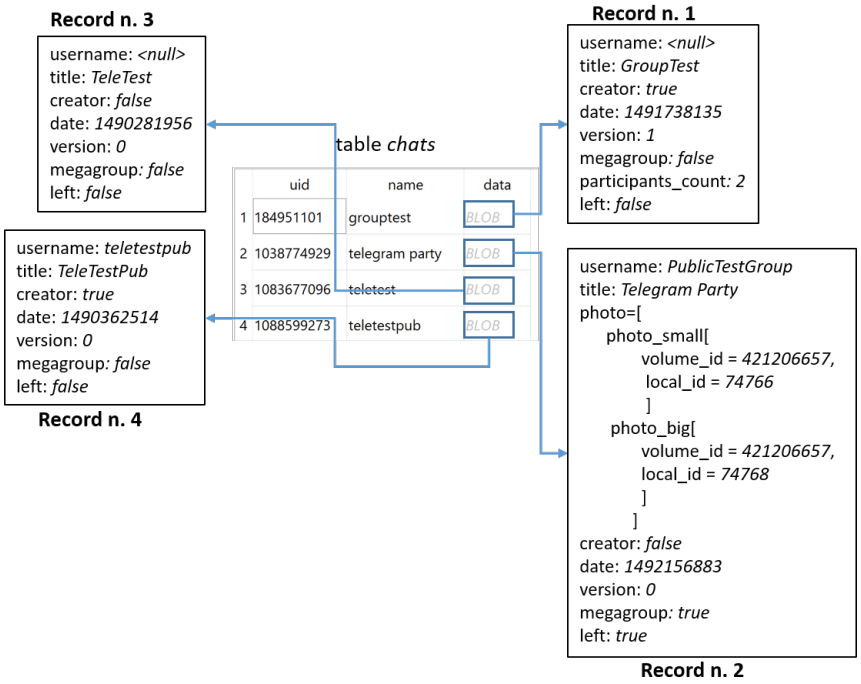

Figure 8: Determining the properties of (super)groups and channels.

Table 12: Structure of table chats and of the TDS Chat it contains.

Structure of table chats		
Field	**Type**	**Meaning**
uid	int	identifier of the (super)group/channel
name	string	name given to the public (super)group/channel, *null* if private
data	TDS Chat	stores additional information about the (super)group/channel
Structure of TDS Chat		
Field	**Type**	**Meaning**
id	int	identifier of the (super)group/channel (same value stored in field uid)
username	string	username if the (super)group/channel is public, *null* if it is private
title	string	name given by the creator
photo	TDS UserProfilePhoto	profile photo of the (super)group/channel (see Table 8)
creator	bool	*True* if the user is the creator of the (super)group/channel, *False* otherwise
date	int	date and time of creation (if creator=*True*) or joining (creator=*False*), expressed in Unix epoch format
admin_enabled	bool	*True* if the (super)group/channel has administrators, *False* otherwise
admin	bool	*True* if the local user is an administrator of the (super)group/channel, *False* otherwise
version	int	0 for channels or supergroups, 1 for standard groups
megagroup	bool	*True* for supergroups or channels, *False* for standard groups
participants_count	int	number of participants for standard groups (set to 0 for channels and supergroups)
left	bool	*True* if the user has left the (super)group/channel, *False* otherwise

- Record no. 2 corresponds to a public (username=*PublicTestGroup*) supergroup (version=0 and megagroup=*True*), whose title is '*Telegram Party*', which the local user has joined (creator=*False*) on April 14, 2017 at 8:01:23 UTC (date=1492156883), and has subsequently left (left=*True*). This supergroup is associated with a profile photo, stored in files *421206657_74766.jpg* (smaller size) and *421206657_74768.jpg* (larger size), both stored in the cache folder (see Fig. 2).

- Record no. 3 refers to a private (username = *null*) channel (version=0 and megagroup=*False*), whose title is '*TeleTest*', that has been joined by the local user (creator=*False*) on March 23, 2017 at 15:12:36 UTC (date=149028195), and to which the user is still participating as a member (left=*False*).

- Record no. 4 refers to a public (username = *teletestpub*) channel (version=0 and megagroup=*False*), whose title is '*TeleTestPub*', which has been created by the local user (creator=True) on 24 March, 2017 at 13:35:14 UTC (date = 1490362514).

4.5.4.4. Reconstructing the chronology and contents of messages.
The information concerning the messages exchanged in the various dialogs, including secret chats, is stored in table messages (see Table 13), that contains a record for each message. This record stores the unique *Message Identifier* (*MID*) and the DID of the corresponding dialog (fields mid and did, respectively), whether the message has been sent or not (field send_state), whether it has been read or not (field read_state), whether it is an incoming or outgoing message (field out), the date of the last modification of its status (field date), and its contents that are encoded into a TDS of type Message (see Table 14) stored in field data.

The reconstruction of the chronology of message exchanges can be carried out by means of the information stored in fields out, date, send_state, and read_state of table messages.

As an example, in Fig. 9 we report the chronology of messages exchange reconstructed from the records of table messages shown in Fig. 6.

The contents of both textual and non-textual messages are instead encoded into the TDS of type Message (see Table 14) stored in field data of table messages. It is worth pointing out that the following discussion applies also to messages belonging to secret chats, that are stored unencrypted in the main database.

Table 13: Structure of table messages.

Field	Type	Meaning
mid	int	Message ID (MID) of the message
uid	int	DID of the dialog the message belongs to
read_state	int	2: delivered but not read, 3: delivered and read
send_state	int	indicates whether the message has been already sent (0) or it is pending (1). If pending, the mid is negative, and the field date corresponds to the timestamp when message sending has been requested. After actual transmission, mid gets the next available sequence number, and the field date is modified with the timestamp of the sending
date	int	date and time of the last modification of the message status, expressed in the Unix epoch time format
data	TDS Message (see Table 14)	stores information about the message content, sender, and recipient
out	int	indicates whether the message has been sent (1) or received (0) by the user

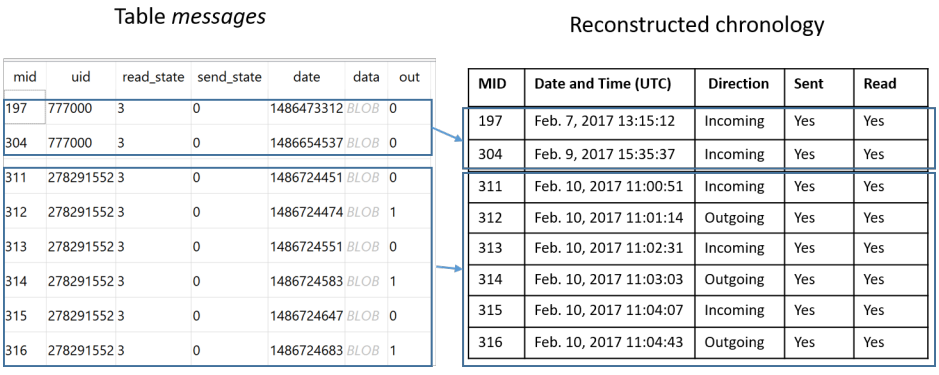

Figure 9: Reconstruction of the chronology of message exchanges for the two dialogs shown in Fig. 6.

Table 14: Structure of the TDSs used to encode information about messages.

Structure of TDS Message		
Field	**Type**	**Meaning**
message	string	for textual messages, stores the body of the message, encoded as an UTF-8 string
media	TDS MessageMedia	for non-textual messages, stores the details about transferred files
action	TDS messageActionPhoneCall (see Table 20)	details of the voice call (for records corresponding to voice calls, see Sec. 4.5.5)
attach_path	string	for non-textual messages, stores the pathname of the file that has been sent
from_id	int	TID of the sender (for channels, stores the DID of the dialog the message belongs to)
to_id	TDS Peer	Information about the recipient (content depends on the type of dialog)
Structure of TDS Peer		
Field	**Type**	**Meaning**
user_id	int	TID of the correspondent user for regular chat (0 for other dialog types)
channel_id	int	Identifier of the channel (0 for other dialog types)
chat_id	int	Identifier of the (super)group of the sender (0 for other dialog types)

In particular, the content of textual messages is stored in field message. Conversely, for non-textual messages two distinct fields are used, namely: (a) field media, that encodes into a TDS of type MessageMedia either the content or the information about its location (depending on the type of data carried by the message), and (b) field attach_path, that stores the location of the file that has been sent (if any). The other fields contain instead the identifiers of the sender (field from_id) and of the recipient (field to_id) of the message.

While the decoding of the content of a textual message is straightforward, as it is stored as an UTF-8 string in field message, for non-textual ones the situation is more elaborate. In particular, the MessageMedia TDS, stored in field media, is instantiated to a different TDS according to its content type, as specified in Table 15.

In the following, we describe how to decode the information stored in the TDS specific for each content type, and how to retrieve the corresponding content.

Image files

Table 15: List of TDSs used to store the details about non-textual contents.

Type of file/information	**Instantiated with**
picture	TDS messageMediaPhoto
video, audio, generic	TDS messageMediaDocument
contact	TDS messageMediaContact
geo-coordinates	TDS messageMediaGeo or TDS messageMediaVenue

When a file containing an image is sent or received, Telegram Messenger creates several copies of it, that differ from each other in their size, and stores them either in the Telegram Images subfolder of the media folder or in the cache folder (see Fig. 2).

The details of these files are stored in the messageMediaPhoto TDS (see Table 16) that – for image files – instantiates the MessageMedia TDS contained in field data of table messages. In particular,[5] the information about

Table 16: Structure of TDSs used to store the information about exchanged pictures.

Structure of TDS messageMediaPhoto		
Field	**Type**	**Meaning**
id	long	picture identifier
user_id	int	TID of the sender
date	int	date of creation of the picture (optional)
caption	string	textual description of the picture (optional)
geo	GeoPoint (see Table 19) TDS	description of the geo-coordinates associated with the picture (optional)
sizes	list of PhotoSize TDS	list of available images
Structure of TDS PhotoSize		
Field	**Type**	**Meaning**
type	string	type of the thumbnail[5]
location	FileLocation TDS (see Table 8)	location of the file on the local storage
w	int	image width
h	int	image height
size	int	size in bytes of the file

these files is stored in field sizes, that contains a list of TDSs of type PhotoSize, each one storing the detail concerning one of them, and in particular its name (field location), its width (field w) and height (field h), and its size in bytes (field size).

[5]See https://core.telegram.org/constructor/photoSize for the list of possible thumbnail types.

Other possibly relevant information stored in messageMediaPhoto are the TID of the sender (field user_id), and various optional fields (storing the date in which the picture has been created, a textual caption specified by the sender, and geo-coordinates of the place where the picture has been taken).

Finally, in the sender's database, attribute attach_path of the Message TDS (see Table 14) reports where – on the device file system - it is stored the file that has been sent.

An example is depicted in Fig. 10, where we show how to decode the data field of a record in table messages corresponding to a picture that has been sent, together with the copies of this file that have been saved in the folders mentioned before.

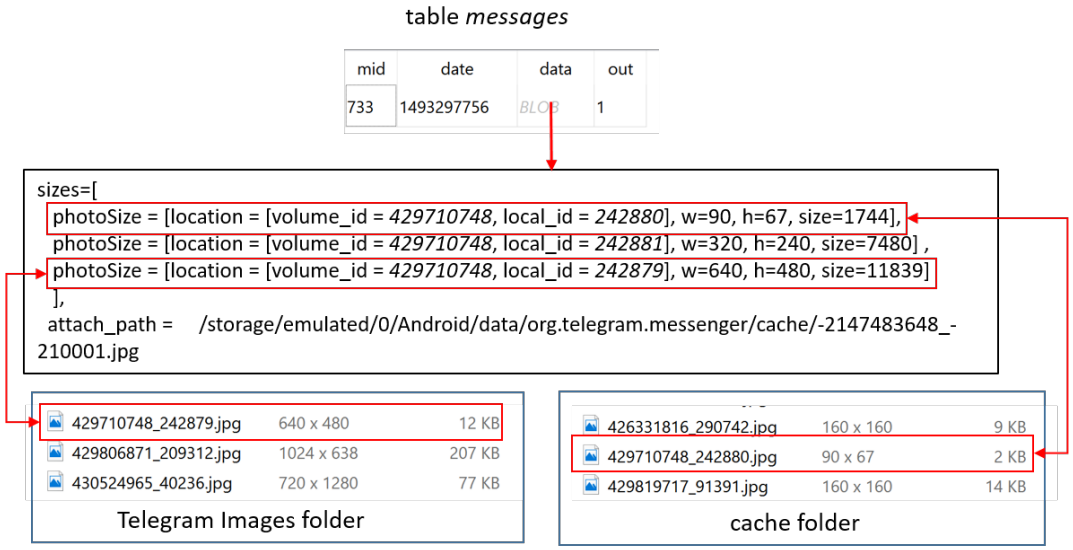

Figure 10: Data stored in the main database for a picture that has been sent.

In the figure, we see the path of the file that has been sent (field attach_path). We also see that three distinct copies (whose details are stored in attribute sizes) have been sent. However, as also shown in the bottom part of Fig. 10, only two of them are actually present either in the cache or in the Telegram Images folder. It is unclear whether the third one has been transferred (and, subsequently, automatically deleted by Telegram) or not. The situation in the recipient database is exactly the same shown in Fig. 10, with the only difference that, in this case, the attach_path attribute is empty.

Audio, video, and generic files
For files different from images, Telegram Messenger stores only one copy (instead of multiple ones) in one of the subfolders of the Telegram media folder (see Fig. 2), according to its specific type. In particular, audio, video and generic files are stored in folder Telegram Audio, Telegram Video, and Telegram Documents, respectively.

The information concerning these files are stored in the messageMediaDocument TDS (see Table 17) that – for files of the above type – instantiates the MessageMedia TDS contained in field data of table messages.

Table 17: Structure of TDSs used to store the information about exchanged files other than pictures (only forensically-relevant fields are shown).

Structure of TDS messageMediaDocument		
Field	**Type**	**Meaning**
id	long	filename of the file on local storage
user_id	int	TID of the sender
date	int	date of creation of the file (optional)
mime_type	string	MIME type of the file
size	int	size in byte of the file
thumb	list of PhotoSize TDS	list of available thumbnail images (only for video files)
attributes	list of either documentAttibuteVideo, documentAttributeAudio, or documentAttributeFileName TDSs (the actual instantiation depends on the file type)	information about the file
Structure of TDS documentAttibuteVideo and documentAttibuteAudio		
Field	**Type**	**Meaning**
duration	int	duration (in seconds) of the audio/video
performer	string	name of the author of the audio/video
title	string	title of the audio/video file
w	int	video width
h	int	video height
Structure of TDS documentAttibuteFileName		
Field	**Type**	**Meaning**
file_name	string	name of the file as on the sender's file system

As reported in this table, the name of these files is an integer value stored in field id (unlike image files that, instead, use a FileLocation TDS), while the TID of the sender is stored in field user_id. Other information include the size in bytes of the file (field size), its (optional) date of creation (field date), and the type of the data it stores (field mime_type).

Other properties of the file are encoded into the TDSs stored as a list in field attributes, that are instantiated with either documentAttributeAudio, documentAttributeVideo, or documentAttributeFileName TDSs according to the file type. The attributes of these TDS are also described in Table 17.

An example is reported in Fig. 11, where we show how to interpret the information stored in the TDS messageMediaDocument of three records of table messages, corresponding to a video, an audio, and a PDF file sent by the user.

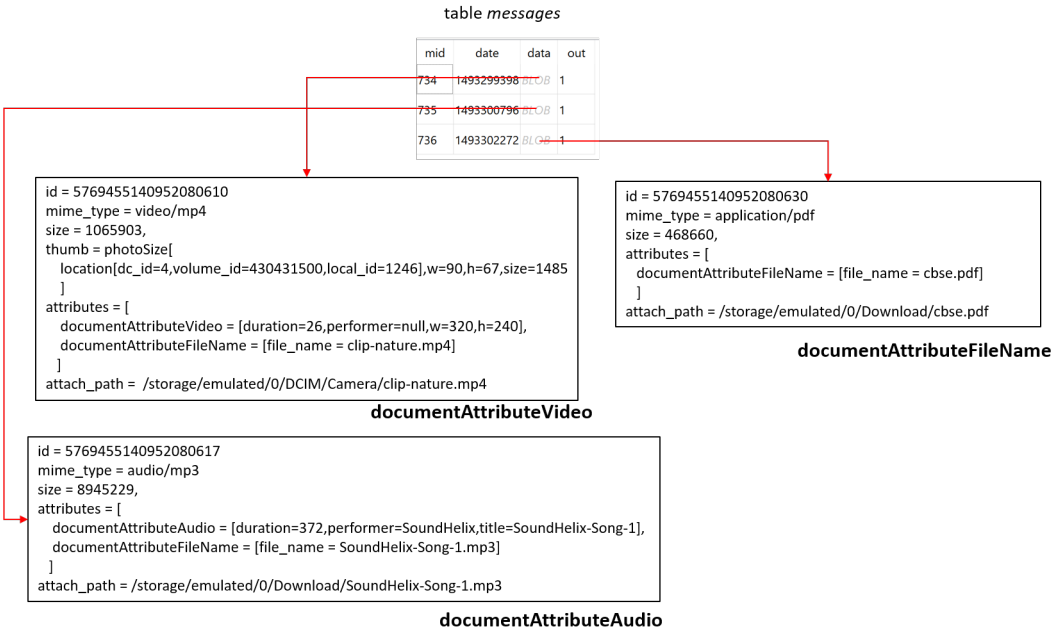

Figure 11: Data stored in the main database for files of various types that have been sent.

Phone contacts

In addition to files, Telegram allows users to exchange information about contacts using non-textual messages. As indicated in Table 15, for this type of non-textual messages Telegram Messenger uses a messageMediaContact TDS (see Table 18) to store the details concerning the contact that has been exchanged. An example is shown in Fig. 12, where record with mid=9 corresponds to the sending of a contact whose phone number is 111222333, and whose first and last names are *Fake* and *Contact*, respectively.

Table 18: Structure of messageMediaContact TDS.

Structure of TDS messageMediaContact		
Field	**Type**	**Meaning**
phone_number	string	phone number of the contact
first_name	string	first name of the contact (as given by the sender)
last_name	string	last name of the contact (as given by the sender)

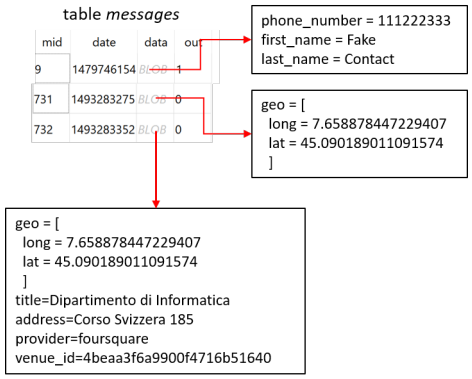

Figure 12: Data stored in table contacts for two non-textual messages corresponding to a contact and two geo-locations.

Geo-coordinates

The last type of non-textual message allowed by Telegram transfers is geo-location information. As indicated in Table 15, for this type of non-textual messages Telegram Messenger uses either a messageMediaVenue or a messageMediaGeo TDS (see Table 19) to store the details concerning the geo-location information that has been exchanged.

As can be seen from this table, messageMediaGeo has only field geo, and is used when only the geo-coordinates are sent, while messageMediaVenue has also fields title, address, provider, and venue_id, and is used when also a map is transferred.

An example is shown in Fig. 12, where the record of table messages whose mid=731 corresponds to the sending of just the coordinates of a point, while the one with mid=732 corresponds to the sending of a map. For the latter

Table 19: Structure of TDSs used to store the information about exchanged geo-locations.

Structure of TDS messageMediaVenue and messageMediaGeo		
Field	**Type**	**Meaning**
geo	TDS GeoPoint	geo-coordinates of the location
title	string	title of the map (for messageMediaVenue only)
address	string	address corresponding to coordinates (for messageMediaVenue only)
provider	string	identifier of the map provider (for messageMediaVenue only)
venue_id	string	identifier of the map (for messageMediaVenue only)
Structure of TDS GeoPoint		
Field	**Type**	**Meaning**
long	double	longitude of the location
lat	double	latitude of the location

one, we see also that field provider is set to '*foursquare*', and field venue_id stores the identifier that can be used to retrieve the map using the FourSquare API.

4.5.5. Reconstruction of the voice calls log

To reconstruct the logs of the voice calls made or received by the user, we carry out the experiments reported in Table 5. The analysis of the results collected during these experiments show that the data stored in the main database allow one to reconstruct the log of the voice calls in which the user has been involved. In particular, we found that, for each incoming or outgoing call, Telegram Messenger stores a record in table messages, that in turn stores into field action of the corresponding Message TDS (see Table 14) the identities of the calling and called parties, the duration of the call and, for unsuccessful calls, the reason of the failure.

More specifically, field action stores a TDS of type messageActionPhoneCall (see Table 20), where the above information are encoded.

As can be observed from Table 20, this TDS stores the identifier of the call (field call_id), its duration (field duration), and the reason of its termination (field reason). The TIDs of the partners involved in the call are instead stored in fields from_id and to_id of the Message TDS for outgoing calls, while for incoming calls both these fields store the value of the caller (as the callee is the local user).

An example is shown in Fig. 13, where we show the records in table

Table 20: Structure of TDS messageActionPhoneCall.

Structure of TDS messageActionPhoneCall		
Field	**Type**	**Meaning**
call_id	int	identifier of the call
duration	int	duration of the call (in seconds)
reason	string	reason of the termination of the call: possible values are *hangup*, *missed*, and *busy*

messages corresponding to four distinct voice calls, together with the information recovered from the various relevant fields of the TDS stored in their data fields. From the data shown in Fig. 13, it is easy to see that the

table *messages*

mid	date	data	out
64864	1492761889	*BLOB*	1
64868	1492762054	*BLOB*	1
64869	1492762187	*BLOB*	0
64870	1492762304	*BLOB*	0

call details

mid	from_id	to_id	call_id	duration	reason
64864	116935258	264990279	502233110225450005	69	hangup
64868	116935258	264990279	502233109243660426	0	busy
64869	264990279	264990279	1138124583031812013	70	hangup
64870	264990279	264990279	1138124583844004291	0	missed

Figure 13: Data stored in the main database for four voice calls.

first two records correspond to outgoing calls towards the user whose TID is *264990279*, while the last two correspond to incoming calls coming from that user. We can also see that the first and third calls have been successful, and lasted 69 and 70 seconds, respectively, while the second and fourth ones were unsuccessful because the user was busy (second) and did not answer (fourth). Finally, the date and time of the call can be retrieved by decoding the field date of the records in table messages.

4.5.6. Dealing with deletions

In the attempt to hide past interactions, the user may delete various information from the main database, namely contacts, individual messages, or entire chats.

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 in some cases (notably, if the SQLite engine has

not vacuumed the involved tables yet), they can be recovered (Jeon et al., 2012).

To verify whether such a recovery is possible with Telegram Messenger, during the various experiments we performed, we collected the main database after the various delete operations reported in Tables 2, 3, and 4, and we analyzed them with Oxygen Forensic SQLite Viewer (Oxygen Forensics, Inc., 2013b) (a tool which is able to recover deleted records from SQLite databases).

This analysis revealed indeed that deleted records persist in the main database, and may therefore be recovered. However, in general, the persistence time of these records is unpredictable, as they are permanently deleted when the database is vacuumed, an operation that is under the complete control of the SQLite engine. Therefore, although the possibility of recovering deleted information exists in theory, it cannot be assumed that – in general – such a recovery is possible for the specific case at hand.

5. Conclusions

In this paper we have presented a methodology for the forensic analysis of applications running on Android, and we have applied it to the analysis of Telegram Messenger. The general methodology is based on the execution of suitably designed experiments on virtualized smartphones, instead of physical ones, so as to obtain generality and reproducibility of the results, and has been suitably instantiated for Telegram Messenger. The accuracy of the results obtained by using our methodology has been assessed by validating them against those obtained from the execution of a subset of the experiments on a real smartphone.

Thanks to the application of this methodology, we have been able to identify all the artifacts left by Telegram Messenger on Android smartphones, and we have shown how these artifacts can provide many information of investigative value. In particular, we have discussed methods to interpret the data stored in the main database and in the user configuration file, to determine the specific Telegram account used on the local device, and to reconstruct the contact list of the user, the chronology and contents of exchanged textual and non-textual messages, as well as to determine whether the user created or administered a secret chat, a (super)group, or a channel.

More importantly, we have also shown the importance of correlating among them the artifacts generated by Telegram Messenger in order to gather

information that cannot be inferred by examining them in isolation. As a matter of fact, while the analysis of user profile information, or of the contact list, makes it possible to determine some of the relevant information (e.g., the Telegram identifier of these users or their phone number), other information of investigative interest (such as the profile photos, or the files that have been exchanged) may be determined only by correlating the data stored in the main database with the files stored in the various folders used by Telegram Messenger.

The results discussed in this paper have a two-fold value. On the one hand, they provide analysts with the full picture concerning the decoding, interpretation, and correlation of Telegram Messenger artifacts on Android devices. On the other hand, they represent a benchmark against which the ability of mobile forensic platforms to retrieve and correctly decode all the Telegram Messenger artifacts can be assessed.

As future work, we plan to investigate the extension of our methodology to other mobile platforms (most notably, iOS and Windows Phone). As a matter of fact, while the individual stages of the methodology are agnostic with respect to the mobile platform, its core properties of repeatability and generality require the availability of a virtualization solution enabling the use of virtualized smartphones. At the moment of this writing, virtualization platforms for Windows Phones are available (Microsoft Corp., 2017), while the same is not true for iOS. Once the methodology has been extended, it will be possible to analyze Telegram Messenger, as well as other applications, on these mobile platforms.

References

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

Al Barghuthi, N., Said, H., Nov. 2013. Social Networks IM Forensics: Encryption Analysis. Journal of Communications 8 (11), 708–715.

Anglano, C., Sept. 2014. Forensic Analysis of WhatsApp Messenger on Android Smartphones. Digital Investigation 11 (3), 201–213.

Anglano, C., Canonico, M., Guazzone, M., Dec. 2016. Forensic Analysis of the ChatSecure Instant Messaging Application on Android Smartphones. Digital Investigation 19, 44–59.

Anglano, C., Canonico, M., Guazzone, M., 2017. Configuration and Use of Android Virtual Devices for the Forensic Analysis of Android Applications. Technical Report TR-INF-2017-06-02-UNIPMN, University of Piemonte Orientale, http://www.di.unipmn.it/TechnicalReports/TR-INF-2017-06-02-UNIPMN.pdf.

Azfar, A., Choo, R., Liu, L., Sept. 2016. An Android Communication App Forensic Taxonomy. Journal of Forensic Sciences 61 (5).

C. Budd, Aug. 2016. Following the mark: Hackers begin to leverage Telegram messaging app. Available at https://goo.gl/Q84fJe.

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

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

Compelson Labs, 2017. Mobiledit Forensic Express. Available at http://www.mobiledit.com/forensic-solutions/.

DrKLO, 2017. Telegram Messenger for Android. Available at https://github.com/DrKLO/Telegram.

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

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

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

Gregorio, J., Gardel, A., Alarcos, B., 2017. Forensic analysis of telegram messenger for windows phone. Digital Investigation (In Press).

Husain, M. I., Sridhar, R., 2010. iForensics: Forensic Analysis of Instant Messaging on Smart Phones. In: Goel, S. (Ed.), Digital Forensics and Cyber Crime. Vol. 31 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg.

J. Warrick, Dec. 2016. The app of choice for jihadists: ISIS seizes on Internet tool to promote terror. The Washington PostAvailable at https://goo.gl/3MKSnP.

Jeon, S., Bang, J., Byun, K., Lee, S., 2012. A recovery method of deleted records for SQLite database. Personal and Ubiquotous Computing 16 (6), 707–715.

Marc McLoughlin, 2008. The QCOW2 Image Format. Available at https://people.gnome.org/ markmc/qcow-image-format.html, accessed on June 21st, 2017.

Mehrotra, T., Mehtre, B. M., Dec 2013. Forensic analysis of Wickr application on android devices. In: 2013 IEEE International Conference on Computational Intelligence and Computing Research. pp. 1–6.

Micro Systemation, 2016. XRY. Available at http://www.msab.com/xry/xry-current-version.

Microsoft Corp., 2017. Windows Phone Emulator for Windows Phone 8. `https://msdn.microsoft.com/en-us/library/windows/apps/ff402563(v=vs.105).aspx`.

Ovens, K. M., Morison, G., Jun. 2016. Forensic analysis of Kik messenger on iOS devices. Digital Investigation 17.

Oxygen Forensics, Inc., 2013a. Oxygen Forensics. Available at http://www.oxygen-forensic.com/en/features/analyst.

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

Satrya, G. B., Daely, P. T., Nugroho, M. A., Oct 2016a. Digital forensic analysis of Telegram Messenger on Android devices. In: 2016 International Conference on Information Communication Technology and Systems (ICTS). pp. 1–7.

Satrya, G. B., Daely, P. T., Shin, S. Y., July 2016b. Android forensics analysis: Private chat on social messenger. In: 2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN). pp. 430–435.

Susanka, T., Jan. 2017. Security Analysis of the Telegram IM. Master's thesis, Czech Technical University inPrague, Faculty of Information Technology, https://www.susanka.eu/files/master-thesis-final.pdf.

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

Telegram Messenger LLP, Feb. 2016. 100,000,000 Monthly Active Users. Available at https://telegram.org/blog/100-million.

Telegram Messenger LLP, May 2017a. Binary Data Serialization. Available at https://core.telegram.org/mtproto/serialize.

Telegram Messenger LLP, May 2017b. TL Language. Available at https://core.telegram.org/mtproto/TL.

Telegram Messenger LLP, May 2017c. TL Schema. Available at https://core.telegram.org/schema.

Telegram Messengers LLP, 2017. Telegram Applications. Available at https://telegram.org/apps.

The Telegram Team, Jan. 2017. Android Developers Never Sleep. Available at https://telegram.org/blog/unsend-and-usage#android-developers-never-sleep.

Tso, Y.-C., Wang, S.-J., Huang, C.-T., Wang, W.-J., 2012. iPhone Social Networking for Evidence Investigations Using iTunes Forensics. In: Proc. of the 6th International Conference on Ubiquitous Information Management and Communication. ICUIMC'12. ACM, New York, NY, USA, pp. 1–7.

United Nations Office on Drugs and Crime, Feb. 2013. Comprehensive Study on Cybercrime. Tech. rep., United Nations.

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

Walnycky, D., Baggili, I., A.Marrington, Moore, J., Breitinger, F., 2015. Network and device forensic analysis of Android social-messaging applications. Digital Investigation 14, Supplement 1, S77–S84, proc. of 15th Annual DFRWS Conference.

Wu, S., Zhang, Y., Wang, X., Xiong, X., Du, L., 2017. Forensic analysis of WeChat on Android smartphones. Digital Investigation.

Zhang, L., Yu, F., Ji, Q., July 2016. The Forensic Analysis of WeChat Message. In: 2016 Sixth International Conference on Instrumentation Measurement, Computer, Communication and Control (IMCCC). pp. 500–503.

Zhou, F., Yang, Y., Ding, Z., Sun, G., June 2015. Dump and analysis of Android volatile memory on Wechat. In: 2015 IEEE International Conference on Communications (ICC). pp. 7151–7156.