Forensic Science International: Digital Investigation 40 (2022) 301342

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

DFRWS 2022 EU - Selected Papers of the Ninth Annual DFRWS Europe Conference

Extraction and analysis of retrievable memory artifacts from Windows Telegram Desktop application

Pedro Fernández-Álvarez, Ricardo J. Rodríguez*

Dpto. de Informática e Ingeniería de Sistemas, Universidad de Zaragoza, Spain

ARTICLE INFO

Article history:

Keywords:
Digital forensics
Memory forensics
Instant messaging
Telegram Desktop
Windows

ABSTRACT

Instant messaging applications have become a very common way of communicating, and today there are many applications of this type. The forensic analysis of these applications can help provide essential clues to solve or clarify a possible crime. This type of applications generally store their data in a secure way or transmit it through encrypted channels and thus, the forensic analysis of memory takes on special relevance to analyze them. Following a three-phase forensic analysis methodology, this work has developed a forensic analysis environment for instant messaging applications composed of two tools. One of the tools is responsible for extracting the content of a process that runs on a Windows system, while the other focuses on studying the information present in the process memory of an instant messaging application. This second tool can be easily adapted and extended to provide analysis support for any instant messaging application. As a case study, we focus on the Telegram application for Windows systems called `Telegram Desktop`. Adapting these tools to this application, their joint use allows obtaining forensic artifacts of interest for an investigation, such as user contacts or the content of conversations that have taken place, among others, even when the application is blocked. Obtaining these data is of great help for a forensic analyst, since the analysis of these data can be vital to clarify the events that occurred in some type of criminal act. Both tools are open source under the GNU/GPLv3 license to promote their use and extensibility to applications of other instant messaging services.

1. Introduction

Instant messaging (IM) applications allow us to communicate quickly and easily. Today, a large part of society makes use of these applications to have text, audio, or even video conversations on a daily basis (Statista Research Department, 2020). Unfortunately, the misuse of IM apps allow cybercriminals to utilize them for malicious purposes (such as harassment, extortion, or fraud; to name a few) (Mohtasebi and Dehghantanha, 2011). Likewise, they can be useful to clarify criminal cases. In both cases, the forensic analysis of IM applications can help provide essential clues to solve or clarify a possible crime.

The digital artifacts of interest in IM applications are, among others, the sent/received messages or the contacts. Some of these artifacts are securely stored in a database using encryption, which protects the confidentiality of data against attackers who have physical access to the devices. Likewise, IM application communications are typically established using end-to-end encryption, which also ensures data confidentiality against network attacks. Therefore, digital storage forensics fails to provide evidence. To overcome this, the running IM application can be used to obtain evidence, as the encrypted data must be decrypted when the application needs to use it. However, the application could be blocked, preventing access and making recently viewed data inaccessible. Furthermore, the data deleted by the user is not accessible through the GUI. Hence, memory forensics becomes especially interesting for working with IM applications to get even more evidence.

With the exponential growth of IM applications and the emergence of smartphones, many researchers have shown big interest in investigating IM applications on mobile platforms to acquire evidences for forensic analysis (Cichonski et al., 2012). However, little attention has been paid to desktop platforms.

To fill this gap, in this paper we focus on the forensic analysis of IM applications for desktop platforms, and, in particular, for the Windows operating system (Windows, for short), which at the time of this writing is the predominant operating system on the market

* Corresponding author.
E-mail address: rjrodriguez@unizar.es (R.J. Rodríguez).

https://doi.org/10.1016/j.fsidi.2022.301342

worldwide, with almost a 75% share (GlobalStats, 2021). In this regard, we follow an analysis methodology on which we then build an analysis environment for IM applications that can be easily extended for forensic practitioners and researchers to improve their analysis capabilities. To show its practicability, we study in detail the memory artifacts that can be extracted from a `Telegram Desktop` application running on Windows, one of the 5 most popular IM apps as of July 2021, according to the number of monthly active users (Statista Research Department, 2021).

Consequently, this paper has the following contributions. First, we provide an analysis environment for IM applications based on the three-phase analysis methodology commonly used in digital forensics (extraction, analysis, and reporting). In addition, we also develop and publish other tools that are necessary for our analysis environment. Second, we consider the `Telegram Desktop` application for Windows as case study, providing a detailed study of the memory artifacts that we can obtain. As a by-product of our research, we also publish the tools built on top of our analysis environment to analyze the `Telegram Desktop` application for Windows. All the software we developed is publicly and freely available under the GNU/GPLv3 license to foster research in this area.

The rest of this paper is organized as follows. Section 2 reviews related works covering IM forensics which also analyze RAM in chronological order. Section 3 provides some background on virtual memory, memory mapping of Windows processes, and Telegram. Section 4 describes the analysis methodology we followed and the tools we developed to extract memory artifacts from IM applications. These analysis tools are subsequently evaluated with the `Telegram Desktop` application in Section 5. Finally, Section 6 concludes this paper.

2. Related work

Yusoff et al. (2017) analyzed the information that three IM applications (in particular, `Telegram`, `OpenWapp`, and `Line`) store in internal storage and in RAM of smartphones running on Firefox OS. As for the internal storage, they only retrieved the user's phone number in the `OpenWapp` application. However, they successfully identified the user's phone number in the three IM applications, and the user messages (only in the `OpenWapp` application) when they analyzed the contents of RAM.

Nisioti et al. (2017) focused on data stored in RAM corresponding to the IM applications `Messenger`, `WhatsApp`, and `Viber` running on Android. They retrieved messages from RAM even from conversations that occurred 16 months before the analysis is done.

In (Gregorio et al., 2018), the authors investigated the data related to `Telegram Desktop` application stored on the disk of macOS computers. The authors first found the location where the application's local database was stored, which was encrypted. However, they successfully retrieved the messages through another type of action. In particular, they copied the `Telegram Desktop` application and the (encrypted) data from a forensic disk image to a forensic environment and then ran the application in that controlled environment. After that, the application loaded its database and decrypted the messages. Hence, they were able to see them in the application's user interface. A limitation of this study is that they do not provide details about the configuration of the application that allows them to view the messages in this way.

Thantilage and Le Khac (2019) analyzed data located in RAM on Windows and macOS platforms, particularly focused on `Skype`, `WhatsApp`, `iMessage` (macOS only), `Viber`, and `Messenger` applications. The authors were able to retrieve data on conversations for all the IM applications and operating systems they considered in their work.

In (Barradas et al., 2019), the authors studied memory artifacts stored in RAM on Windows, macOS, and Ubuntu platforms. However, they analyzed IM web applications rather than desktop applications. In particular, they considered the IM services of `Trillian`, `Messenger`, `Hangouts`, `Skype`, `WhatsApp`, and `Telegram`, in different web browsers: `Google Chrome`, `Mozilla Firefox`, `Opera`, `Microsoft Edge`, and `Safari`. The authors found data corresponding to chats for all operating systems and web browsers for `Messenger` and `Skype` (in this case, except `Mozilla Firefox` on Windows). However, they did not find this information for any operating system or web browser on the other IM platforms. In addition, they also studied the data stored in RAM on Android smartphones related to `Viber`, `Signal`, and the same applications mentioned above (except `Skype`). In this case, they found data related to chats in `Messenger`, `WhatsApp`, `Viber` and `Hangouts`, but not in the other three IM applications evaluated.

Kazim et al. (2019) studied data related to the IM application `Hangouts` stored in RAM of Windows computers. Although they found information about conversations, they could not reconstruct them chronologically because they did not find the dates on which the messages found were sent.

Al-Rawashdeh et al. (2020) investigated the `Kik` application for Android smartphones. The authors retrieved information related to chats and contacts by analyzing the contents present both in the RAM and in the internal storage of the phone.

The above related works show that relevant artifacts can be found in RAM of IM applications. However, to the best of our knowledge, at the time of this writing we are not aware of any research related to what data the Windows `Telegram Desktop` application stores in RAM and what data can be retrieved. For this reason, we chose this application as a case study. This work is complementary to the aforementioned works, since we have followed a general analysis methodology that we applied to the field of memory forensics in IM applications and we have provided an analysis environment that allows the analysis of any IM application, after a careful reverse engineering process and an effort of software development.

Our work also demonstrates that encrypted data of any application must be decrypted when the app needs to use it. Therefore, memory forensics becomes especially interesting for working with IM applications that store their data securely or transmit it through encrypted channels.

3. Background

This section details key concepts of the virtual memory manager and Windows process memory mapping that are relevant to better understand this paper, as well as the multiplatform IM service Telegram.

3.1. Paging and the virtual memory manager

On Windows, each process has its own private virtual address space (Ligh et al., 2014), which is a linear memory space (i.e., with contiguous addresses) divided into blocks of the same length, called pages. A page of a virtual address space of a process can be in different states (Microsoft Docs, 2018b) (*free*, *reserved*, or *committed*). Page sizes can be small or large. The small page size is 4 KiB, while the large page size ranges from 2 MiB to 4 MiB (on x86 and x64 architectures and on ARM, respectively) (Yosifovich et al., 2017).

Operating systems that have support for virtual memory need to somehow maintain the relationship between virtual memory and physical memory. In Windows, this is done through the page table entries (PTE), which map a process virtual memory page to a physical memory page.

The virtual memory manager is a separate Windows process, primarily responsible for managing physical memory usage. To do this, it tracks each page of physical memory and ensures that when a thread in the context of a process reads/writes to addresses in its virtual memory space, it refers to the correct physical addresses thanks to the PTEs (Microsoft Docs, 2018a).

This process keeps track of the virtual addresses that are reserved or used in the process address space through the Virtual Address Descriptor (VAD) tree (Yosifovich et al., 2017). The VAD tree is a self-balancing binary tree that has a root (named *Vadroot*) and leafs (named *Vadnodes*) (Dolan-Gavitt, 2007).

Additionally, the virtual memory manager is also responsible for paging files, which are the pages that are routinely saved to disk when not in use or when the memory required by the running process exceeds the available physical memory. These pagefiles are later retrieved by returning them to physical memory at the request of the owning thread (that is, when the thread accesses some virtual memory address of the page saved to disk). Also, since pages are only loaded into memory on demand (called *demand paging*), not all content from an application program or shared library will be continuously resident in memory. Recent research has quantified the number of resident pages on a Windows 10 system at 80% and 20% of the total content for application programs and dynamic shared libraries, respectively (Martín-Pérez and Rodríguez, 2021).

3.2. The Windows process memory mapping

On Windows, any executable, shared dynamic library, or driver file that is loaded as part of the kernel or a user-mode process is named *image*, while the file (as in disk) is named *image file*. Internally, an image and a process are represented by a module (Microsoft Docs, 2017). In what follows, we adhere to this terminology.

A Windows image file follows the Portable Executable (PE) format (Microsoft Corporation, 2021), which is a data structure used in 32-bit and 64-bit versions of Windows that encapsulates the information necessary for the Windows PE loader to manage the executable code.

When an image file runs, the Windows PE loader creates a virtual address space for the process and maps the image file from disk to the process address space. It tries to load the image at its preferred base address (defined in a PE field) and maps the PE sections into memory. During this mapping, multiple pages (typically 4 KiB pages) are allocated to accommodate the content of the image file. In addition to the memory required to hold the PE sections, more memory is allocated to hold the process stack and heap. In addition, external dynamic shared libraries on which the program depends are also loaded into the same memory space in a similar way, allocating memory appropriately.

3.3. Telegram

Telegram is a multiplatform IM service with client-server architecture. Currently, there are official Telegram clients for the most used operating systems both at the desktop (Windows, GNU/Linux, macOS) and mobile level (Android, iOS). Additionally, Telegram can also be used via web browser.

Telegram has a public API that allows access to its functionalities, which allows unofficial clients developed by third parties to also exist, in addition to official clients. Telegram's official clients are open source, unlike the code that runs on Telegram's servers.

To use Telegram, it is necessary to have a user account that is uniquely associated with a mobile phone number. Telegram allows the simultaneous use of 3 user accounts. A Telegram account can also be associated with a public username, which can be chosen by the owner of the account and allows users of the platform to find other users simply by searching by username, without having to know their associated phone numbers.

In Telegram, in addition to textual conversations, end-to-end encrypted voice and video calls can be made. Regarding text conversations, there are 3 types: *one-on-one conversations*, *groups*, and *channels*.

One-on-one conversations are conversations in which only two users participate. These conversations can be *regular chats* (which are stored in the Telegram's servers) or *secret chats* (which are conversations held through an end-to-end encrypted communication channel). Furthermore, Telegram allows one-on-one conversations with a special type of user, known as bots, who are computer programs rather human beings.

Group conversations are conversations in which all the users involved can send messages and read the messages that other members of the group have sent. There are two types of groups: *normal groups*, which are always private and have a maximum of 200 members; and *supergroups*, which can be private or public and can have up to 200,000 members.

Channels can be seen as a special type of group, in which only certain users (called administrators or publishers) can send messages, while the other users of the channel (called subscribers) can only read messages. A channel can be private or public and can have an unlimited number of subscribers.

The fact that a group or channel is public means that any Telegram user can join the conversation. On the other hand, if the conversation is private, it is possible to control who can participate in the conversation. Messages sent in normal conversations, groups and channels travel from the sender to the Telegram servers in an encrypted manner, where they are decrypted and subsequently sent to the receiver or receivers, also in an encrypted manner.

Telegram uses a symmetric encryption scheme called MTProto to send and store these messages on Telegram's servers. As a consequence, Telegram can decrypt them, allowing these conversations to be synchronized between various devices. In contrast, secret conversations have end-to-end encryption, which means that Telegram cannot theoretically decrypt their contents and therefore these conversations cannot be synchronized between different devices. Recent research has highlighted weaknesses in MTProto's key exchange algorithm (Albrecht et al., 2022).

Telegram applications can be typically locked with a password so that the user must enter it when accessing the application. This prevents an unauthorized person who physically seizes a device from being able to access the app without knowing the unlock password.

4. Memory artifact analysis methodology for IM applications

In this section, we first describe the analysis methodology that we followed and then we explain the software tools we created to obtain and analyze memory artifacts from IM applications using this methodology.

4.1. Memory artifact analysis methodology

The memory artifact analysis methodology that we followed has three phases. Fig. 1 outlines the analysis methodology. The first phase (*extraction phase*) is dedicated to obtaining the memory content of the virtual address space of any process into appropriate files for analysis. The second phase (*analysis phase*) deals with the analysis of the data extracted in the previous phase, obtaining representative objects from the dumped files. Finally, the third phase (*reporting phase*) focuses on generating a report containing relevant information on all analyzed memory artifacts, such as the

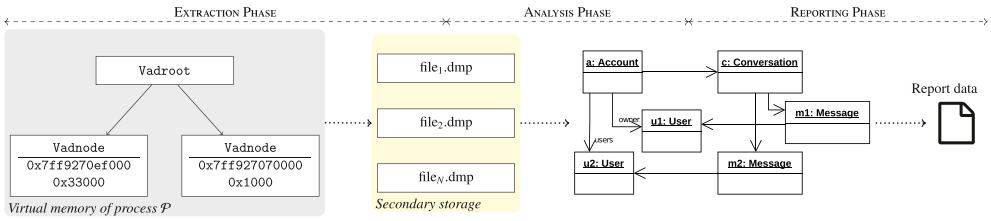

Fig. 1. Memory artifact analysis methodology followed for IM applications.

number of conversations, the number of messages, and other statistics.

Each of these phases is supported by command line tools, facilitating integration into broader analysis pipelines. In particular, we have developed a tool (dubbed `Windows Memory Extractor`) for the extraction phase and another tool (dubbed `IM Artifact Finder`) for the analysis and reporting phases. Furthermore, this latest tool has been designed to support any IM service following the best software engineering practices. Below, we give more details about each of these tools.

4.2. *Extraction phase:* `Windows Memory Extractor`

To analyze the memory of a process or an image, we must first dump it to disk. Multiple tools can be used for this purpose, with `ProcDump` (Microsoft Docs, 2020a) being one of the most popular tools. However, navigating the virtual addresses of the dumped module is only feasible through `WinDbg` (Microsoft Docs, 2020b), as this tool works with a complete knowledge of the internal structures of Windows contained in the dumped data. A process dump can also be created with the `Windows Task Manager`, but as with `ProcDump`, we were only able to match an offset in the dump file to a virtual address using `WinDbg`. The dependency of this tool complicates accessing these addresses programmatically. In addition, it requires installing the symbols corresponding to the version of Windows where the memory dump was acquired, which makes it very difficult to generalize.

Likewise, `Process Hacker` (SourceForge, 2021) is another popular tool for examining the memory contents of all processes running on Windows systems. This tool allows the extraction of individual memory regions of processes, generating dump files with filenames that indicate the region of memory dumped and its size. Obtaining these files for all the memory space of the process allow us to navigate through the virtual addresses of the dumped module, since we can pass from an object A that points to an object B simply by calculating the appropriate file that contains the virtual memory address of B and accessing it with the appropriate offset to locate such object. However, these dump files must be extracted manually using the `Process Hacker` GUI. The inability to obtain these files programatically makes this task time-consuming and totally impossible in real scenarios.

After exploring other existing tools for dumping Windows modules, we couldn't find any tool that would facilitate programmatic navigation through virtual addresses of the dumped modules. Likewise, other memory frameworks such as `Volatility` (Ligh et al., 2014), which is a widely used tool in memory forensics, only provide us with partial content when extracting processes from a full memory dump, as only resident pages (i.e., not swapped pages) can be retrieved. To solve this problem, we developed our own tool, dubbed `Windows Memory Extractor`, as a C++ command line tool to facilitate its deployment in any analysis workflow.

By using Windows APIs that depend on VADs, the tool can dump all memory regions of a process into files, simply by specifying the process identifier of interest. By default, it only considers memory regions that do not have execute permissions. The tool accepts optional arguments to specify the protections of the memory regions to extract. In addition, it also accepts another optional argument to specify the name of a process module when it is only necessary to extract the memory regions corresponding to that module.

`Windows Memory Extractor` saves memory regions in separate files with the extension `dmp` in a directory named «PID_Day-Month-Year_Hour-Minute-Second_UTC» to clearly identify each dump. The nomenclature of these files is similar to `Process Hacker`, indicating the starting virtual address and the size of the memory region dumped, separated by an underscore. In addition, it also creates the `results.txt` file that contains a list of all generated `dmp` files, their SHA-256 hash, and their memory protection. Finally, our tool accepts another optional argument to, given a module, generate a single file containing all the entire virtual address space with the necessary padding between memory regions, when applicable. This file makes it easy to analyze a module as a whole.

This tool allows us to create memory dumps with fine-grained precision, which are necessary to extract memory artifacts from the modules. In our particular case, we have used `Windows Memory Extractor` to generate memory dumps of processes of IM applications running on a Windows system. In such memory dumps, we are particularly interested in the memory regions that do not have execute permissions, because those are the memory regions in which data such as messages, contacts, or user account information is stored.

`Windows Memory Extractor` is open source and released under the GNU/GPLv3 license to promote further memory forensic research (Pedro Fernández-Álvarez and Ricardo J. Rodríguez, 2021b). We have also released a portable version of this tool for use without installation. As a result, a forensic analyst can have the tool in their USB drive and run it from that location, minimizing the potential contamination introduced into the live systems under analysis.

4.3. *Analysis phase:* `IM Artifact Finder`

`IM Artifact Finder` is a Python tool designed as a framework for obtaining memory artifacts from a dump of an IM application process. This tool can be used as a command line tool or as a library, making it easy to use and integrate into other forensic analysis pipelines. `IM Artifact Finder` is designed to work independently of a specific IM platform, operating system, or device. In this sense, the framework can be extended to support any IM application available for different operating systems and devices.

This tool requires two arguments: (1) the path of a directory generated with `Windows Memory Extractor`; and (2) the name of the IM application to which the memory dump corresponds. The second argument (the name of the IM application) is required to let the tool know the structure of the memory artifacts contained in

the dump and load the appropriate artifact finder. For future work, we will investigate methods to detect the IM application automatically (for instance, looking for signatures) and eliminate the need for this argument. Logically, these artifacts depend on each IM application and thus, when extending the framework to a new IM application, a prior reverse engineering task is needed to understand how the application stores its data in memory and when it is present. The same effort is needed to check if the data storage has changed when a new version of the IM application is released. This step becomes mandatory to know how and when to obtain relevant data for analysis, as well as to understand how to interpret it.

When the IM application is proprietary and there is no source code available, the reverse engineering task is error prone and time consuming, making analysis much more difficult. Fortunately, when the IM application is open source, we can analyze its source code and use software engineering best practices to recreate the application design. For instance, we can recreate the Unified Modeling Language (UML) (OMG, 2011) class diagram and UML sequence diagrams to represent the static and dynamic behavior of the system, respectively, but mainly focused on the parts of interest from a forensic point of view. Recall that UML is the *de facto* industry standard for software modeling. Although there are automatic tools to obtain UML diagrams from source code, our empirical tests show that, in general, IM applications are complex software systems and manual analysis is more suited to accurately identify the most relevant elements from a forensic point of view.

We performed an initial automatic analysis to obtain a class diagram of the `Telegram Desktop` source code using `Visual Paradigm` (Visual Paradigm, 2021). This automated process gave us a general idea of how the application is structured. After obtaining the entire class diagram, we identified in it the elements that we considered more relevant for a forensic investigation, and then we manually analyzed the source code of those elements in detail.

The memory artifacts found by the tool are modeled as classes without behavior, since we are only interested in their attributes that will store the information found in memory. The common elements that IM applications have, such as conversations, messages, or user account information, are represented by abstract classes that are concretized based on the specific IM application being analyzed. In addition to storing data, each concrete class is responsible for representing itself in each supported report format, as all of these classes must implement an interface class that contains the definition of representation methods.

Following the best software engineering practices, we have designed our framework to make extension as easy as possible to any IM application. We used the *Abstract Factory* design pattern (Gamma et al., 1995), which is useful for defining an interface for creating sets of related objects. This pattern is used to represent the specific details of the artifacts of each IM application.

This design allows `IM Artifact Finder` to ignore the knowledge of internal details of each application, since each supported IM application requires its own factory to analyze a memory dump properly. In summary, if we want to support a new IM application in `IM Artifact Finder`, we must implement a set of interfaces and the concrete classes that represent the memory artifacts of such IM application. More implementation details on how our framework should be extended are provided in its source code repository (Pedro Fernández-Álvarez and Ricardo J. Rodríguez, 2021a).

This analysis phase is fully automated, so this methodology is scalable as `IM Artifact Finder` will retrieve memory artifacts without manual intervention, regardless of the number of conversations, users, or messages stored in RAM.

4.4. *Reporting phase:* `IM Artifact Finder`

When `IM Artifact Finder` is used as a command line tool, it will generate a report containing information about the artifacts that were found. On the other hand, if it is used as a library, the creation of the report is optional, depending on whether the user wants to create it or not. By default, the generated reports are in JSON format, a commonly used open standard file format for data exchange (International Organization for Standardization, 2017). Currently, this is the only supported format for the report data. We have also designed `IM Artifact Finder` to make adding new report formats as easy as possible. We will expand the support for the CSV format in a future release.

For each Telegram account found, the generated report includes the contents of the retrieved conversations, information about Telegram users related to the account, and details about the account owner. Details about the artifacts that are included in the generated reports are described in Section 5.3. Getting a report in JSON format allows other applications to analyze it and process the information to, for example, search for specific words in conversations, or find all messages sent by a specific user. These reports can also be analyzed by a person and are formatted accordingly to improve readability.

As before, `IM Artifact Finder` is also open source and is released under the GNU/GPLv3 license to facilitate memory forensic research in IM applications (Pedro Fernández-Álvarez and Ricardo J. Rodríguez, 2021a). In the next section we show a particular example of using `IM Artifact Finder` to analyze the `Telegram Desktop` application from a memory forensic point of view.

5. Case study: `Telegram Desktop`

This section is dedicated to the forensic analysis of the `Telegram Desktop` application. We first describe how we extend `IM Artifact Finder` to find memory artifacts present in a `Telegram Desktop` application memory dump. Next, we introduce the experiments carried out and the obtained results. Finally, we present a discussion about the results of the experiments.

5.1. *Extension of* `IM Artifact Finder` *to analyze* `Telegram Desktop` *memory artifacts*

In this paper, we focus on the `Telegram Desktop` application as a case study, which is the official Telegram client for different desktop operating systems (see Section 3.3 for a more detailed explanation of Telegram). In particular, we consider version 2.7.1 of `Telegram Desktop` for Windows systems. Windows is chosen because it is the predominant operating system on the world market at the time of writing, with almost a 75% share (GlobalStats, 2021). In the same way, we choose `Telegram Desktop` as it is one of the 5 most popular IM apps as of July 2021, according to the number of monthly active users (Statista Research Department, 2021), and it has not yet been analyzed from a memory forensics point of view.

First we study the source code of `Telegram Desktop`. Fig. 2 shows an extract of the `Telegram Desktop` UML class diagram of interest for forensic analysis. The *PeerData* class represents a conversation. The *UserData* class represents both a Telegram user and a one-on-one conversation. On the other hand, *ChatData* models a group and *ChannelData* a channel. The *HistoryMessage* class represents a message and the *History* class relates each message to its corresponding conversation.

The *UserData* class has the *_phone* attribute, which is the number of the mobile phone associated with the user's account.

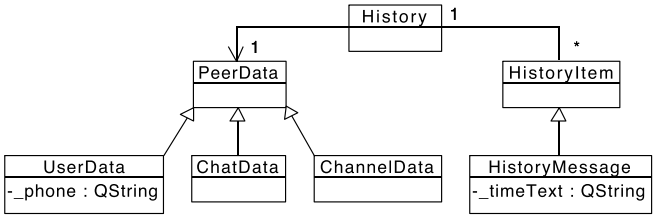

Fig. 2. Extract of the `Telegram Desktop` (version 2.7.1) UML class diagram of interest for forensic analysis.

Therefore, to find *UserData* objects in a memory dump we look for patterns of phone numbers. We use a very general phone number pattern to recognize phone numbers from as many countries as possible. In particular, we look for groups of digits, where each group must have at least 7 digits and no more than 16. The *_phone* attribute is of type *QString* (Qt Documentation, 2021), which is a class that belongs to `Qt` (Qt, 2021), a framework for creating cross-platform applications that run on computers, mobile devices, and embedded systems. Since `Telegram Desktop` uses `Qt`, other relevant information (such as user names or message contents, for instance) is stored in attributes of type *QString*.

Therefore, we have studied in detail how objects of class *QString* are represented in memory to be able to find this information in the analysis phase. In memory, a *QString* object has the following format. The characters of a text follow the UTF-16 format and start at offset 24. From the previous 24 bytes, we have seen that the length of the string is stored at offset 4, occupying 2 bytes and followed by two null bytes, and that the byte at offset 16 is always `0x18` followed by 7 null bytes. We have not been able to discover the meaning of the other 12 bytes. A *QString* string can therefore be retrieved by simply iterating until the UTF-16 null byte that indicates the end of the string is found, or by previously accessing its length at offset 4. Therefore, we can check if the data within the content of an object follows this pattern. If so, it corresponds to a *QString* object, and we can retrieve its text appropriately.

In the particular case of *UserData* objects, we know the virtual address where a phone number begins (VA_1) when we find a pattern of a phone number thanks to the output format provided by `Windows Memory Extractor`. Once we know VA_1, we can subtract 24 bytes from it and get VA_2, which is the virtual address that points to the beginning of the associated *QString* object. Also, the value of VA_2 enables us to relate the phone number to a specific user: looking for VA_2 inside the memory dump and knowing how the *_phone* attribute is stored within *UserData* objects can provide us the virtual address of the *UserData* object associated with that phone number.

Now, exploring the data located around that virtual address we can extract all the contents of that *UserData* object. Then these contents are analyzed and other *QString* objects are sought, as we already know about the analysis of the source code. The *UserData* class has, among others, the following *QString* type attributes: *firstName*, *lastName*, and *username*, as well as the *name* attribute inherited from *PeerData*. Therefore, we look for these *QString* objects by looking at their locations within the *UserData* object found.

In addition, as we have the raw memory data corresponding to several objects of the same type, we continue to analyze how they are structured in memory using the `Telegram Desktop` source code as a reference. We observed that the attributes of the objects are stored in memory in the same order that they are defined in the classes and, if there are inherited attributes, they are stored first. In this way, we distinguish the locations where certain attributes (whose type is not *QString*) are stored within an object, and what their values mean.

This analysis to discover the location of a certain attribute within an object must be done manually the first time to know its location precisely. Once this location is known, the process can be automated since the attribute will always be in the same place within the found objects of the same class.

In `Telegram Desktop`, we can see the time each message was sent, as shown in Fig. 3. This information is stored in the *_timeText* attribute of the *HistoryMessage* class. Considering this, we look for time patterns to be able to find *HistoryMessage* objects present in a memory dump. In the case of *HistoryMessage* objects, we follow the same approach to analyze them as described for *UserData* objects, since the *_timeText* attribute of *HistoryMessage* is also a *QString* object.

Once we find *UserData* and *HistoryMessage* objects in a memory dump, pointers to other objects can be identified within them. Since it is possible to navigate through RAM from one object to another, additional related objects can be obtained. As we analyze the source code of `Telegram Desktop`, we know which pointers we have to follow to get objects that are relevant from a forensic point of view. As a result, we identify the sender of each message, the conversation to which each message belongs, and the account to which each conversation belongs, among other things.

When we reach the object *B* from a pointer that is in object *A*, we have the beginning of object *B*. However, when we reach the *UserData* or *HistoryMessage* objects as described above, we are in some part within the object. This is a small difference when it comes to identifying new objects. Using the source code of `Telegram Desktop`, we know the location in which we are within an object, and we can analyze what information is around it. Likewise, having source code also helps us analyze objects when we are at the beginning of the object.

All this information allow us to create new classes by implementing the appropriate `IM Artifact Finder` interfaces. These new classes allow us to use `IM Artifact Finder` to automatically analyze memory dumps from the `Telegram Desktop` application, since the classes contain the source code needed to obtain the artifacts without manual intervention. Below, we explain the experiments that we conducted and discuss our findings.

5.2. Description of experiments

We define 17 experiments grouped into 7 different categories, whose objective is to evaluate the memory artifacts that we can obtain from the `Telegram Desktop` application. These categories are general categories that can be applied to any IM application. In particular, these categories and the related experiments in each category are:

Accounts. This category encompasses experiments performed to see what memory artifacts can be extracted that are related to user accounts. We define two experiments:

Experiment A_1 «*Single account*». *Get information about the account owner, when there is only one account.*

Experiment A_2 «*Multiple accounts*». *Find information about the owners of the accounts added to the IM application, when there is more than one account.*

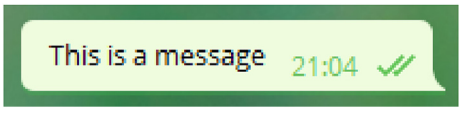

Fig. 3. Example of a message displayed in the `Telegram Desktop` GUI.

Conversations. This category includes experiments performed to see what memory artifacts can be extracted that are related to conversations, such as identifying conversations and reconstructing them. Here, we define five experiments:

Experiment C_1 «*All conversations*». *Identify existing conversations. Note that the goal of this experiment is not to get the content of those conversations, but simply the conversations and their participants.*

Experiment C_2 «*Recently accessed conversations*». *Identify the conversations that the user has recently accessed, whether to send content or simply to read messages.*

Experiment C_3 «*Reconstruction of conversations*». *Find in RAM the information necessary to reconstruct conversations. This experiment focuses only on text messages, not on multimedia messages.*

Experiment C_4 «*Deleted messages*». *Delete received and sent messages, which will disappear from the conversations in the IM application GUI, and then check if they are still stored in RAM.*

Experiment C_5 «*Deleted conversations*». *Check if it is possible to retrieve the content of a conversation from RAM after deleting it.*

Users. This category encompasses experiments performed to see what memory artifacts can be extracted that are related to users, such as contact lists or blocked users. We define three experiments:

Experiment U_1 «*Contact list*». *Get the contact list for each user account.*

Experiment U_2 «*Deleted contacts*». *Delete a contact and then check if their information is still present in RAM.*

Experiment U_3: «*Blocked users*». *Identify if a certain user is blocked.*

Privacy. This category only includes one experiment, which is dedicated to verifying the privacy of the phone number of the IM application users. IM applications can have different privacy policies, allowing a user to choose that everyone, only their contacts, or that no one can see their phone number.

Experiment P_1 «*Phone number privacy*». *In case user A does not share their phone number with user B, check if the phone number of A is present in the content of RAM extracted from the IM application used by B*

Multimedia. This category encompasses three experiments, which are dedicated to verifying what information can be extracted from messages other than text messages sent or received by a user.

Experiment M_1 «*Files*». *Get information about the files received or sent.*

Experiment M_2 «*Shared contacts*». *Find contacts shared with other users.*

Experiment M_3 «*Geographic locations*». *Get the geographic locations received.*

Locking. This category includes two experiments dedicated to verifying what information can be retrieved when the application is locked or password protected.

Experiment L_1 «Locked application». *Check the information available in the memory after locking the application.*

Experiment L_2 «*Unlock password*». *Search in the memory the password required to unlock the IM application.*

Session. This last category includes an experiment, which is dedicated to verifying what information remains in memory once the user logs out from their IM application account.

Experiment S_1 «*Log out*». *Check the contents present in the memory after logging out of the IM application.*

Note that the experiments C_2, C_4, C_5, U_2, L_1, and L_2 cover information that generally cannot be found by simply clicking through the running application. The experiments were carried out on two Windows 10 64-bit virtual machines, one of them with the Education edition (version 20H2) and the other with the Enterprise Evaluation edition (version 1809). As for `Telegram Desktop`, we installed on both virtual machines the version 2.7.1 for 64-bit systems. Also, we use two Telegram accounts to do all the tests.

For each experiment, we interact manually with the `Telegram Desktop` application, performing the actions corresponding to the experiment. Then we follow the analysis methodology explained in Section 4: first, we use `Windows Memory Extractor` to get a memory dump of the `Telegram Desktop` process; and then, we run `IM Artifact Finder` to analyze these dumps and get the report of the extracted results. Regarding the results obtained, we found no differences between the two versions of Windows 10 used in the experiments.

5.3. Experimental results

We describe the results for each experiment below.

***Experiment A_1* «Single account».** In this scenario, we have retrieved the following information about the account owner: their identifier number (unique for each user), full name, phone number, and username.

***Experiment A_2* «Multiple accounts».** In this case, we have correctly identified the number of accounts added to the application. For each account, we have obtained the same information as in the previous experiment (their identifier number, full name, phone number, and username).

***Experiment C_1* «All conversations».** We have successfully identified the conversations that the user has accessed. However, we were unable to find in RAM those that were not accessed. In each conversation found we can distinguish the type of text conversation. In addition, we can obtain the two users participating in a one-on-one conversation and a subset of the participants in the case of groups. The name of the identified groups and channels can also be obtained. Finally, we can know the account to which each conversation belongs in case of multiple accounts.

***Experiment C_2* «Recently accessed conversations».** We can successfully retrieve the conversations that the user has recently accessed, since they are the only ones loaded in RAM.

***Experiment C_3* «Reconstruction of conversations».** We were able to reconstruct the conversations that the user accessed, whether they were one-on-one conversations, groups, or channels. In particular, we have successfully retrieved text messages, message replies, and forwarded messages. However, we have not been able to find the edited messages in any way. Regarding multimedia messages, we have successfully identified the fact that they were sent and their associated text (if any). Multimedia content is outside the scope of this experiment as it is considered in the *Multimedia* category below. When a user accesses a conversation, only the latest messages are loaded into RAM. Afterwards, if the user wants to see the past messages, they will be loaded into RAM on demand (that is, as the user consults the conversation). As a result, the number of retrievable messages is highly dependent on how the user interacts with the application.

***Experiment C_4* «Deleted messages».** With our current methods implemented to analyze memory dumps of a `Telegram Desktop` application, we were unable to retrieve sent or received messages after deleting them. This may be because once they are deleted, the objects that represent these messages are freed and are then no longer associated with other objects.

***Experiment C_5* «Deleted conversations».** Surprisingly, after deleting a conversation, we can get parts of it back. However, the information retrieved is not entirely reliable, as it is not always

correct and is sometimes incomplete. For instance, although it happened very rarely in our experiments, we found that messages of different conversations were retrieved and assembled together in the same conversation. As before, this may be because once they are deleted, the objects that represent these messages are freed and are then no longer associated with other objects, but they are not zeroed.

***Experiment U_1* «Contact list».** We can successfully retrieve those users who share their phone number with the account owner and some users who do not. In case of multiple accounts, we can identify the account to which each user belongs. We have found that to find this information in RAM it is not necessary to interact with the `Telegram Desktop` application after opening it. We can also distinguish whether or not a user is a contact with the account owner. Finally, for each retrieved user we can obtain the following information: identifier, full name, phone number (if shared), username, and if the user is a bot or not.

***Experiment U_2* «Deleted contacts».** In all the tests we did we were able to retrieve contacts after deleting them. Additionally, after being retrieved, the information about them reflects that they are not the user's contacts.

***Experiment U_3* «Blocked users».** The information retrieved from the contacts allows us to determine if a user is blocked or not. However, the user must be involved in a one-on-one conversation previously loaded into RAM in order to detect this.

***Experiment P_1* «Phone number privacy».** We cannot find the phone number of a user if the user does not share their phone number. After conducting this experiment we conclude that Telegram only sends phone number information of users to whom it has permission to view it. For a more detailed approach, we would also need to study the code that runs on the Telegram servers, which is not open source.

***Experiment M_1* «Files».** When a file is attached to a message, we can determine its name and type. Also, we can retrieve the name and type of each transmitted file if multiple files are sent in the same message. We have not managed to retrieve the contents of the file from memory though.

***Experiment M_2* «Shared contacts».** Based on our experiments, we can successfully retrieve the name and phone number of shared contacts.

***Experiment M_3* «Geographic locations».** Although `Telegram Desktop` does not allow sending geographic locations, we focus on reception as a user can receive locations from other users using another different Telegram client. Our experiments show that we can successfully retrieve the latitude and longitude of each transmitted geographic location. In addition, we can also retrieve additional information, such as the name of the place or its address, since they can appear as text accompanying the location.

***Experiment L_1* «Locked application».** In all our tests performed, we have observed that when the application is locked we can retrieve the same information from the memory as when it is not locked. Therefore, locking the application has no effect on the amount of information retrievable from the memory dumps.

***Experiment L_2* «Unlock password».** After unlocking the app, we were able to manually find the password located in the memory. However, we have not been able to find it programatically. Also, we have empirically tested that the password is no longer present in the memory after a short period of time. Hence, we were unable to reliably retrieve the password from the memory dumps. This may be motivated because the application stores the password in memory encoded or encrypted in some way. More research is needed to evaluate this in more depth.

***Experiment S_1* «Log out».** We have successfully retrieved information about accounts, conversations, and users after logging out. However, the number of artifacts retrieved was always less than the number of artifacts retrieved before logging out. This may be motivated because the application destroys the objects (i.e., it frees the memory) as they are no longer needed. Also, the retrieval of memory artifacts after logging out is not completely reliable, as sometimes some of the information retrieved is incorrect or incomplete, in a manner similar to that described in the experiment C_5.

5.4. Discussion and final remarks

The fact that the aforementioned information can be retrieved from the memory of the `Telegram Desktop` application means that, although its local database is encrypted and communications with the Telegram servers are also encrypted, a forensic analyst will be able to obtain valuable artifacts from the memory of a computer where `Telegram Desktop` is running.

A forensic analyst with access to a computer that is on, unlocked and with the installed `Telegram Desktop` application could open it and collect the necessary information. However, this action would severely alter the RAM status compared to its state when the computer was seized. If the forensic analyst accesses the conversations to see their contents before dumping the `Telegram Desktop` process, the information about which conversations the suspect had recently accessed will be lost. In addition, the fact of interacting with the application could cause the loading of information on memory occupied by recently deleted elements, which makes the recovery of these elements not possible.

In the same way, if the forensic analyst interacts with the application when the computer is connected to the Internet, messages could be read that the suspect had not read or received, also notifying their respective senders of these events. This action would also cause the account owner to appear online, making it appear that the suspect is using the app at a given time. The interaction with the `Telegram Desktop` application on a seized computer must be done carefully in a forensically sound manner.

Similarly, a forensic analyst can interact with a suspect's computer where the application is running and is locked or logged out. In these cases, there will be valuable information that cannot be seen in the application's GUI and remains hidden. Memory forensics can also help in this case, as all this information can be extracted from memory (if present).

Knowing the information about the account (or accounts) owner can help identify the person to whom the seized computer belongs. In addition, obtaining users related to a certain account, and differentiating whether they are contacts or not, can provide clues about the people related to a suspect, as well as to identify new people of interest to investigate. On the other hand, there usually must be a reason for one user to block another user. Therefore, being aware of this fact can provide relevant details about the personal relationship between both users.

The possibility of knowing the conversations that users have recently accessed can be relevant to know what they were doing or with whom they were communicating moments before a certain event occurred. Similarly, the possibility of chronologically reconstructing a conversation, knowing for each message the moment it was sent, who sent it, who received it and its content, can be crucial in solving a case. In addition, knowing in which groups and channels a suspect participates (actively or passively) can reveal information about their interests. For instance, finding that the suspect is in a group where illegal material such as child pornography is shared is an important discovery in an investigation.

Likewise, although retrieving conversations after deletion and retrieving artifacts after logging out are not completely reliable processes, a substantial part of the information obtained in our experiments was accurate. This can also be helpful to a forensic

analyst, as this information is not accessible through the `Telegram Desktop` GUI.

As for messages where not only text is sent, knowing the contacts that were shared in a conversation can be helpful, since when a person receives a contact from another user it is natural to think that there is no direct connection between those three individuals. Regarding geographical locations, having information about them is useful to know where the suspects or the victims may have been or to know the places that are related to them. On the other hand, files transmitted via `Telegram Desktop` are not located in the memory as they are stored in secondary storage (by default, in the user's "Downloads" folder, although this location may change). However, knowing the transmitted file names and types is helpful as it guides a forensic analysis of the disk to find those files and understand their contents.

Finally, as shown by the results of experiment L_1 and experiment S_1, we have retrieved memory artifacts after locking or logging out of the `Telegram Desktop` application respectively. As a consequence, valuable information can be retrieved even if it cannot be viewed in the application's GUI when the application is locked or logged out.

6. Conclusions and future work

In this work, we have provided a forensic analysis environment to extract memory artifacts from IM applications. The analysis methodology that we followed covers three phases: extraction, analysis, and reporting. We have developed two tools to achieve these phases. As current tools for process memory acquisition are insufficient for various reasons, our first tool (dubbed `Windows Memory Extractor`) provides process dumps that can be used for further analysis, including virtual address lookup. Our second tool, dubbed `IM Artifact Finder`, provides an extensible framework to analyze those contents and obtain forensically relevant memory artifacts. Both tools are freely and publicly available under the GNU/GPLv3 license, so forensic analysts and law enforcement agencies can use them while the forensic community can contribute by adding the support for different IM applications.

In particular, our tools allowed us to obtain and analyze memory artifacts related to the Windows `Telegram Desktop` application that —if in use— definitely contains a lot of forensically relevant artifacts. Among others, we obtained information about the users of the application and their contacts, reconstructed conversations chronologically, and retrieved certain artifacts even after they were deliberately deleted, after having locked the application, or even after having logged out. As shown, forensic analysis of volatile memory is essential to retrieve valuable information in decrypted form in scenarios where databases or communications are encrypted. In addition, it can also help guide other forensic analyses, such as disk forensics.

An important limitation of this work is that we focus on version 2.7.1 of `Telegram Desktop`. Therefore, `IM Artifact Finder` must be properly maintained and updated to support future versions of `Telegram Desktop`. As future work, our goal is to adapt the tools that we built on top of `IM Artifact Finder` to support the `Telegram Desktop` application on platforms other than Windows, as well as to study other memory artifacts of forensic interest that can be extracted.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Jan-Niclas Hilgert, for their feedback. The research by Ricardo J. Rodríguez was supported in part by the University, Industry and Innovation Department of the Aragonese Government under *Programa de Proyectos Estratégicos de Grupos de Investigación* (DisCo research group, ref. T21-20R) and by the University of Zaragoza and the *Fundación Ibercaja* under grant JIUZ-2020-TIC-08.

References

Al-Rawashdeh, A.M., Al-Sharif, Z.A., Al-Saleh, M.I., Shatnawi, A.S., 2020. A post-mortem forensic approach for the Kik messenger on android. In: 2020 11th International Conference on Information and Communication Systems. ICICS, 079–084.

Albrecht, M.R., Mareková, L., Paterson, K.G., Stepanovs, I., 2022. Four attacks and a proof for Telegram. In: 2022 2022 IEEE Symposium on Security and Privacy (SP) (SP). IEEE Computer Society, Los Alamitos, CA, USA, pp. 223–242.

Barradas, D., Brito, T., Duarte, D., Santos, N., Rodrigues, L., 2019. Forensic analysis of communication records of messaging applications from physical memory. Comput. Secur. 86, 484–497.

Cichonski, P., Millar, T., Grance, T., Scarfone, K., 2012. Computer Security Incident Handling Guide. Techreport SP 800-61 Rev. 2. National Institute of Standards and Technology (NIST). Special Publication (NIST SP).

Dolan-Gavitt, B., 2007. The VAD tree: a process-eye view of physical memory. Digit. Invest. 4, 62–64.

Fernández-Álvarez, Pedro, Rodríguez, Ricardo J., 2021a. IM Artifact Finder v1.0.0. Online; https://github.com/reverseame/instant-messaging-artifact-finder. (Accessed 29 November 2021).

Fernández-Álvarez, Pedro, Rodríguez, Ricardo J., 2021b. Windows Memory Extractor v1.0.8. Online; https://github.com/reverseame/windows-memory-extractor. (Accessed 29 November 2021).

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Longman Publishing Co., Inc., USA.

GlobalStats, 2021. Desktop operating system market share worldwide. Online; https://gs.statcounter.com/os-market-share/desktop/worldwide. (Accessed 22 September 2021).

Gregorio, J., Alarcos, B., Gardel, A., 2018. Forensic analysis of Telegram messenger desktop on macOS. International Journal of Research in Engineering and Science 6, 39–48.

International Organization for Standardization, 2017. ISO/IEC 21778:2017: information technology — the JSON data interchange syntax. Online; https://www.iso.org/standard/71616.html. (Accessed 3 December 2020).

Kazim, A., Almaeeni, F., Ali, S.A., Iqbal, F., Al-Hussaeni, K., 2019. Memory forensics: recovering chat messages and encryption master key. In: 2019 10th International Conference on Information and Communication Systems (ICICS), pp. 58–64.

Ligh, M.H., Case, A., Levy, J., Walter, A., 2014. The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory. John Wiley & Sons, Inc.

Martín-Pérez, M., Rodríguez, R.J., 2021. Quantifying paging on recoverable data from Windows user-space modules. In: Proceedings of the 12th EAI International Conference on Digital Forensics & Cyber Crime. Springer, p. 19.

Microsoft Corporation, 2021. PE format. Online; https://docs.microsoft.com/en-us/windows/win32/debug/pe-format. (Accessed 1 October 2021).

Microsoft Docs, 2017. Modules. Online; https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/modules. (Accessed 15 February 2020).

Microsoft Docs, 2018a. Memory management. Online; https://docs.microsoft.com/en-us/windows/win32/memory/memory-management. (Accessed 15 February 2020).

Microsoft Docs, 2018b. Page state. Online; https://docs.microsoft.com/en-us/windows/win32/memory/page-state. (Accessed 15 February 2020).

Microsoft Docs, 2020a. ProcDump. Online; https://docs.microsoft.com/en-us/sysinternals/downloads/procdump. (Accessed 17 September 2021).

Microsoft Docs, 2020b. WinDbg. Online; https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/debugger-download-tools. (Accessed 17 September 2021).

Mohtasebi, S., Dehghantanha, A., 2011. A mitigation approach to the privacy and malware threats of social network services. In: Snasel, V., Platos, J., El-Qawasmeh, E. (Eds.), Digital Information Processing and Communications. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 448–459.

Nisioti, A., Mylonas, A., Katos, V., Yoo, P.D., Chryssanthou, A., 2017. You can run but you cannot hide from memory: extracting IM evidence of Android apps. In: 2017 IEEE Symposium on Computers and Communications. ISCC), pp. 457–464.

OMG, 2011. Unified Modelling Language: Superstructure. Object Management Group. Version 2.4, formal/11-08-05.

Qt Documentation, 2021. QString class. Online; https://doc.qt.io/qt-5/qstring.html. (Accessed 23 September 2021).

Qt, 2021. Qt framework. Online; https://www.qt.io/. (Accessed 23 September 2021).

SourceForge, 2021. Process hacker. Online; https://processhacker.sourceforge.io/. (Accessed 17 September 2021).

Statista Research Department, 2020. Number of mobile phone messaging app users worldwide from 2018 to 2025. Online; https://www.statista.com/statistics/483255/number-of-mobile-messaging-users-worldwide/. (Accessed 22 September 2021).

Statista Research Department, 2021. Most popular global mobile messaging apps 2021. Online; https://www.statista.com/statistics/258749/most-popular-global-

mobile-messenger-apps/. (Accessed 22 September 2021).
Thantilage, R.D., Le Khac, N.A., 2019. Framework for the retrieval of social media and instant messaging evidence from volatile memory. In: 2019 18th IEEE International Conference on Trust, Security and Privacy in Computing and Communications/13th IEEE International Conference on Big Data Science and Engineering. TrustCom/BigDataSE), pp. 476–482.
Visual Paradigm, 2021. Visual Paradigm. Online; https://www.visual-paradigm.com/. (Accessed 23 November 2021).
Yosifovich, P., Ionescu, A., Russinovich, M.E., Solomon, D.A., 2017. Windows Internals, Part 1: System Architecture, Processes,threads, Memory Management,and More, seventh ed. Microsoft Press, Redmond, WA, USA.
Yusoff, M., Dehghantanha, A., Mahmod, R., 2017. Forensic investigation of social media and instant messaging services in Firefox OS: facebook, twitter, Google+, Telegram, OpenWapp, and line as case studies. In: Choo, K.K.R., Dehghantanha, A. (Eds.), Contemporary Digital Forensic Investigations of Cloud and Mobile Applications. Syngress, pp. 41–62 (chapter 4).