You Can Run but You Cannot Hide from Memory: Extracting IM evidence of Android Apps

Antonia Nisioti, Alexios Mylonas, Vasilios Katos, Paul D. Yoo
Department of Computer and Informatics
Bournemouth University
Bournemouth, United Kingdom
{anisioti, amylonas, vkatos, pyoo}@bournemouth.ac.uk

Anargyros Chryssanthou
Hellenic Data Protection Authority
Athens, Greece
achrysanthou@dpa.gr

***Abstract*—Smartphones have become a vital part of our business and everyday life, as they constitute the primary communication vector. Android dominates the smartphone market (86.2%) and has become pervasive, running in 'smart' devices such as tablets, TV, watches, etc. Nowadays, instant messaging applications have become popular amongst smartphone users and since 2016 are the main way of messaging communication. Consequently, their inclusion in any forensics analysis is necessary as they constitute a source of valuable data, which might be used as (admissible) evidence. Often, their examination involves the extraction and analysis of the applications' databases that reside in the device's internal or external memory. The downfall of this method is the fact that databases can be tampered or erased, therefore the evidence might be accidentally or maliciously modified. In this paper, a methodology for retrieving instant messaging data from the volatile memory of Android smartphones is proposed, instead of the traditional database retrieval. The methodology is demonstrated with the use of a case study of four experiments, which provide insights regarding the behavior of such data in memory. Our experimental results show that a large amount of data can be retrieved from the memory, even if the device's battery is removed for a short time. In addition, the retrieved data are not only recent messages, but also messages sent a few months before data acquisition.**

Keywords— Android; Android forensics; Memory forensics; Instant messaging apps

I. INTRODUCTION

Nowadays the use of mobile devices, such as smartphones, tablets etc., has increased and they have become pervasive. In just over 20 years, the number of mobile subscriptions has reached 7.4 billion [1], equal to the population of Earth (7.4 million) [2]. Out of these devices, 3.4 billion are smartphones, a number that was estimated to surpass those of feature phones in the third quarter of 2016[1]. These devices contrary to feature phones, offer advanced processing and networking capabilities, instead of simply offering the ability to receive and place calls and SMS. In addition, smartphones offer a plethora of applications (or 'apps'), which extend the functionality of the device. According to a study by Pearson et al. in [3], 13 percent of participants classified themselves as addicted to smartphones, with an average daily usage of 3.6 hours per user. Furthermore, smartphones are not only popular among individual consumers but also governments and enterprises, as discussed in [4].

Nonetheless, one of the main reasons of using a smartphone is communication. In the past, voice calls and SMS were the communication vector, but nowadays Instant Messaging (IM) applications are the users' preferred communication method. Specifically, chat application messaging had already overtaken SMS texts in 2012, when 19 billion messages were sent by applications of this type and 17.6 billion by SMS [5]. The user is not only able to send and receive text messages, but also create group conversations, place free calls, send pictures and videos, etc. IM apps are available in different operating systems (OS), in order to satisfy different types of smartphone users (e.g. iOS users, Android users, etc.). Android is the OS that dominates the smartphone market having an 86.2% market share, followed by iOS with 12.9% and Windows Phone with 0.6% [6].

As discussed in [7], [8], a smartphone contains a plethora of heterogeneous user data (personal, business), and is therefore a valuable source of artefacts for a forensic analyst. One could compare a smartphone with a diary, as it contains every detail about the daily life of its owner, from calls and messages to GPS locations and internet history. Forensic examination of mobile devices is not an easy task, as it involves a variety of different hardware, operating systems and applications. Applications typically store their data either in the internal memory of the device or to an external SD card (frequently in a database). Both storage locations are easy to be logically modified / erased by a suspect, criminal or even another application. In this case the only remaining location where evidence might be still retrievable is the device's physical memory (RAM). Memory forensics is a growing field of digital forensics and a vital part of every investigation involving desktop computers/laptops. For instance, fileless malware leaves no trace on the hard drive but its code needs to be loaded in memory to be executed. Likewise, ransomware encrypts the victim's hard drive with a key that might be retrieved from memory in plain text [9]. Moreover, anti-forensic technologies, such as encryption, can prevent an investigator from accessing evidence, but often memory forensics can recover encryption keys to help overcome this problem. Finally, the analysis of volatile data is becoming particularly important in modern devices in general, due to the proliferation of Web 2.0 and the underlying technologies [10].

978-1-5386-1629-1/17/$31.00 ©2017 IEEE

However, to the best of our knowledge, currently the use of memory forensics for retrieving data from Android instant messaging apps is very limited. This paper presents a methodology for retrieving instant messaging data from the Random Access Memory of an Android mobile device. The methodology is applied with a case study that provides insights regarding the behavior of memory on such a device.

The rest of the paper is organized as follows: Section 2 provides background regarding Android OS, the related work of memory forensics and acquisition on this type of devices. Section 3 presents the methodology for the retrieval of instant messaging data from Android memory. Section 4 presents a case study where the proposed methodology is applied in four simulated scenarios and discusses the results of our experiments. Finally, Section 5 concludes the paper and discusses limitations and future work.

II. BACKGROUND

A. Android OS

Android is an operating system created initially for mobile devices, such as smartphones and tablets, but nowadays it has become ubiquitous and popular in other 'smart' devices, e.g., cars, televisions and watches. Its kernel is Linux-based, but also includes components that are not typically found in a Linux kernel. Above the kernel, the libraries, the application framework and finally the applications are typically found [11]. All applications store their data in the /data directory.

Android has a Linux Out-Of-Memory based task killer implemented in its kernel, the Low Memory Killer (LMK), which is responsible for the device's memory management [12]. Once the RAM usage excesses a threshold, LMK starts killing processes to free memory [13]. When an application is launched it is stored in memory where it stays even if the user closes it, until the operating system decides it is necessary to remove it. The reason for this is to minimize the app's response time, by letting the app run in RAM in the background, without affecting the battery or the performance of the device. This is particularly useful for IM apps, which have to receive real time messages and calls, regardless of whether they have been launched or not. For this reason, they are loaded along with their essential data to the memory after every boot. Consequently, IM data of these applications reside in the memory even after they are deleted, assuming no reboot has occurred in the meantime.

B. Android Memory Analysis

Currently, the examination of Android devices typically includes: a) logical acquisition that enables an examiner to access only the data permitted by the operating system (e.g., manually examine the databases of the installed applications and the user's data [14]) and b) physical acquisition of the device with forensic software (e.g., XRY, Cellebrite, FTK Imager), in order to acquire data that cannot be retrieved with a logical acquisition – assuming that the device is not rooted/jailbroken – such as deleted files or system protected files [15]. The physical memory of a system is like a "photograph" of its state at the time of the acquisition. Memory of flagship (Android) smartphone devices nowadays contains considerable user and system activity that can remain, depending on conditions such as the power supply, the reboot frequency, etc. It is worth noting that typically smartphone users do not reboot or shutdown their device (contrary to desktops/laptops). Also, flagship (Android) smartphone devices nowadays contain a considerably large amount of memory, allowing more data to be stored and to reside in it until LMK decides to free the memory space that they occupy.

Memory forensics is a growing field in digital forensics, as vital evidence might exist only in memory, while other data might continue to reside there even after their deletion from other sources. There is currently a limited number of tools capable of analyzing memory images ("memory dumps") from Android devices. The simplest but still very useful method to extract data from a memory dump is with the Unix "strings" command. Unfortunately, this method restricts the analyst to data that can be interpreted as strings, such as words or phrases. Another method to extract data from a memory dump is to use the Volatility Framework [16], which is currently the 'swiss-army tool' for memory analysis for academics, enterprises and law enforcement. Volatility is an extensible, cross-platform, modular platform written in Python. It enables analysts to analyse memory dumps from Linux, Windows, Mac and Android systems and entails a large variety of build-in and custom profiles and plugins. As a result of its extensibility and compatibility, it has a very large and active community of users and developers, which allows it to keep up with new technologies. Currently, there are only a few Linux–related plugins, which became compatible with Android memory images after the "address space" for ARM architecture was added to them [17], and most of them retrieve system data. Finally, another method of analyzing Android memory dumps is with the tool Volatilitux [18]. This tool is similar to Volatility, but targets only Linux systems and thus can be used for Android devices. It includes a limited set of five commands for listing and dumping processes, files related to them and their memory map.

C. Linux and Android memory acquisition

Formerly, Linux memory dumps were acquired by accessing the /dev/mem device file with the use of commands such as dd (with root privileges). Nowadays, on nearly all Linux distributions this feature is disabled for security reasons. Furthermore, this method allows the acquisition of only the first 896MB of RAM, even if more space is available. For this reason and due to the advent 64-bit systems, the /dev/mem memory acquisition method has been abandoned. Currently, three tools are mainly used to acquire the physical memory:

i. fmem [19] that is a loadable kernel module and operates by creating the character device /dev/fmem which can be used by other programs (e.g., dd) to access the memory.
ii. Linux Memory Extractor (LiME) [20] that creates a loadable kernel module (LKM) that runs within the kernel and dumps the content of RAM. In order to use LiME, the analyst has to cross-compile the LKM with the kernel of the device that will be acquired.

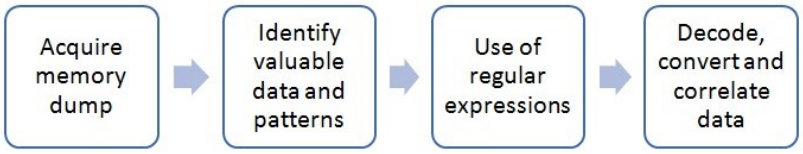

Fig. 1. Summary of our methodology

iii. Forensic Recovery of Scrambled Telephones (FROST) [21], a set of tools that perform cold boot attacks against Android devices in order to: a) retrieve disk encryption keys and other sensitive data from RAM and b) create a memory image.

III. METHODOLOGY

This section presents a methodology for the extraction of volatile data from Android instant messaging apps, which is summarized in Fig.1 Firstly, the memory of the device was acquired and then analysed by means of a hex editor to: (a) identify the data that existed in the memory dump, (b) identify and determine existing patterns, and (c) retrieve any conversations. Finally, further correlations were performed after the data retrieval.

Acquisition. LiME was selected for the acquisition of the memory dumps and Android Device Bridge (ADB) was used to transfer the memory dump back to the investigator's machine. As already mentioned, LiME creates a loadable kernel module (LKM), which has to be uploaded to the target device. A loadable kernel module is used to extend a running kernel, often to support new hardware, add a system call or access hardware. The procedure to create a LKM has three steps, namely: (a) obtain the appropriate source code for the device's kernel, (b) use the right toolchain to build the kernel from the source code, and (c) finally cross compile LiME with the built kernel, according to the instructions provided with it by the device's manufacturer. The source code along with the necessary instructions about compiling, building and also the appropriate toolchain, can be found at the manufacture's site in the case of official firmware, or at the developer's site in the case of a custom rom. The result of the aforementioned procedure is a .ko (kernel object) file that has to be uploaded to the target device and run against it through a shell with the use of ADB.

Pattern identification and regular expression creation. The simplest way of examining a memory dump is using a hex editor. The goal of this step is: i) to locate the messages in the file and then b) to identify the other data that exist along them and need to be retrieved, such as timestamps, data about the sender(s) or the receiver such as mobile numbers, IDs, etc. The aim of this step is to identify the patterns of the targeted data in multiple memory dumps, acquired in different days and times, in order to create accurate and non-circumstantial regular expressions to match them. The produced regular expressions are optimized with thorough testing to retrieve all the available messages and avoid false positives. As discussed in detail in Section 4, for each application more than one pattern might be observed during the analysis and, therefore, more than one regular expression might be used for the retrieval.

Decoding, conversion and correlation. Initially, the retrieved data are not directly readable as they are in hexadecimal form. For example, timestamps must first be converted from hexadecimal to decimal, the last three digits should be removed as they are redundant and the outcome should be converted to date and time. In the case of names, messages and other data that represent text, data have to be decoded with UTF-8 or UTF-16, depending on the used language. Respectively, all the other retrieved data, should be decoded and/or converted to the appropriate form according to their type. As presented later in detail (see Section 4), after the aforementioned process further correlation can be performed on the retrieved information, in order to create more knowledge. For instance, from one message the sender's name may be missing, but her ID might be present. In this case, her ID could be correlated to a linked list of retrieved IDs and names to identify the missing name.

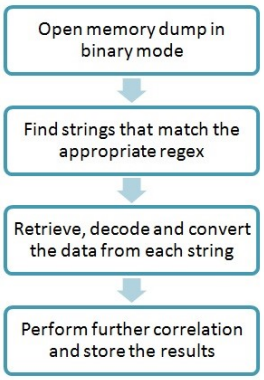

Fig. 2. Structure of the developed scripts for the retrieval of the targeted data

IV. CASE STUDY

This Section presents a case study in which the aforementioned methodology is used in order to collect evidence, which is created by the three most popular Android instant messaging apps. To perform the experiments that are described herein, a Samsung Galaxy III (GT-I9300) International was used, which is rooted and runs Cyanogenmod 10.2 (Android level 4.3) [22]. The three applications that were selected are the most frequently used instant messaging applications according to [23], namely Facebook v.30.0.0.30.174, Viber v.5.4.0.2519, and WhatsApp v.2.12.144.

Our experimental setup also includes as a host an Ubuntu 14.0 LTS 64-bit desktop computer. It is worth noting that the smartphone that was used in the experiments does not contain a limited amount of dummy data, but instead contains real-life instant messaging history of one year, as it was used regularly

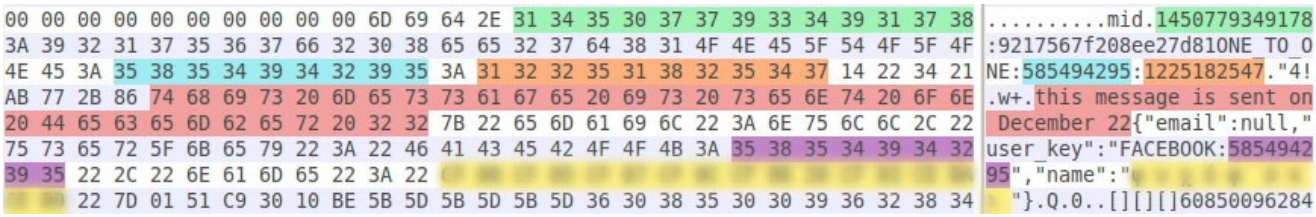

Fig. 3. Example of a retrieved string containing instant messaging data

before the experiments by one of the authors as the main device. Initially the device contained: 8159 individual WhatsApp messages, 12635 group WhatsApp messages, 12224 Viber messages, 6243 individual Facebook Messenger messages and 2815 group Facebook Messenger messages.

Five Python (v. 2.7) scripts were created for the retrieval of the instant messaging data, namely: two for WhatsApp (one for group chats and one for individual conversations), two for Facebook messenger and one for Viber. The structure of the scripts is presented in Fig. 2. Initially, the acquired memory dump is read as binary file. The message body along with the external communication data, which the script intents to retrieve as evidence, are stored in memory as one string. Consequently, in the second step a regex is used to find all the strings that match the appropriate pattern. This regex depends on a) the application that created the messages (i.e. Facebook, WhatsApp, and Viber) and b) on the type of conversation (group, individual) that the messages relate to (if applicable). As a result, a list of strings is created in the end of this step. In step three, each string is processed in order to retrieve and decode the desirable data, which will be stored in a database in the last step. Data of interest can be extracted from each string either with the use of regular expressions, or their offsets in the string (refer to Appendix A for the regexes that were used in this work). Data decoding depends on its type. For instance, names, numbers and text messages are converted from hexadecimal to UTF-8 or UTF-16, while UNIX timestamps are converted to decimal and then to time and date, taking into consideration the device's timezone, which is also retrieved with a regex.

Fig. 3, presents an instance of an individual Facebook Messenger conversation. The different data types inside the string are coloured differently and the name of the sender has been blurred. More specifically, date and time in Unix timestamp format is coloured in green, the message is in red, the sender's name in yellow, the device's owner id in orange (local id), the remote participant's id (remote id) depicted in blue and the sender's id in purple. Furthermore, the correlation between the three ids and the retrieved receivers' names, can be used to identify the receiver's id and name. The regular expression that can parse and extract the aforementioned data is given in the following regex provided in Snippet 1 and Table 1 provides its breakdown (Appendix A includes the rest regexes that were used in this work).

```
mid\x2E\d{13}:[0-9az]+ONE_TO_ONE:\d+:
\d+[\x00-\xFF]{2,300}[{]"email":null,
"user_key":"FACEBOOK:\d+","name": "[\x00-
\xFF]{2,50}"[}]
```

Snippet 1: Regex for individual Facebook Messenger conversation

As Table 1 suggests, some parts of the regular expression contain valuable data, such as the main body of the message, whereas others are just constant values, such as the first four characters "`mid\x2E`". Also, the string contains a sub pattern (i.e. `:[0-9a-z]+`) that we were not able to interpret.

Table 1. Break down of the regular expression used for retrieving Facebook Messenger conversations

Regular Expression	Valuable Info	Type
mid\x2E	No	Constant Value
\d{13}	Yes	Message id/ Unix timestamp
:[0-9a-z]+	Unknown	Unknown
ONE_TO_ONE:	No	Constant value
\d+:	Yes	Remote id
\d+	Yes	Local id
[\x00-\xFF]{2,300}	Yes	Main body of the message
[{]"email":null,"user_key" :"FACEBOOK:	No	Constant value
\d+	Yes	Sender's id
","name":	No	Constant value
"[\x00-\xFF]{2,50}"[}]	Yes	Sender's name

Table 2. Hexadecimal, decoded and converted values of the string presented in Fig. 3

Hex Value	Decoded Value	Converted Value	Type
31 34 35 30 37 37 39 33 34 39 31 37 38	1450779349178	Tue, 22 Dec 2015 10:15:49 GMT	Message id/ Unix Timestamp
35 38 35 34 39 34 32 39 35	585494295	-	Remote id
31 32 32 35 31 38 32 35 34 37	1225182547	-	Local id
74 68 69 73 20 6D 65 73 73 61 67 65 20 69 73 20 73 65 6E 74 20 6F 6E 20 44 65 63 65 6D 62 65 72 20 32 32	this message is sent on December 22	-	Main body of the message
35 38 35 34 39 34 32 39 35	585494295	-	Sender's id

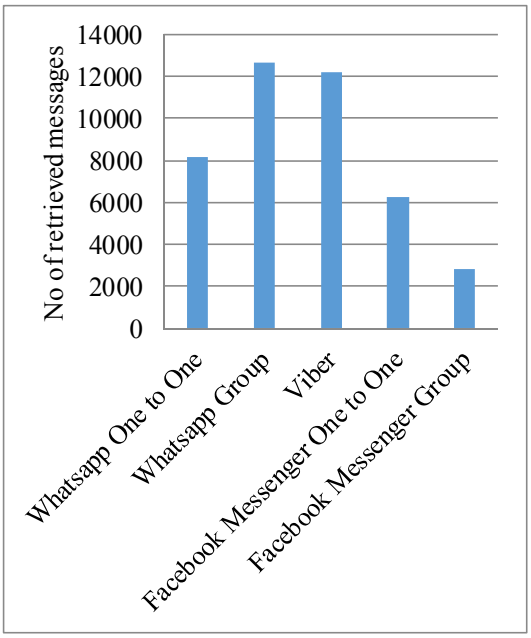

Fig. 4. Retrieved messages after a simple reboot (Experiment 1)

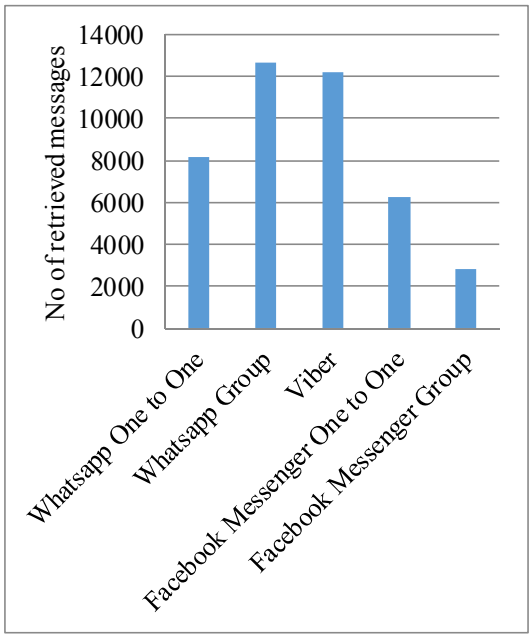

Fig. 5. Retrieved messages after battery removal (Experiment 2)

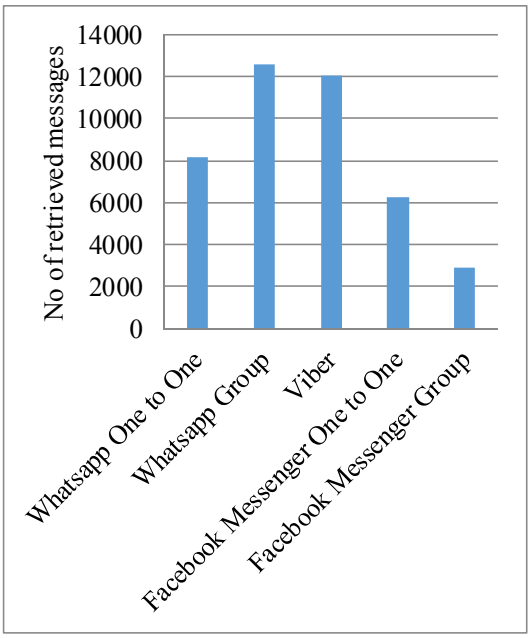

Fig. 6. Retrieved messages after a day of frequent usage (Experiment 3)

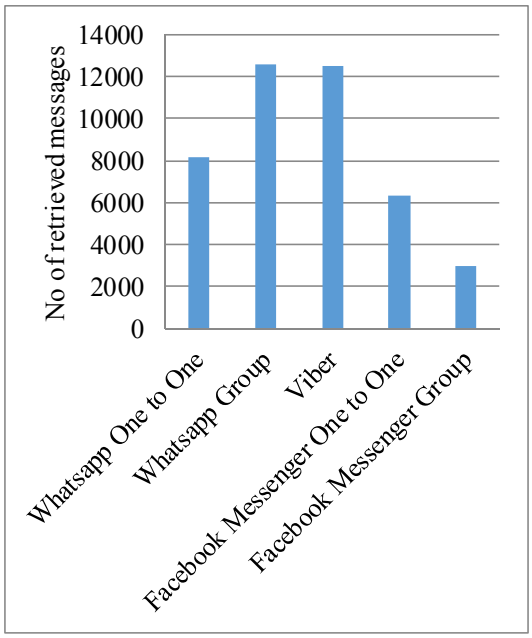

Fig. 7. Retrieved messages after a day of use and a reboot (Experiment 4)

Our work suggests that Facebook Messenger uses two patterns to store individual conversations in memory. As discussed earlier, this has to be taken into consideration when creating the patterns for an instant messaging application in order to retrieve the maximum number of messages. After their extraction the data are decoded and converted. Table 2, presents the hexadecimal, decoded and converted values of the string that is shown in Fig. 3, omitting the name of the sender.

The rest of this section presents the results from four experiments that were conducted to study the connection between the number of retrievable messages in the memory and the device's time of operation, since the last reboot, as well as the effect of a reboot or the removal of the battery on it.

The first memory dump was collected after a simple reboot of the device and without any application usage. The number of retrieved messages for each application are given in Fig. 4. The second memory dump was collected ten minutes after the first one, after the device was shut down, the battery was removed and replaced after 60 seconds, and the device was activated again. Again none of the applications was launched

after the activation. The purpose of this experiment is to "remove" the power of the RAM for an amount of time, in this case 60 seconds, and observe the effects on the number of the retrieved messages. It is worth noting that this work also examined the effect to data retrieval by the time before placing the battery back to the device, finding that time slots between 10 seconds to 2 minutes provided similar results. As Fig. 5 suggests, a huge amount of messages are retrievable despite the battery removal.

The third memory dump was acquired a day after the second one and its results are presented in Fig. 6. As already mentioned previously in this Section, for the case study a real-life device was used instead of a dummy one and so it was used regularly in the meantime between experiment two and three. The fourth and final memory dump was collected ten minutes after the previous one, after the device was rebooted and its results are presented in Fig. 7.

Our results suggest that the amount of retrieved data does not vary significantly across the experiments. Specifically, when the battery was removed the results are almost identical (see Fig. 4 and 5). Furthermore, in the third experiment the amount of retrieved data is the lowest among the four (see 6). As this is the experiment where the device was used for a day without rebooting, this can be explained from the volatile nature of memory, as some parts of it need to be overwritten throughout the day. Both experiments 2 and 4 include the removal of device's battery in an attempt to remove the power of the RAM and clear its contents. In contrast to what one would expect, Fig. 5 and 7 show a high volume of retrieved data. This can be justified from the way the operating system and its applications work. As discussed in section 2.1, instant messaging applications run in the background as they need to receive calls and messages in real-time, therefore, their data are loaded in memory after each reboot. Regarding the messages' time of origin, it is worth mentioning that during the experiments not only recent but also older messages (up to 16 months old) were retrieved.

V. Conclusion and Future Work

Nowadays smartphones, especially Android devices, have become pervasive and are therefore part of our everyday life. As a result, they have become a great source of evidence in digital investigations. One popular communication vector with smartphones is instant messaging applications, whose usage has overcome the SMS since 2012. Therefore, the forensic analysis of their data has become more than necessary. In most cases, this analysis is based on the application's databases, which are found in the device's internal memory. However, these databases can be tampered or even erased (intentionally or not). In this case the use of memory forensics is vital, as it provides access to a different source of evidence and the ability of retrieving deleted or tampered evidence. Moreover, the increase of the memory size on smartphone devices along with the fact that most of the users do not reboot their devices makes the memory a source of valuable evidence, as nowadays evidence tends to reside in RAM without being replaced.

This paper presented a methodology for the retrieval of instant messaging data from the memory of Android devices. It also provided a case study in which the methodology is used against four scenarios that serve the purpose of studying the behavior of such data on the memory. Our results reveal that a considerable volume of data can be retrieved from the memory of the device. It is worth noting that the messages that are recoverable are not only the recent ones, but also messages dated a few months before the memory dump.

Our results are limited to Linux-based smartphones, such as Android devices. Unfortunately, the methodology cannot be performed on iOS devices, until their kernel source code becomes available and therefore their memory can be extracted using a LKM. However, Android currently has the biggest user base and can also be found in devices other than smartphones in which instant messaging apps can be installed, such as tablets, smart watches, etc. In addition, the acquisition of the memory in order to replicate of our work is a complicated and difficult process and therefore demands technical skills. It also demands a rooted device, as well as having the right kernel source code, which might not always be possible. Also, as the applications are frequently updated, the patterns might need to be also frequently updated.

For future work we plan to update our regular expressions so they match the data patterns of the latest versions of the applications. Additionally, we plan to extend our work by including other popular instant messaging applications. Finally, we plan to examine further correlation and statistical analysis on the retrieved data in order to help the analyst identify the most important data.

APPENDIX A

This section presents the data that can be extracted from memory for each instant messaging app that is in the scope of our work and the regexes that were used to collect them.

WhatsApp. Our work suggests that the following data are stored by this app in the RAM:

- **keyremoteid**: Stores the senders id, as follows: a) in a 1-1 chat includes the senders number with the suffix @s.whatsapp.net, b) in a group chat includes the group's id, which is created by the number of the groups admin (i.e., the user that created the group) and the prefix @g.us.
- **timestamp:** stores the time and date that the message was read or send
- **keyid:** stores the message id
- **sendfromme:** Boolean value that is 1 when the device has sent the message
- **message:** stores the body of the message
- **remotesource:** stores the senders id in a group chat (similarly to keyremoteid), otherwise defaults to -1
- **timereceived**: stores the date and time that the message was received
- **receipt_server_timestamp:** stores the time and date that the messages was read. This field will have a value only in the case of an outbound message

In the case of *1-1 chats* the following regular expression is used in order to parse *incoming messages*:

```
\d{12}@s.whatsapp.net\d{10}[-]\d [\x00-
\xFF]{2,300}\x01[\x00-\xFF]{5}\x30\x01 [\x00-
\xFF]{5}\xFF{2}
```

Specifically, the regex identifies the following data:

- a 12-digit integer and the suffix @s.whatsapp.net, which constitutes the *keyremoteid* field
- a ten digit interger followed by a '-' and a one-digit integer that form the message's *keyid*
- 2 to 300 hex numbers that hold the message's body (irrespective of the language that is used)
- Six hex numbers starting with 0x01, which constitutes the *timestamp* field
- a constant hex number 0x30
- six hex numbers starting with 0x01 that stores the *timereceived* field
- 0xFFFF which terminates the message's string

In the case of *1-1 chats* the following regular expression is used to parse *outgoing messages*:

```
\d{12}@s.whatsapp.net\d{10}[-]\d [\x00-
\xFF]{2,300}\x01[\x00-\xFF]{5}\x30\x01 [\x00-
\xFF]{5}\xFF\x01 [\x00-\xFF]{5}
```

This regex is similar to the one that parses *incoming message* strings, differentiating after the timestamp field as follows:

- 0xFF that works as a separator
- Six hex numbers starting with 0x01 that store the *receipt_server_timestamp* field

In the case of *group chats* the following regular expression is used in order to parse *incoming messages*:

```
\d{12}[-]\d{10}@g.us\d{10}[-]\d [\x00-
\xFF]{2,300}\x01[\x00-\xFF] {5}\x30
\d{12}@s.whatsapp.net\x01 [\x00-\xFF]{5}
```

Specifically, the regex identifies the following data:

- a 12-digit integer followed by a '-', a one-digit integer and the suffix @g.us, which constitutes the group's *keyremoteid* field
- a ten digit integer followed by a '-' and a one-digit integer that form the message's *keyid*
- 2 to 300 hex numbers that hold the message's body (irrespective of the language that is used)
- Six hex numbers starting with 0x01, which constitute the *timestamp* field
- a constant hex number 0x30,
- six hex numbers starting with 0x01 that store the *timereceived* field
- a 12 digit integer follow by @s.whatsapp.net, which constitute the *remotesource* field
- Six hex numbers starting with 0x01 that constitute the *timereceived* field

In the case of *group chats* the following regular expression is used in order to parse *outgoing messages*:

```
\d{12}[-]\d{10}@g.us\d{10}[-]\d{1,3}
\x04[\x00-\xFF]{2,300}\x01 [\x00-\xFF]
{5}\x30\x01 [\x00-\xFF]{5}\xFF\x01 [\x00-
\xFF]{5}
```

Similarly to what has been described earlier, this regex is similar to the one that parses *incoming message* strings for *group messages*, with the following differences:

- the 0x04 value resides between the *keyid* field and the message body
- Since this is an outgoing message *remotesource* field does not exist after the *timestamp* and 0x30, instead similarly to the second regex the following fields are: *timereceived* and *receipt_server_timestamp*

Viber. Our work suggests that the following data are stored by this app in the RAM:

- **number:** this field stores the number of the second communication party (where the first communication party is the device that has been acquired)
- **message:** this field includes the message's body
- **date:** stores the receipt timestamp
- **token:** this field stores a unique token for each message

The following regular expression is used to parse Viber messages:

```
[+]\d{12}\x01[\x00-\xFF]{5}\x02 [\x00-
\xFF]{2,300}\x74\x65\x78\x74
```

Specifically, the regex identifies the following data:

- the character '+' followed by a 12-digit integer that form the *number* field
- Six hex numbers starting with 0x01, which constitutes the *date* field
- a constant hex number 0x02
- 2 to 300 hex numbers that hold the message's body and the token field, which have to be split and decoded
- the hex value of the string "text" that terminates the string of each message

Facebook Messenger. Our work suggests that the following data are stored by this app in the RAM:

- **Timestamp:** this field stores the date and time that the message was received
- **Recipientid:** includes the receiver's id, which is present only in 1-1 messages
- **Senderid:** includes the sender's id
- **Recipientname:** includes the receiver's name
- **Sendername:** includes the sender's name
- **Message:** this field stores the body of the message
- **Groupid:** includes the group's id (only present in group chats)
- **Groupparticipants:** this field includes the names of the

group's participants (only present in group chats)

The following regular expression is used to parse *1-1 Facebook Messenger messages*:

```
mid\x2E\d{13}:[0-9a-z]+ONE_TO_ONE:\d+
:\d+[\x00-\xFF]{8}[\x00-\xFF]{2,300}
[{]"email":"\d+@facebook.com","user_key":"FACE
BOOK:\d+","name":"[\x00-\xFF]{2,50}"[}]
```

The structure of this regex is analysed in Section 4.

To parse *group messages* in Facebook Messenger two regexes are needed, namely: a) one that recovers the data of the message itself and b) one that recovers the app specific actions of the group members (e.g. add participant, leave the group, name the group, etc.).

The following regular expression is used to parse *group messages* created by Facebook Messenger:

```
mid\x2E\d{13}:[0-9a-z]+GROUP:\d+ [\x00-
\xFF]{8}[\x00\xFF]{2,300}[{]"email":
"\d+@facebook.com","user_key":"FACEBOOK:\d+","
name":"[\x00-\xFF]{2,50}"[}]
```

This regex is similar to the regular expression that is used to parse *1-1 Facebook Messenger messages*, differentiating with the substring "GROUP" instead of "ONE_TO_ONE".

Finally, the following regular expression is use to parse *group actions*:

```
m_action:\d{13}\x30{6}GROUP:\d+[\x00\xFF]{8}[\
x00\xFF]{1,300}[{]"email":"\d+@facebook.com","
user_key":"FACEBOOK:\d+","name    ":"[\x00-
\xFF]{2,50}"[}]
```

This regular expression is similar to the two previous regexes and identifies the following data:

- it starts with the substring "m_action" instead of "mid"
- 13 digits that store the timestamp similarly to the previous regexes
- 6 hex values
- the rest of the regex is similar to the previous regex

Timezone. In all three instant messaging apps the following regex was used in order to recover the device's timezone, namely:

```
setTimeZone+[=]+[a-zA-Z/]{2,}
```

REFERENCES

[1] Ericsson Mobility Report, June 2016

[2] 2016 World Population Data Sheet, Population Reference Bureau, August 2016

[3] Pearson, C., & Hussain, Z. (2015). Smartphone use, addiction, narcissism, and personality: A mixed methods investigation. International Journal of Cyber Behavior, Psychology and Learning (IJCBPL), 5(1), 17-32.

[4] Grover, J. (2013). Android forensics: Automated data collection and reporting from a mobile device. Digital Investigation, 10, S12-S20.

[5] Informa, 2013: http://www.informa.com/

[6] Gartner , Market Share Alert: Preliminary, Mobile Phones, Worldwide, 2Q16

[7] Mylonas, A., Meletiadis, V., Tsoumas, B., Mitrou, L., & Gritzalis, D. (2012, June). Smartphone forensics: A proactive investigation scheme for evidence acquisition. In IFIP International Information Security Conference (pp. 249-260). Springer Berlin Heidelberg.

[8] Mylonas, A., Meletiadis, V., Mitrou, L., & Gritzalis, D. (2013). Smartphone sensor data as digital evidence. Computers & Security, 38, 51-75.

[9] Ligh, M. H., Case, A., Levy, J., & Walters, A. (2014). The art of memory forensics: detecting malware and threats in windows, linux, and mac memory. John Wiley & Sons.

[10] Barmpatsalou, K., Damopoulos, D., Kambourakis, G., & Katos, V. (2013). A critical review of 7 years of Mobile Device Forensics. Digital Investigation, 10(4), 323-349.

[11] Android Developers: https://developer.android.com/

[12] Singh, A., Agrawal, A. V., & Kanukotla, A. (2016, January). A method to improve application launch performance in Android devices. In Internet of Things and Applications (IOTA), International Conference on (pp. 112-115). IEEE.

[13] Nomura, S., Nakamura, Y., Sakamoto, H., Hamanaka, S., & Yamaguchi, S. (2014, October). Improving choice of processes to terminate in Android OS. In 2014 IEEE 3rd Global Conference on Consumer Electronics (GCCE) (pp. 624-625). IEEE.

[14] Al Mutawa, N., Baggili, I., & Marrington, A. (2012). Forensic analysis of social networking applications on mobile devices. Digital Investigation, 9, S24-S33.

[15] J. Lessard & G. Kessler (2010), Android Forensics: Simplifying Cell Phone Examinations, Purdue University

[16] The Volatility Foundation: http://www.volatilityfoundation.org/

[17] Joe Sylve , Andrew Case , Lodovico Marziale , Golden G. Richard (2011), Acquisition and analy-sis of volatile memory from android devices, Digital Investigation

[18] Girault, E. (2010). Volatilitux: Physical memory analysis of Linux systems.

[19] Github fmem: https://github.com/NateBrune/fmem

[20] Sylve, J. (2012, January). Lime-linux memory extractor. In Proceedings of the 7th ShmooCon conference.

[21] Müller, T., & Spreitzenbarth, M. (2013, June). Frost. In International Conference on Applied Cryptography and Network Security (pp. 373-388). Springer Berlin Heidelberg.

[22] CyanogenMod: http://cyanogenmod.org/

[23] Most popular messaging apps 2016, Statista April 2016: https://www.statista.com