Universidad
Carlos III de Madrid

Bachelor Thesis

Development of an Extensible Forensic Analysis Framework. Application to user-side cloud scenarios

Author: Jorge Rodríguez Canseco
Tutor: Jose María de Fuentes García Romero de Tejada
Co-tutor: Lorena Gonzalez Manzano

Monday 22nd June, 2015

Todas las opiniones aquí expresadas son del autor, y no reflejan necesariamente las opiniones de la Universidad Carlos III de Madrid.

Título: Development of an Extensible Forensic Analysis Framework. Application to user-side cloud scenarios
Autor: Rodríguez Canseco, Jorge
Tutor: Jose María de Fuentes García Romero de Tejada PhD.
Co-Tutor: Lorena Gonzalez Manzano PhD.

EL TRIBUNAL

Presidente:

Vocal:

Secretario:

Realizado el acto de defensa y lectura del Trabajo Fin de Grado el día ..... de ..... de ... en ....., en la Escuela Politécnica Superior de la Universidad Carlos III de Madrid, acuerda otorgarle la CALIFICACIÓN de:

VOCAL

SECRETARIO PRESIDENTE

This page is intentionally left blank.

Agradecimientos

Hay tanta gente a la que me gustaría agradecer tanto, que solo pensar en el asunto me abruma. Pido perdón por adelantado ya que una simple hoja no hace justicia a lo que recibo día a día. Permitidme que me tome la libertad de ordenar los nombres alfabéticamente, y disculpadme si me olvido de alguien. Yo se que estais ahí.

Me gustaría comenzar dando las gracias a mis padres. Sin ellos no estaría donde hoy estoy, y no solo en el sentido biológico. Por su apoyo incondicional en las decisiones que he ido tomando durante mi vida y a sus desvelos por hacer de mi una buena persona. Espero poder devolveros algun dia de alguna forma todo el esfuerzo que habéis puesto en mi.

A mis amigos, aunque no pueda mentaros a todos. Daniel, Felix, Jose, Mario, Miguel, Nacho, Nelson, Sergio(s), Toan, Víctor y al resto de mis pequeños. Muchas gracias por hacer del tiempo libre (y no tan libre) experiencias memorables y por aguantarme tanto en los buenos como en los malos momentos.

A mis compañeros de la Universidad. Álvaro, Irina, Jérôme, Juan, Luis y Ramsés, por estar ahi sufriendo conmigo siempre y por enseñarme lo que significa realmente la universidad. A Manuel, por haber conseguido llegar al final conmigo a pesar del duro camino.

A mi equipo, Abel, Ángel, David(s), Óscar, Pavlo y Victor, por hacerme seguir disfrutando del deporte y por todo lo vivido durante estos cuatro años. Cuando necesitaba desconectar, siempre estabais ahí (menos para el físico, claro).

A mis compañeros de laboratorio, Alex y Joaquin y Jose, porque habéis tenido que aguantar el verme la cara todos los dias y a cada letra que escribía en este proyecto. El trabajo en buena compañía se hace menos duro.

Finalmente, quiero agradecer a Chema y a Lorena su confianza y esfuerzo, tanto en este proyecto como en mi mismo. Sin vosotros nada de esto sería posible.

Abstract

Computer forensic procedures are among the most important methods for nowadays crime investigations, since IT devices are increasingly more and more present in our society and life as main tools for productivity enhancement and social communication.

This improvement in relevance for such procedures is however increasing the efforts in by-passing such methods in an more sophisticated manner. IT forensic analysts must face the complexity of new techniques used by criminals in order to hide their activities and avoid the evidence recovery as much as possible [1]. Some of those include, but are not limited to:

- Covered communication channels and information leakage.
- Obfuscation and information hiding regarding the operations of a malicious agent.
- Operation in volatile media such as RAM and similar procedures with a relatively small digital fingerprint in the system.

Current forensic tools highly depend on the analyst awareness of aforementioned covert channels and evidences existence in order to retrieve them by means of a proactive search methodology. This project documents the creation of Monocle, an open-source extensible framework for automated forensic analysis. Monocle provides automation over the forensic procedure by means of user-created plugins, reducing the complexity of evidence retrieval in target's machine hard disk and memory. The software makes use of external tools such as the Volatility Framework in order to provide extended functionality to the executed plugins. To show the applicability of the proposal Monocle is applied to two user-side cloud storage scenarios – iCloud and Box. This application is further used in order to study such scenarios and their usefulness when targeting cloud storage systems from a forensic point of view.

Resumen

Los procedimientos de cómputo forense se encuentran entre los más importantes para las investigaciones criminales de hoy en dia, puesto que los dispositivos electrónicos se encuentran cada vez más presentes en nuestra sociedad, ya sea como medio para mejorar la productividad personal o como conectores sociales.

Este incremento de relevancia en dichos procedimientos ha provocado no obstante que los esfuerzos por contrarrestarlos se vuelvan más sofisticados. Los analistas forenses deben enfrentarse a la complejidad de las nuevas técnicas empleadas por los criminales para ocultar sus actividades y dificultar la recuperación de pruebas en la medida de lo posible [1]. Algunas de estas técnicas incluyen, pero no están limitadas a:

- Canales de comunicación encubierta y fugas de información por métodos no convencionales.
- Ofuscación y ocultación de informacion relacionada con las actividades maliciosas del agente criminal.
- Operaciones realizadas en memoria RAM y procedimientos de similar naturaleza, los cuales dejan muy poca huella digital en el sistema.

Las herramientas forenses actuales dependen en gran medida de la capacidad del analista para tener en cuenta estos métodos de ocultación, además de conocer la localización y formato de las pruebas potenciales a encontrar, a fin de ser capaz de recuperarlas por medio de una herramienta forense. Este proyecto documenta la creación de Monocle, un framework extensible y de código abierto para la automatización de analisis forense. Monocle dota los procedimientos forenses de automatización gracias a plugins creados y definidos por el usuario, lo que reduce la complejidad a la hora de recuperar informacion tanto del disco como de la memoria del sistema analizado. Monocle hace uso de herramientas externas tales como Volatility Framework a fin de otorgar funcionalidad extendida a los plugins en ejecución. Para demostrar la applicabilidad de la propuesta, Monocle ha sido evaluado en dos escenarios de analisis forense en cloud desde el lado del cliente, iCloud y Box. Esta evaluación permitirá además el estudio de estos escenarios y la aplicabilidad del análisis desde el lado del cliente a la hora de analizar entornos de cloud storage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and objectives

SECTION 1.1

Introduction

Computer forensics has proved to be a key factor in the conduction of criminal investigations and law enforcement as IT devices are increasingly more present in our society [3]. Such procedures are of high importance not only in the elucidation of traditional criminal cases where the new technologies are present as a helper tool for criminals to operate, but also in modern crimes where IT devices are the main platform (i.e. illegal distribution of copyrighted material or online fraud) [4].

In order to conduct such investigations law enforcers make use of a huge variety of forensic tools as to help in the collection of evidences. This is of particular interest because most best forensic practices are standardized, so they can be done in an algorithmic way (e.g. evidence integrity maintenance, evidence classification or data carving). For example, Spanish UNE 71506 describes a forensic analysis methodology [5]. It is thus interesting to automate the collection of such elements in order to speed up the whole process.

Nowadays forensic tools might target a specific source of evidences, or they might compile a set of tools in order to target entire systems with multiple evidence sources at the same time. The latter tools are usually commercial software with a relatively high price due to their complexity. The importance of the forensics is however of such a relevancy that there is an increasing number of open source tools already available.

Additionally, as technologies develop, the countermeasures to such investigation tools improve too. Obfuscation and hiding techniques are more complex as time goes by, and also easier to implement. These techniques include for example the operation over volatile media (i.e. such as RAM) or stenography. Not only this but the fact of technology becoming more complex with time also difficult the forensic procedure, adding new layers of difficulty to traditional analysis. Although a professional forensic analyst must be aware of such techniques during investigations, the amount of possible evasion techniques and covert channels makes it difficult to detect all information attempts *on the fly*, even by using forensically-focused tools.

This document presents MONOCLE, an open source framework for forensic analysis implementing an extensible script-driven system. The software targets relevant sources of information on IT systems, such as volatile memory dumps and disk images. Monocle is used afterwards to study a pair of user-side cloud scenarios in different platforms, namely iCloud [6] and Box [7]. This implies that several modules will be created to test the effectiveness of the framework in those scenarios and to study the scenarios as such.

Monocle works in a procedural manner by means of extensible plugins. Such plugins can be added and removed dynamically in order to adapt Monocle to the requirements of the investigation. Monocle is in charge of managing the evidence sources to analyze and operate over the evidences extracted from such media. Besides, Monocle provides a set of tools which can be used by executed plugins in order to aid the evidence retrieval process. Figure 1 describes Monocle operations regarding its three main stages. The plugin setup (1) stage prepares the environment and resources needed for the analysis to start depending on the requirements of the user. The plugin carving (2) phase operates over the target's memory and disk digital containers in order to retrieve meaningful information regarding the conducted investigation. In the Evidence Management (3) phase, Monocle process each single evidence elicited during phase 2 and performs several operations in order to guarantee integrity and availability of such evidences.

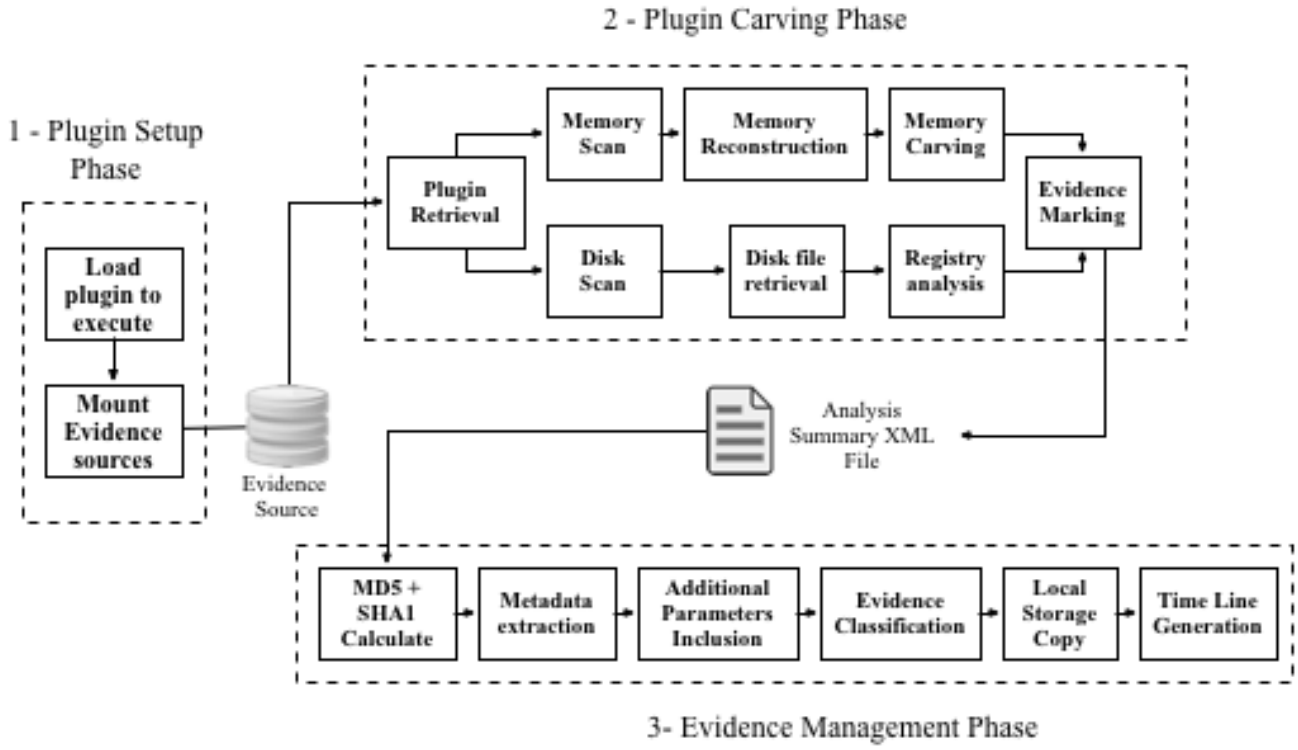

Figure 1: Monocle Functionalities Overview.

SECTION 1.2

Motivation

Most of the current forensic analysis tools (especially commercial ones, such as EnCase [8]) provide an interactive interface to analyse the target system. By doing so the analyst is able to perform a proactive analysis to find where relevant information regarding the conducted case can be found. At the same time, an overview of the overall system state is provided. This can help the analyst to find new sources of information that he might not have appreciated at a first glance.

There is, however, a disadvantage within this paradigm, namely the fact that the user has to manually select which elements to categorize as evidences. Whether this can be useful when conducting a non-deterministic case in which the procedure to retrieve the evidences or their nature are not clear, it is a tedious process when facing some cases in which the traces to be found are quite standardized (e.g. traces regarding the installation of a well-known program). It also requires the analyst to actually know *which* evidences to look for and *where* to find them, as well as *what does it means* the absence of such elements within the system.

Automated scripts might aid the investigation process by decreasing the number of elements the analyst must care of. Although some tools provide a way to extend and automatize user-defined procedures by means of scripts (e.g. [9], [10]), this is not straightforward. Furthermore, it usually implies buying a license for specialized programs.

These issues together with the fact that technologies are becoming more complex as time goes by has also revealed some new challenges for forensic analysis tools, being cloud forensics a relevant one. Such scenarios include a brand new set of problems for law enforcers in order to conduct the analysis.

A research must be performed in order to effectively find methods that can potentially become a good source of evidences when targeting such scenarios. This context would make useful the existence of an open source tool able to conduct forensic procedures applying automated scripts. The existence of such software would avoid organizations and individuals affording the expensive licenses of commercial forensic tools.

SECTION 1.3

Objectives

This section is intended to show the different objectives which are pursued within the present project. Such objectives are extracted from the needs depicted in subsection 1.2, and are considered during the analysis phase of the present project in section 3.

The main objective of the present paper is the creation of a utility framework (Monocle) for the development of automated analysis on third party target machines. Many of the current tools (recall Section II) in the forensic analysis field are either commercial software, or present a limited scope in terms of evidence sources. In this respect, the proposed tool would **target main evidence sources** present on IT devices, e.g. namely volatile memory dumps and disk images (refer to subsubsection 2.1.2).

In addition to that, most of tools do provide a proactive (interactive) search paradigm. This stands for the fact of the tool carving data in different areas, and presenting it to the analyst so as for him to select elements to categorize as evidence. This is a time-consuming process when facing data retrieval of well-known elements, such as program installation data or web access history. By contrast, MONOCLE provides a **script-based analysis** procedure. This implies that the analyst, targeting specific well-known elements, can code several modules. Such plugins can be reused in further investigations with minimum effort, speeding up the analysis process and reducing the analyst manual workload. In this respect, it is necessary for the tool to be **extensible** as to adapt new needs of the analyst. Besides, MONOCLE detects and loads seamlessly new scan plugins (i.e. scripts) without incuring into extra efforts to the user.

Due to this need of extensibility, the proposed software includes several utility tools which can be used in order to speed up the creation of plugins. Such utilities include automatic evidence management, RAM memory reconstruction, registry hives parsing, and visual timeline representation of results. The implementation of a **GUI** for the easy of use and the release of the tool under an **open source license** are also current objectives.

Finally, this project will be employed as a use case for the analysis of user-side cloud scenarios. Cloud scenarios are of particular interest, as they are among the most difficult to analyze nowadays [11]. Problems in such architectures range from jurisdictional issues due to cloud disperse nature [12], to the fact that several users might share the same machine.

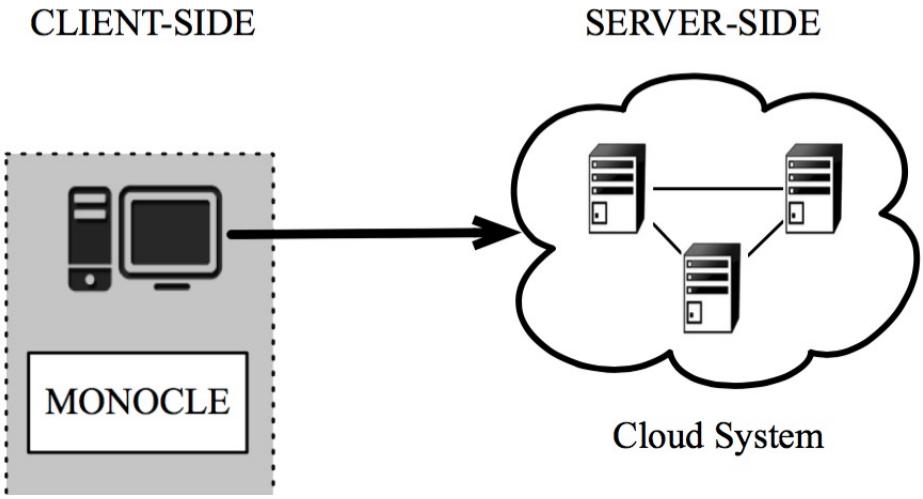

Figure 2: Monocle in the cloud infrastructure.

Given that cloud forensics is hard to analyze from the back end (server-side) point of view, an study on an alternative client-based approach is also an objective within the present project. Order to do so, Monocle addresses computer forensics from the front end (client-side) perspective (see Figure 2) rather than from the remote (server-side) cloud system. This implies that several modules will be created both to test the effectiveness of the framework when targeting those scenarios and to study those scenarios as such.

SECTION 1.4

Organization of the Document

This section has as objective to provide a general overview over the organization of the present document and the contents commented on it. A brief summary of each one is to be provided.

The present document will be divided in six different sections, each of one are described below as follows:

1. **Introduction and objectives**. Being this the present section, it shows an introduction to the topic to be discussed in this project, as well as the motivation behind its implementation and the objectives pursued during the project development.
2. **State of the Art**. This section is targeted to provide some background knowledge to the reader in the topics which are related with the process in some way. This section presents contents in a progressive manner in order to clearly state the concepts from a topdown approach.
3. **Analysis**. This section is intended to discuss the different options in terms of design which can be take in order to accomplish the goals stated in section 1. Different technologies will be discussed, as well as a general schema of the functionality of the system to be developed. Use cases which can happen during the framework usage and the requirements for the system are also described here in order to clarify the design process on the following section.
4. **Design**. This section explains how the underneath system is to be developed according to the requirements and preferences extracted from the previous section. This consists in the design of a detailed component diagram, as well as a class diagram for each of the identified components. The interactions between such components are also to be explained within this section. Use cases are defined also in this section, which will show the underlying behavior of the system while the execution of the use cases.
5. **Software Implementation**. This section explains the different decisions at an implementation level which were taken during the creation of the present system, as well as a brief reasoning on why such decisions were chosen.
6. **Evaluation in cloud scenarios**. The framework evaluation will be done at the same time an analysis of cloud client-based scenarios is performed. This section presents both the study of the different conclusions elicited during the study of such scenarios and the evaluation of the framework with respect to them.

7. **Conclusions and future work**. This section will explain the different conclusions achieved during the development of the present project, as well as future lines of work which can be taken in order to further extend the features and scope of the software.

In addition to these sections, there will be also two appendices to the document, namely the economical analysis and management of the project, which will show the different costs undertaken when developing the present software, and a user manual which will provide the necessary instructions for a user with no experience to interact with the software in an effective way.

The structure of the contents enumerated here will be additionally divided into different sub-levels in order to provide a clear understanding of what is presented.

Chapter 2

State of the Art

SECTION 2.1

Digital Forensics Overview

This section is intended to provide an overview regarding the concept of digital forensics and its implications. Section subsection 2.1 provides a definition on the concept of digital forensics and differentiates the different sources of information available when performing forensic analysis. subsection 2.2 describes the peculiarities of cloud forensics, as well as the issues related with such environments and some literature providing different proposals to solve such issues. subsection 2.3 describes different forensic analysis tools and compares their capabilities with the features proposed for the present project in subsection 1.3. subsection 2.4 and subsection 2.5 describe two different open source forensic tools existing nowadays and their relation with the present project. Finally, subsection 3.3 enumerates the different legal aspects to take into account when dealing with digital forensics and, more specifically, when developing a forensic tool.

2.1.1. The forensic process

Digital forensics techniques consist in the discovering and extraction of digital evidences regarding the usage (i.e. malicious or not) of a digital device, which can provide certain information regarding the actions undertaken by an actor over it [13]. This is a relatively new field of study when compared to other IT research fields, such as cryptography protocols and data exchange security. Incident response techniques related with forensics are nowadays a remarkable source of information regarding digital investigations and further legal actions.

As in traditional crime forensics, digital forensics must follow a strict procedure in order to guarantee the integrity and validity of the discovered evidences. This is not always easy, and it has been a big topic of discussion among forensic investigators to standardize a common framework of action for forensic incident response. [14], [15], [16] are among examples of such a discussion. Spanish UNE 71506 [5] describes a forensic analysis methodology, but once again, different countries usually have different regulations of the matter.

2.1.2. Sources of information

The forensic procedure relays on the analysis of numerous hybrid sources of evidences regarding a digital system. These evidence sources must be secured in a certain manner in order to prevent further manipulation of their original state, which can eventually corrupt or even destroy such evidences. Data integrity is thus one of the most important aspects in digital forensics [17].

Historically, the evidences found on a system were in the physical hard drive of the machine (i.e. registry, files) and the RAM memory, as the computer was disconnected from the internet and attached devices (i.e. destroying potential data such as remote connections). Nowadays procedures prioritize the most volatile sources of data in the recovery order [18].

The usual procedure for evidence acquisition is to make a copy of the data sources in order to have backups in case integrity or the chain of custody are compromised. Such copies might be stored differently depending on the requirements and format usage, but they do have to provide at least a loseless way of storing the evidences. The simplest format to traditionally store such evidence sources was the RAW format [19], in which the whole evidence source is copied byte per byte (i.e. including empty space). Modern approaches however use compression in order to exploit redundant data and to reduce the final need of storage capacity while conducting forensic investigations. These formats are however proprietary formats and do depend on an external entity or company which developed them (e.g. E0, IDIF).

SECTION 2.2

Cloud Forensics Overview

This section is intended to analyze the current state of the art related to cloud forensics, as well as the specific problems related to such environments when conducting forensic analysis over them.

Cloud services are becoming popular among companies and end users. The main reason behind this is the fact that a cloud environment allows the use of hybrid hardware and systems in order to create virtual structures which can be created and modified on user demand. This implies that there is no need to purchase new equipment each time the needs company change. Cloud systems are traditionally structured into three main categories depending on the service model they provide, namely Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) [20].

2.2.1. Cloud Forensics challenges

Although this versatility is quite useful for companies, the nature of a cloud environment makes such to structures include a brand new set of problems for law enforcers in order to conduct the analysis [21]. This is due to the fact that the systems the user is interacting with are usually fragmented among different hardware connected over the network. Cloud computing forensics are thus highly related to network forensics [1], as this information is transmitted through the network.

In addition to this fact, there are also another ones when coming to analyze cloud environments. Some of the most important ones are the fact that cloud service provides usually have different data centers around the world in order to guarantee data availability in case of failure. This implies that the data might be replicated or fragmented, and that such pieces of information are located in different countries (i.e. with different laws regarding digital information management). Another important fact to consider is that physical drives and other hardware will most probably be shared among different end users or entities. Law enforcers will have thus troubles in order to get such evidences as the analysis of this hardware will imply a privacy violation regarding users which are not under investigation. There are differences however in the complexity involved within the analysis for each of the aforementioned services (SaaS, PaaS and SaaS).

2.2.2. User-Side Cloud Scenarios

Although cloud forensics are hard to analyze from a server infrastructure point of view (i.e. as show in the previous point), the problem can be addressed in a different way if we consider the fact that there are end users behind the operations in the cloud. Such end users will be accessing this services by means of personal computers, which are more suitable to traditional computer forensics procedures.

One of the most popular cloud services among users is cloud storage. Some examples of cloud storage providers are Dropbox, iCloud, Google Drive, Box or Mega. Cloud storage platforms usually provide two different ways to access their services. The first one is the access by means of a web application using a web browser. Such application allows the user to remotely interact with stored data within the server. In addition to the web browser, most of cloud storage service providers have a synchronization client program which updates data on the server side based on changes performed at client-side.

This implies that information regarding such services can be retrieved from the suspect drive by analyzing the web browsers and by looking for third-party syncing programs downloaded for such cloud storage services.

Regarding what can be found in memory, one of the most relevant evidences to carve are the URLs associated to the web interface of the cloud services. Memory-mapped structures such as registry keys are also valuable as they might contain data regarding to the web browser (e.g. URLs recently typed and temporal files downloaded or visualized within the browser).

The analysis of the disk image provides more information than memory analysis. Temporal files are stored in each web browser folders. All registry hives are usually present. The most interesting refers to traces left by a newly installed program which are easy to identify. In this context, the installation folder of the program, the different registry keys, which have been modified by the software, and all data the tool is syncing with the cloud service are potential sources of information. System logs are also valuable as they provide a way to track changes within the system.

2.2.3. Current proposed solutions

Previous research contributions have addressed cloud forensics and their peculiarities regarding traditional forensics problems, such as jurisdictional issues and privacy concerns. Most of this literature focuses on the study of the open problems and possible improvements in cloud structures rather than in the actual forensic analysis of cloud systems within the current architectures. Zawoad et al. [11], and Shah et al. [22] discuss the different problems inherent to cloud systems regarding digital forensics. Almulla et al. propose different approaches to provide forensics friendly cloud services [23], but such solutions require cloud service providers to adopt them and do not solve the problem in the short term scope.

The increasing number of users of the SaaS cloud architectures in which data storage is offered as a service has motivated the research on this specific area. Quick et al. [24] provide an extensive listing of traces left on client machines by different cloud storage platforms, such as Dropbox [25], Google Drive [26] Microsoft SkyDrive [27] or ownCloud [28], found by means of different forensic tools. Such studies reveal the utility of client-side forensics in which they call Storage as a Service (StaaS) platforms.

SECTION 2.3

Forensic Analysis Tools

Due to the fact that forensic procedures comprehend a notably number of repetitive and tedious tasks, there are plenty of forensic tools available in order to simplify and speed up this process. This section is intended to provide an overview to the current state of the art regarding automatic analysis tools, as well as a brief description for the most famous ones.

Most commercial tools provide a wide set of procedures in order to recover data evidences, such as data carving [29] and memory data analysis [30]. Those tools are implemented in a generic way in order to allow the flexibility needed when conducting forensic investigations (i.e. as there are a wide number of interpretations an analyst might give to an evidence element, usually depending on the context of the conducted case).

One of the most famous tool is probably EnCase Forensic, developed by Guidance Software [8]. This software provides a high number of analysis elements, such as hard drives, removable media, automatic generation of reports. This software has become almost an standard among forensic analysts as a high valuable tool for conducting such an investigations. In the presented project, we decided not to use this tool as it is a privative software (i.e. a license is worth currently around $3000).

Another interesting tool commonly used by forensic analysts is the forensic toolkit (FTK) from AccessData. This tool provides a similar set of tools as EnCase, and it provides visualization of data in real time, multiple source image detection and automatic password recovery from a big set of applications, among others. This is however as in the case of EnCase, a privative software, whose licensing is worth around $4000.

The Volatility Framework, from Volatility Foundation [31], is also an interesting project to comment here. Volatility was initially a framework to perform analysis over memory dumps on Windows machines. There has been however an increasing number of targets due to the contributions of the community (i.e. and the fact that anyone can create its own profiles in order to target an specific system), and now Volatility targetes also some versions of Linux and Mac OSX.

The Volatility project has an enormous advantage over the two aforementioned ones, which is the fact that it is an open source project. This implies that there is a big community behind the project development, which actively contributes to its enhancement. Due to the open source nature of this software, it has been decided to include its technology as part of

the present software.

There are other several tools which target specific fields of the forensic analysis, such as The Sleuth Kit (i.e. which also provides useful tools for forensic analysis) and Exiftool (i.e. analyzes file metadata) among others. The incorporation of this tools in the present software was out of the scope of this project due to time limitations and the complexity involved.

2.3.1. Forensic Analysis Tools Comparison

This subsection is targeted to show the differences between currently existing forensic analysis tools in a quick and concise way.

There is a wide variety of ways to classify the different tools available nowadays, such as which systems do they target, or whether they are privative software or not. In order to make this classification as appropriate as possible for the scope of the present project, the classification of the given tools has been done regarding their accessibility parameters and their features.

The first classification attending the accessibility parameters is to be done according tot he following parameters:

- **Interactivity / Proactive analysis**. Whether the tool allows the analyst to operate over the results and to perform different analysis over the image in a non-deterministic way. This is the opposite to a scripting deterministic analysis where the tool has a predefined set of operations to run and then exits.
- **Scripting Capability**. Whether the tool has the capability of execute user-defined scripts which will run different analysis or procedures in a sequential way without the user interaction.
- **Extensibility**. Whether the original functionality of the system (i.e. its capabilities) can be enhanced by means of tools provided by the system itself.
- **Extensibility type**. How extensibility is performed within the given application if the application allows for an extensible behavior.

The following table depicts the classification of different well-known tools in the computer forensics field according to the criteria described above.
asdasdasd

Forensic Tools Comparison				
Software	**Interactivity**	**Scripting**	**Extensibility**	**Extensibility Type**
Encase	Yes	Yes	Yes	EnScript. A scripting language automating EnCase tools
FTK	Yes	No	No	–
IEF	Yes	No	No	–
Volatility	No	Yes	Yes	Volatility Framework plugin system. Allows to create new analysis within the Volatility Framework structures.
TSK	Yes	Yes	Yes	Sleuth Kit plugin framework. Allows the automation of the different stages of TSK
ProDiscover v9	No	No	No	–
Bulk Extractor	No	No	No	–

Table 2: Forensic Tools Comparison

- **Reference**:
  - **FTKv4**: Forensic Toolkit, version 4
  - **TSK**: The Sleuth Kit, version 4.1.3
  - **IEF**: Internet Evidence Finder Bundle (IEF Standard + IEF Triage)

It is interesting to notice how Volatility is the only tool offering a non-proactive analysis approach with an extensible implementation, whether the others either require the user interaction (interactivity) or do not provide a way to extend the tool functionality. Volatility however only targets memory (refer to Table 3)

Apart from this usability-faced classification, it is also interesting to list the different characteristics of each software listed above. Such classification will be performed according to the following parameters:

- **GUI Interface Available**. Whether the interaction with the tool can be performed by means of a graphical user interface or not.
- **Privative Software**. Whether the software is a free-to-use tool or requires the purchase of a license.
- **Target**. Whether the tool targets Memory or Disk as sources of evidences.
- **Target State**. Whether the system can run on a live system (i.e. running in the same system which is being analyzed) or targets a dead system (i.e. where the system data comes from a memory or disk dump).
- **Target Device**. Whether the tool targets desktop computers, mobile devices or both.

The following table depicts the classification of different well-known tools in the computer forensics field according to the criteria described above.

Forensic Tools Comparison					
Software	**GUI**	**Privative**	**Target**	**Target State**	**Target Device**
Encase	Yes	Yes ($2,995)	Memory, Disk	Live, Dead	Desktop, Mobile
FTK	Yes	Yes ($3,995)	Memory, Disk	Live, Dead	Desktop, Mobile
IEF	Yes	Yes ($1,999)	Memory, Disk	Live, Dead	Desktop, Mobile
Volatility	No	No	Memory	Dead	Desktop, Mobile
TSK	Yes*	No	Disk	Dead	Desktop
ProDiscover v9	Yes	Yes ($50)	Memory, Disk	Dead	Desktop, Mobile
Bulk Extractor	No	No	Disk	Dead	Desktop

Table 3: Forensic Tools Comparison

***** TSK GUI is provided by means of Autopsy.

- **Reference**:
  - **FTKv4**: Forensic Toolkit, version 4
  - **TSK**: The Sleuth Kit, version 4.1.3
  - **IEF**: Internet Evidence Finder Bundle (IEF Standard + IEF Triage)

The most important thing to notice in this table is the fact that no open-source tool targets more than one type of evidence (either disk or memory) at the same time. Monocle is intended to target both memory and disk.

SECTION 2.4

The Volatility Framework

This section is intended to briefly depict the general functioning and capabilities of the Volatility Framework, as it is one of the technologies included within the present software. Justification regarding the incorporation of this technology within the present software is presented in subsection 3.6.

Volatility is a forensic analysis framework firstly released in 2007, and it is targeted to volatile data storage (i.e. RAM memory). It was one of the tools firstly introducing people to the importance of analyzing the running state of a machine rather than only the persistent storage, such as the hard drive. This persistent storage were the main focus of digital forensic procedures until that point.

The Volatility Framework is mantained by a non-profit organization, which released the framework under the GPL version 2 license. This fact has helped in order to create a community of collaborators around the project, which actively contribute to its development. This has allowed the project to seamlessly grow from its begging to its current 2.4 version.

2.4.1. Volatility Profiling system

RAM volatile memory is usually fragmented in a different way depending on the operating system running in the device, which traditionally supposed a huge problem when the analysis phase occurs (i.e. it was difficult to know at a first glance the memory space of a process or the location of resources mapped to memory). The main strength in the Volatility Framework is the fact that it is able to reconstruct a whole RAM memory image once the operating system it belongs to is known.

This is done by means of *profiles*. The Volatility profiles define the characteristics of a given OS when regarding to memory allocation and management in order to reconstruct the whole image. This reconstruction is standardized so as the general use plugins can operate over the memory dump independently of the operating system it came from (i.e. this of course won't work if the plugin being used is targeted to found something not existent in the source OS, such as Registry entries on a Linux system).

The Volatility Foundation provide many profiles for the most common operating systems,

such as Windows versions XP to 7, or the most recent Linux or MacOSX systems. There is however the possibility for the analyst to specify its own characteristics within the profile in order to create new custom profiles (i.e. useful in case the target system is a rare one or it is not among the already supported ones).

2.4.2. Volatility Module system

As the nature and needs of conducting a forensic analysis are not always fixed, there must be a way for the analyst to decide which tests and checks do he/she wants to perform instead of forcing the analyst to perform all of them.

The way many forensic tools (i.e. including the Volatility Framework) manage this variance in the procedures is by means of making the analysis software extensible by nature. The software itself provides a common interface in order to access the analysis results, whereas the different operations performed over such results are performed by the so-called *modules*.

A module is just a set of operations and checks (i.e. like a small program) which use the functionality provided by the main tool or framework. In the case of Volatility, this modules run on top of the reconstructed image, and perform a wide variety of tasks, from listing the processes running in the machine, to retrieve the internet history of the system.

The same way as it does occur with the Volatility profiles, modules can also be created by a third-party analyst in order to fit its needs. Here is where the fact of having a community around the project is noticeable, as there are many individual-developers which create their own modules in order to perform specific well-targeted scans.

2.4.3. Volatility as a Python library

The Volatility framework is implemented completely using the Python Language (i.e. widely used in the IT security field, as described in subsection 3.6). This has allowed the Volatility Foundation to be able to ship the Volatility Framework as a third-party library which can be included within a Python script for its automatic usage (i.e. not using command-line tools but using the Python API itself).

The main problem with this implementation is however the fact that it is complex to set

up the environment properly in order to use the framework from an external script. There are many different parameters and configurations which must be properly settled up, while the documentation provided by the Volatility Foundation regarding this fact is not quite clarifying. This fact is disappointing as the Volatility Framework is a highly-useful tool, and an easier inclusion would be of a high value for the digital analyst community.

SECTION 2.5

The Sleuth Kit Framework

This section is intended to briefly describe the capabilities of the Sleuth Kit Framework [2] (i.e. usually referred to as its acronym, TSK) due to its relevance in the computer forensics field. This framework is also relevant as it does make use of similar procedures than the present developed system.

Notice that although the tool is not directly incorporated in the present system, it is a firm candidate to be so in future versions of the framework proposed in this Thesis.

2.5.1. TSK Overview and Capabilities

The Sleuth Kit is a library and collection of command line tools which allow the forensic investigation of disk images. It is among the most popular open source tools for forensic analysis due to its maturity and stability.

The community behind TSK has provided the tool with a wide compatibility regarding file types, analysis methods and forensic procedures, making the framework a reliable and stable yet free forensic analysis utility.

The tool is composed of a multi-stage procedural flow, which allows the user to specify which actions are to be executed for each analysis type. The main stages the application flow goes through are depicted in Figure 3, and can be described as follows:

1. **File Extraction**. The files are carved from the disk and incorporated to a central DB which indexes them.

2. **File Analysis**. The set of analysis and operations to be performed are executed in order, in a pipelined-based model (e.g. calculate hash, unzip files...).

3. **Post Processing / Reporting**. This process is also pipelined, and it does provide tools in order to merge results together or generate automatic reports.

In addition to the set of libraries and command line tools, the Sleuth Kit project also counts with a graphical user interface, namely Autopsy. Autopsy is a project intended to provide a set of easy-to use GUI so the user doesn't need to make use of the command line. This

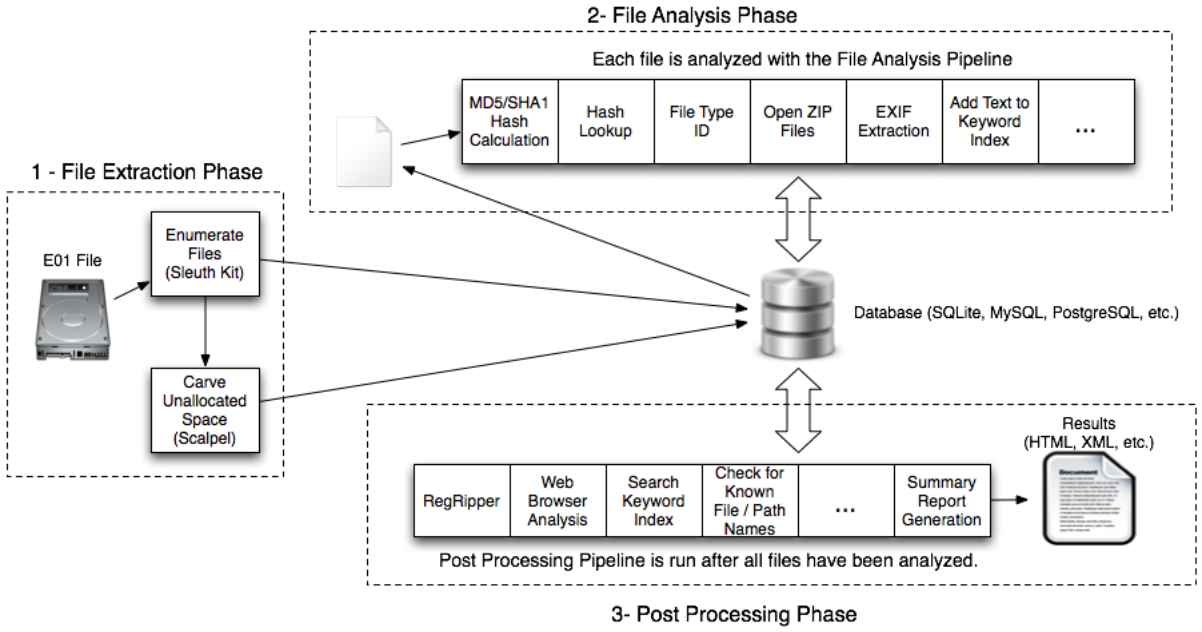

Figure 3: The Sleuth Kit pipeline process [2]

saves time and efforts to the analyst when performing his/her tasks.

The pipeline process of TSK is a good visual reference for what a forensic process is. It is easily appreciated how such processes are composed of several independent tasks belonging to different phases (e.g. analysis and processing). The elements elicited during those phases will be related afterwards in order to present the actual reports. It is also noticeable how every single evidence found is related with similar evidences (i.e. TSK does this by means of a DB).

2.5.2. Relation with this project

As it was commented at the beginning of the present section, The Sleuth Kit framework was not included within the present software project. It is interesting however to take a look to the differences between TSK project approach and the ones here in order to perform analysis and the one proposed in this project. It has to be noticed however that TSK is a mature project while the one presented in this document is barely on his first beta version.

The main difference arises in the reading process. TSK will read the volumes from a disk dump and reproduce the control structures of such volumes in order to determine the disk

internal structure (e.g. Allocation tables, inodes...). The present tool relies in the mount utility of the current system in order to access the file contents and elements contained within the disk in order to mount such volumes in a read-only way.

This has an effect on the direcperations which can be performed over the disk. The present tool is not able to perform data carving over non allocated space, whether The Sleuth Kit provides a carving utility which is able to find deleted files within the unallocated areas of a disk.

The pipelining process mentioned in subsubsection 2.5.1 is another area in which TSK excels. The present project provides a single-module based plugin system, meaning that plugins are executed independently of each others and do not have relation with the results coming from the others. The DB-centralized approach of the sleuth kit allows this to be done in order to run many analysis over the same data set so they complement each other.

There is however a huge difference between TSK and the present project which must be remarked here. While The Sleuth Kit project is intended to analyze only hard disk dump files, the present project can target both hard disk and memory dumps in order to perform a more wide variety of analysis types. This is of special interests as the malicious agents evolve and the criminal procedures become more complex (e.g. memory-allocated virtual file systems).

So as a conclusion, and although The Sleuth Kit would be a perfect fitting for the present project, it has not been included due to the time and economic limitations affecting this prior version of the software. Future versions however might integrate the framework as to provide all the power of the aforementioned tool.

Chapter 3

Analysis

SECTION 3.1

General Perspective of the system

This section is intended to further develop the description of the system in order to fulfill the objectives stated in page 15 of the present document, and it will give a more specific idea of the elements and components required.

The proposed system must be implemented as a utility framework for the development of automated analysis on third party target machines. The framework must perform, without the requirement of user interaction, some of the tedious tasks which are common to any digital crime scene investigation phase [32].

- **Digital-feasible documentation of the evidence**. This includes the objective data which can be recovered from the evidence itself, as its location within the digital container, the metadata of the evidence (if any), and the destination where this evidence will be stored after analysis.
- **Recovery of evidence data**. Which is referred to the fact of gather the evidence file or evidence elements (if any), and to store the interpretations which can be optionally given by the investigator depending on the context. This also includes to organize the data for further ease of access.
- **Securing of evidence data**. Implying the insurance of consistency by means of hash functions. This ensures the consistency of the files and the evidence tree generated, as well as the generated reports and elements of the analysis.

Regarding the target user of the project, the present system must allow the forensic analyst to develop effective and functional scan modules for evidence recovery. In addition, those developed modules must be flexible enough to fulfill the investigator's own requirements (i.e. the requirements of the analysis to be performed over the target system).

The framework must target the two main evidence containers within an IT system, as described in subsubsection 2.1.2, page 24, which are Memory and Disk.

Due to the nature of the framework itself, it must be completely extensible by means of user modules, and it must provide a standardized way to proceed in order for it to be able to process the data retrieved from the different extensions. The framework must also include utilities for the analysts to use when developing the different extensions as to make this development easier and faster.

The system shall be implemented using a **cross-platform technology**, which allow its usage in multiple environments. This will give the investigators the opportunity to use it without incurring in extra infrastructure costs.

As to reduce the scope for a first version of the framework, its target will be reduced to the conclusions described in section 2 of the present document, which means that **only *RAW* formats of evidences coming from Windows systems will be supported.**

SECTION 3.2

Socio-economical study

This section is intended to provide a socio-economical point of view of the project. This point of view takes into account not only technical issues, but the effect in the community and potential targets of Monocle.

Monocle's main purpose is digital forensic analysis on personal computers. Such type of analysis is intended to prove interaction between the user and a digital system in a legal and consistent way. Such implications make the tool quite useful both for law enforcers and system managers at a first glance.

Law enforcers can benefit from this tool as it might prove the user is violating local or international laws. For example, Monocle can be used in order to recover elements existing on a user machine related with illegal activity, such as child pornography or illegal distribution of copyrighted material. Moreover, Monocle can be also used in order to detect malware on a system, as it is equipped with the tools to recover and analyze forensic dumps of the live machine. System managers can also benefit from Monocle, as it can be used in order to check specific areas on employee's equipment, thus preventing potential threats and information leakage. In case such information leak occurs, it can be traced back within the system.

However, Monocle could not only benefit end users, but also promote academic study on malware due to the fact that such malicious executing elements can be reconstructed and dumped for further analysis. This is useful in order to freeze malware during execution, which is a relevant approach to study behavioral malware. This is known as malware dynamic analysis [33], [34].

Due to its open-source and extensible nature (refer to subsection 1.3), Monocle is potentially a community-driven software. This way, Monocle provides better low level analysis of the results, as found evidences can be automatically classified and processed. If the applied plugin already exists, the analyst can execute it without dealing with low level details beneath the implementation. This saves time and effort in the investigation process, and implies several analyst can share and improve plugins being used in the application in order to benefit overall functionality.

SECTION 3.3

Legal framework of digital forensics

The present section describes the different jurisdictional issues related applying to computer forensics within current Spanish law regulation.

Being digital forensics a method employed in judicial cases and law enforcement, there are several considerations and regulations to take into account when interacting with forensics scenarios. Forensic tools are highly regulated due to the fact that they might be used as tangible evidences and statements which can lead to punishment of real persons and to administrative fines or even imprisonment. Legal issues regarding forensics are highly related to information privacy, legal acquisition of evidences and maintenance of the chain of custody for such evidences.

The following legal framework is established by means of the Civil Prosecution Law (Ley de Enjuiciamiento Civil, LEC 1/2000, January 7th) [35] legal framework, which regulates the civil processes available in order to claim material rights, e.g. civil or penal.

3.3.1. Fundamental Rights

Fundamental rights are those granted by means of the Spanish Constitution to every human being and which are considered essential in the political framework Spain is composed of. Such rights can be mainly divided into social, economical, personal and public types. Of specific interest regarding the present project are the following fundamental rights:

- **Right to juridical security and juridical guardianship**. Which is aimed to grant a penal process following a fair and guaranteed methodology. This includes veracity of the evidences stated and fairness in the punishment.
- **Right to private life**. This comprises rights regarding self-image and own-life rights. This is related with forensics as any digital investigation has often to dive into personal assets of the subjects under investigation.
  - **Fundamental right of data protection**. Stated in year 2000, SSTC 292/2000 [36] specifically defines the data protection right as a different right than the private life right.

3.3.2. Personal Data Protection and Data Conservation Laws

Personal Data Protection Law (in spanish, Ley de Protección de Datos de Carácter Personal, LOPD 15/1999, December 13th) [37] is a fundamental law aimed to protect physical persons with respect to the treatment their personal data. This law regulates *who* has the data, *why* is it used for and *who* this data can be shared with. This law also recognizes some rights the owners of the data have over them, such as modification, checking and deletion in some cases. Depending on the nature of such data and the content, they can be classified in either low, medium or high level of security. The definition of such level is determined by the criteria stablished in LOPD RD 1720/2007, art 81 [38]. Whether this security level and the corresponding obligations have to be considered when dealing with digital evidences, it is important to bear in mind that **all data gathered with law enforcement purposes belong to the high security level**.

Although this personal data can be found in the user machine, it is difficult to trace once the information is send over the internet or any other communication channel. In this matter, the Law of Data Conservation regarding Communication and Public Networks (in spanish Ley de conservación de datos relativos a las comunicaciones y las redes públicas, L 25/2007 of October 18th) [39] forces telecommunication operators to keep data relevant to trace the origin and destination of telecommunications (e.g. telephone calls, internet access, e-mail) during a time ranging from 6 months to 2 years.

3.3.3. Felonies to take into account when developing Monocle

In order for Monocle to perform a legit interaction within the user data and a consistent maintenance of the chain of custody during the analysis, it is necessary to keep in mind some legal aspects defined in Spanish Criminal code regarding telematic felonies. Such elements are defined in LO 10/1995, November 23th, art 10 [40].

- **Privacy felonies** (art. 197). Not to gather information of a third party without explicit authorization from its owner or a judge. This includes emails, documents, personal items (e.g. computers), as well as to record, image or intercept someone's communications.

- **Illicit appropriation** (art. 252). Generally applied to material elements. Monocle has to provide a way to track analysts in order to control sensitive information and the belongings of the subjects under investigation.
- **Damage to digital documents** (art 264). This comprehends any damage to be caused to a digital infrastructure. Special punishment exists if the damage was caused by illegally using a third-party credentials or access to the given system. Monocle has to grant the analyzed system is not getting damaged because of Monocle's operations over the data residing in the system.
- **Documental forgery** (art. 390 to 400). This includes to alter essential contents of a document, document impersonation, illegitimate authorship of the document and false claims or statements introduced in an official paper. Monocle has to provide only reliable claims regarding the analysis performed.
- **Custody infidelity** (art. 413 to 416). Including punishment for any public employee unveiling, destroying or hiding documents under his/her responsibility. This implies Monocle must generate unbiased analysis over the evidence sources in order to preserve the information as it is.

SECTION 3.4

Software High level decomposition

Based on the analysis from subsection 3.1 on page 40, a high-level identification of the system can be performed in order to distinguish the different components and elements to be implemented through the creation of MONOCLE. These components represent a general idea of the final implementation, and are a prior approach to the framework final composition.

It is quite important to state properly the different components as to provide the necessary modularity to the system. This will improve the development speed and will additionally make the system easier to maintain and improve, due to the fact that the components are well differentiated and the effects of modifications in one of them will be well known.

Figure 4 shows the high level decomposition diagram of the system attending to the analysis performed on subsection 3.1, page 40. There are four noticeable elements in the system, attending to the role in the system and functionality to perform.

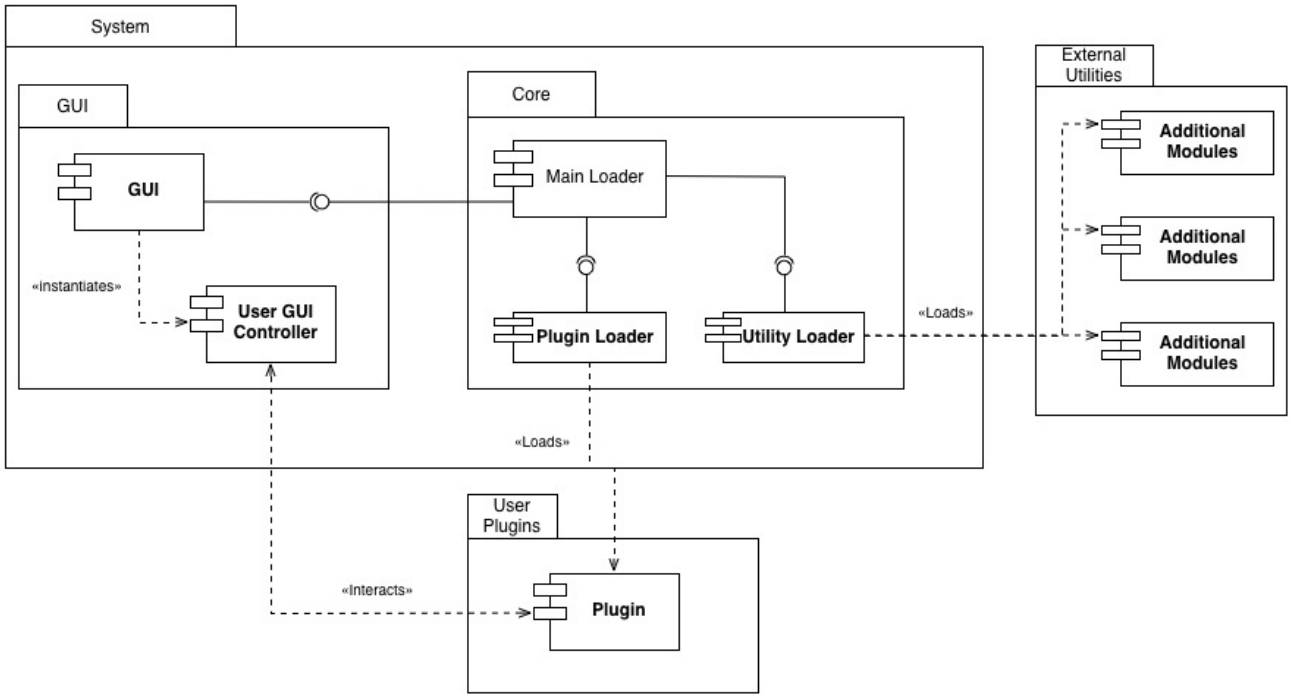

Figure 4: High level decomposition diagram

The following develops the ideas over each of the components defined in Figure 4:

- **GUI**. Graphical User Interface, which will be independent of the backend implementation, and which can be divided into the graphical elements required by the framework itself and the graphical elements which can be required for a user module (described below).

- **Core**. As the name implies, this is the framework core itself. This part of the system does not depend on the user modules employed. The Core component will be in charge of loading the user modules to be executed within the framework, as well as the utility classes the user module can make use of. The tasks to be executed by the core are the following ones.

  - *Loading of User Plugins*. Dynamic loading of user plugins on user demand. This plugins are described further in this section.
  - *Loading of Utilities*. Automatic loading and setup of the tools and elements to be used within the plugin. This elements can be automatically loaded by default or specified by the user plugin on load. The utilities are described further in this section.
  - *Automatic management of evidences*. The *core* defines the Evidences to be retrieved by the user module, and it does perform automatic operations over those evidences, such as hashing, classification, ordering and report generation after the analysis.
  - *Generation of reports*. After an analysis is performed, the system will store the evidences found and the analysis results in a persistent way. In order to do so, a report will be generated in XML format. XML has been chosen due to the fact that it is easy to use it for derived operations (e.g. creation of PDF or parsing it to a program input).
  - *Evidence mounting*. As for Memory and Disk analysis, there will be always an analysis file target, which will be either a disk image or a memory RAW image. The system will transparently mount and prepare the file to be analyzed in order to provide the final user an easy way of interacting with the evidence source (i.e. direct read over the memory file and full browsing over the disk image file system).

- **User Plugins**. Those are the actual plugins or modules the final user will execute over the framework. They will define the actions to be performed against the target system, and can vary depending on the purpose of the analyst coding them (e.g. Whether the module is targeted to RAM memory analysis or to extract registry keys with a given format from a dead hard disk).

- **External Utilities**. The set of tools the framework will provide to the analyst. This tools will add extra functionality to the framework in order to be used by the user modules during their execution. By analyzing the considerations presented in section 2, the following set of utilities has been selected on this first version of the framework for being included.
  - *Memory analyzer and reconstructor*. In order to perform effectively memory forensic analysis, it has been decided to include the Volatility Framework (i.e. introduced in section 2) among the utilities to be loaded on user demand. This will allow the reconstruction of RAM memory and the proper and easy analysis of it by the user plugins.
  - *Registry analyzer*. Being the registry one of the most efficient sources of evidences [41], the framework will provide transparent interaction over the registry hives present in windows systems from the user plugins. The framework will be in charge of parsing and interpreting such hives in order to provide the final user a direct navigation over the key-value pairs stored in the different hives.
  - *Timeline Generator*. As described in section 2, one of the biggest challenges Forensic Analysts face when presenting evidences to a court is the description of the evidences to the jury. The framework will help with this issue by creating a graphical timeline, which will help to clarify and better expose the evidences and the interaction among them.

SECTION 3.5

Technological Analysis

This section is intended to discuss the different technologies that can be used in the different main components the application is divided in (i.e. depicted on Figure 4 of subsection 3.4). Those technologies shall adapt to the objectives of the project and provide a useful functionality, as the inclusion of additional technologies usually requires an additional effort which can be unfruitful if the cost of integration is bigger than the benefits in the overall system.

3.5.1. Imposed Technologies

The present project is neither a continuation nor an improvement over any other existing project. This implies that there are not explicit imposed technologies which are inherent to this system and must be followed during the development. Thus there is freedom of choice in order to develop the framework by using the most convenient and useful technology.

There are however considerations to take into account when talking about the technologies to be used. One of the most important ones is the fact that the current system will be implemented as a framework for the usage of third party plugins. This implies that there is necessary to bear in mind that the used technologies not will only affect the internals of the system, but also might affect the final user by applying a set of constraints over the plugin development procedure.

The scope of the project also must be taken into account for this prior version, as to reduce the compatibility and tasks to perform. This is necessary in order to fulfill the proposed requirements in an effective manner for the first version of this framework.

3.5.2. Technologies applied to the component Core

As described in subsection 3.4, the *Core* component is in charge of load all the components of the framework itself. This implies that this component must employ a technology as compatible as possible with the rest of elements, since this component will be connecting to all the others. It is necessary for this component to implement dynamic loading of modules and

dependencies, as to incorporate additional utilities. Due to the fact that this component will be dealing with the integrity of the data and with the evidence files themselves, it would be positive the usage of a technology which incorporates an easy way to perform this operations.

Some programming languages are better suited for those tasks than others, and we are going to comment the three most relevant ones to take into account in the implementation of this component, namely Java [42], Python [43] and C [44].

Java is a general purpose programming language following an object oriented paradigm. Applications running in Java are executed within a virtual machine, called JVM (i.e. Java Virtual Machine). Thus makes Java a cross-platform language, as it does not depend on the architecture beneath the JVM. This makes the system target much broader. On the other hand, this virtualization causes an overhead which will slow down the computation procedures a bit. Dynamic loading of elements through Java is done by means of the inclusion of JAR (Java Archive) files in the main program by means of the Java ClassLoader. Java is a mature language, with a well-formed documentation and solid structure, which must also be taken into account.

Python is an interpreted (scripting) programming language which also provides cross-platform support. Python supports multiple programming paradigms (i.e. object oriented, function oriented...). The main advantage of python relays on its own implementation and philosophy: Python was designed to be a clean and easy to read language, which makes it quite suitable for fast prototyping and coding of scripts and other utilities. Being Python an interpreted language, the dynamic load of code is quite simple, as Python source code can be directly imported as a library in the main program. Python is already on a mature state, with good documentation, although it is currently suffering a major version change with a deep syntax redefinition, making programs working on some versions not valid for the new ones. (i.e. Version 2.7 vs 3.x)

The third option is the C programming language. C is a compiled language, which implies that it must be compiled on the machine the framework is going to run prior execution. This increases the complexity of deploying this application, but it provides on the other hand a better execution performance as the bytecode generated can be optimized depending on the system where it executes. C programmers operate at a lower level than Java and Python ones, dealing with native data types and memory allocations. Dynamic loading of elements in a C program can be done by means of C *archive* files. C is the most mature language of the aforementioned, and has being developed since 1990.

3.5.3. Technologies applied to the component User Plugins

User Plugins component is the component enclosing all the modules to be developed over the framework being developed. They have thus to be compatible mainly with the *Core* component and the *external utilities*, as they will make use of them by interacting with the framework core. If this compatibility is not possible, there must be at least some way of interfacing the elements between different technologies in order not to reduce the functionality of the overall system. Additionally, it will be useful that the used technology was easy to use and to develop, as to reduce the complexity for the forensic analyst writing the application plugins.

This requirements lead actually to the same options described in subsubsection 3.5.2, as interfacing technologies among them when the same technology can be used in both component is an additional unnecessary overhead unless there are a specifically important reason. The proposed technologies are thus again Java, Python and C.

Regarding Java programming language, it is noticeable that Java is a high-level programming language, helping the coding process quite a lot as many of the underneath procedures performed by the system might be skipped (e.g. the assignation of memory, as Java provides a Garbage Collector in order to optimize required memory at run time). Several third party IDEs (Integrated Development Enviroment) exists for this language (such as Eclipse or IntelliJ IDEA). It is important to bear in mind this fact, as an IDE greatly boost the creation speed and the easement of the overall coding.

With respect to Python, it is also a high-level programming language, thus having the aforementioned advantages. In addition to that, being an interpreted language allows real-time testing on its own interpreter (named IDLE). This way it is easy for the developer to test the functions and elements to be developed in a fast and effective way by performing simple tests. There are some IDEs available for coding in Python, such as JetBrain's PyCharm.

The C programming language is different from this previous two, as it works at a lower level than Python or Java. A tradeoff between efficiency and development speed can be appreciated here, as the fact of gaining more control over the actions performed by the module implies the fact that there are more elements to take into account when developing (i.e. elements which are not managed automatically, as in the two previous cases). C also has several IDEs available, as the aforementioned Eclipse and NetBeans IDE.

3.5.4. Technologies applied to the component GUI

The GUI component will be in charge of managing the user interface, and it will interact both with the *Core* and *User Plugins* components. There must be thus a way for communicating those components asynchronously, as usually graphical interface elements are ran in a different process of execution. The GUI component technology shall allow the creation of a clean and easy to use interface, and shall additionally provide all the means required in order to expose the framework functions which require a graphical display.

Java provides a huge variety of GUI frameworks to use within this component (e.g. AWT, Swing, JGoodies, JavaFX). Those are well integrated with several IDEs (e.g. Eclipse) which allow simply drag and drop of the elements within a frame in order to create the GUI. This is translated into a huge boost in the developing process since it is way easier to follow this procedure than to deal with heights and widths from the source code.

Python provides as well several different libraries to create GUIs, being the most widely known Tkinter and WxPython. This GUI frameworks are quite easy to use, although there are no graphical editors such as the ones existing for Java GUI creators, and both provide cross-platform support.

The programming language C provides also libraries for Graphical User Interface, such as GTK+ or Qt, which provides cross-platform support. There are a variety of editors available for this, such as the aforementioned NetBeans.

3.5.5. Technologies applied to the component External Utilities

As described in section 3.4, page 46, there are a variety of utilities to take into account for this component which can perform the necessary actions required.

The Volatility Framework seems to be a definitive unique option for the memory reconstruction and analysis. Volatility is coded in Python, and can be included as a library to programs using it. There is however another option available, namely its execution through the command line This however will imply the creation of an interface for the different call options to be available to the user plugin.

Regarding the other utilities proposed in subsection 3.4, each one of the aforementioned have their weaknesses and strengths by interacting with them. The utilities proposed are however quite platform-specific, as the operations perform differently depending on the OS being targeted. This implies that the most suitable technology might be in some cases the platform solution proposed.

As we are targeting Windows Systems, use of the *mountvol* command can be done in order to mount a specific file system evidence. Linux and Mac also provide their own commands for doing so, namely *mount* and *hdiutil*, respectively.

Windows also offers utilities for the registry analysis, such as the Registry Utility. Third party solutions might include the use of *RegRipper*, a Perl-written utility for the Windows Registry inspection. Python also provides the *Registry* library, which provides an interface that can operate over windows registry hives.

SECTION 3.6

Non-imposed technologies selection

This section is intended to show the technologies to be used in the final development of the present system, as well as the reasoning and logic behind the decision procedure for choosing them.

As mentioned on subsubsection 3.5.3, there is no advantage in the usage of more than one different technology in the development if it can be avoided, as this implies the necessity of interface the components among them, and will result into an increased number of dependencies. Thus, it has been decided the usage of a single technology for the whole system as to reduce the complexity of the overall system.

The most *restrictive* technology requirement (i.e. although it is still a non-imposed technology) is the usage of the Volatility Framework. The integration of this technology will be highly more efficient if Python were our main programming language. Java and C would require an interface via command-line execution within the Volatility Framework, which can be avoided if possible.

Related to this fact, and regarding the technologies discussed in subsubsections 3.5.2, 3.5.3, 3.5.4 and 3.5.5, there is a main decision to be taken regarding the programming language to be used within our framework.

The first discarded technology was C, as the benefits which can be obtained from its usage (i.e. as described in the aforementioned sections), are not as many as the usage of either Python or the Java languages. C will not provide a portable cross-platform framework, and the fact that it operates at a lower level than the other two options will make the development of the different tools slower than desired.

Regarding Java, there will be indeed advantages when using this technology as the main one. The portability of the technology among different OS and the object-oriented paradigm Java makes use of are great factors to take into account when considering this option. There is however a great drawback with this technology, namely the fact that it needs to run on top of the Java Virtual Machine, and it is not so easy to include user plugins at run time due to the fact that Java is a compiled language.

Because of all this considerations, the final technology to be used for the coding phases of the framework is Python 2.7. Python is a scripting interpreted language which allows mul-

tiple different programming paradigms. It is focused in clarity and it is quite suitable for fast-prototyping of software due its easy coding syntax, which will result into a short time for users in order to create new modules or customize existing ones. Python provides cross-platform compatibility, being only dependent on the Python interpreter for it to run on an end-system. The fact that Python is an interpreted language is also key in this decision, as the inclusion of new modules can be performed lively just by specifying the file to read from (i.e. without an actual compilation, although Python does optimize the code for further usage if needed).

Python is a commonly used option in the IT Security field [45]. It is interesting to bear this in mind as it enhances the possibilities of future tool integration and further work over this project lines. The big standard library provided within the Python distribution [46] makes this technology easy to deploy, as many of the required dependencies are shipped with the standard Python installation. Python provides additional tools, such as Distutils or pip, which allow the installation of dependencies automatically. Further reading can be consulted at [47] regarding this topic.

Python provides third party libraries which will allow to include the functionality described in subsubsections 3.5.2 3.5.5. The *python-registry* library [48] allows transparent navigation through the key-value pairs in a Windows Registry hive, and it does fit perfectly with what is described in subsection 3.4. The *timeline* library [49] provides also an easy way to construct graphical timelines based on WxPython GUI framework. The ElementTree XML API [50], shipped within the standard Python library, is additionally a perfect tool for constructing and parsing XML data from and to the framework.

SECTION 3.7

Software Requirements

In this section there will be introduced the software requirements applicable to the present project in order to guarantee its proper functionality and correct behavior. The requirements presented within this section, and according to the IEEE recommendations for requirements engineering definition [51], will follow the ESA (European Space Agency) methodology [52]

The software requirements attributes present in the beneath tables include the *Identifier*, *Name*, *Description*, *Need* and *Priority* elements, as the rest are assumed to be constant and as described as follows:

- *Stability.* All requirements are stable through the software lifecycle, and do not change as the project progresses.
- *Source.* The source of all requirements is considered subsection 3.5, subsection 3.6 and subsection 4.1 as there is no URD document.
- *Clarity.* Clarity states for the fact that there is no ambiguity in the requirement. It includes an explanation in case of conflicting requirements. All requirements are supposed to be clear (i.e. there are not conflicting requirements).
- *Verifiability.* Whether the requirement can be checked against the software tests and can be demonstrated that the system implements the requirement. All requirements are verifiable and are verified by means of acceptance tests (see subsection 3.9).

Besides, the ESA PSS-05-0 [52] defines each term of the ones considered for each requirement as follows:

- **Identifier (ID)**. Unique numerical or alphanumerical code identifying each requirement. Used for further referencing within the project.
- **Name**. A meaningful phrase in a human-readable format which summarizes as much as possible the essence of the requirement.
- **Description**. A full description of the requirement itself. Extends the name and conforms the requirement itself. Can include references to other requirements.
- **Need**. Degree of importance of the requirement within the project. Higher need requirements are non-negotiable, whether lower need ones are less important for the project and might be modified slightly on user's agreement.

- **Priority**. Used for incremental delivery, it states the degree of priority of a given requirement to be fulfilled in the project.

3.7.1. Functional Requirements

The following table exposes the functional requirements to take into account in the development of the present project.

Functional Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
F-01	Analysis Performance	The system must recover data elements from data dumps by mounting the data dumps digital containers	High	High
F-02	Operation Target	The system must perform [F-01] over a digital dump and not a live system	High	High
F-03	Memory Analysis	The system must be able to analyze memory dumps as part of F-02	High	High
F-04	Disk Analysis	The system must be able to analyze file system dumps as part of F-02	High	High
F-05	Evidence Retrieval	The system must copy evidences found during [F-01] in the local disk	High	High
F-06	Evidence auto management	The system must perform automatic management of evidences as described in subsection 4.1 of this document	Medium	Medium
F-07	Module Loading	The system must load analysis plugins at run time	High	High
F-08	Module Classification	Plugins loaded [F-07] must target either F-03 or [F-04]	High	High
F-09	Persistence of results	The results of the analysis performed must be stored in a persistent way. Such data must be elements contained in the data dumps, and shall include files, registry key-value pairs and memory-residing data.	Medium	High

Functional Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
F-10	Utility Tools inclusion	The framework must include utilities as described in subsection 4.1 of this document	High	High
F-11	Volatility tool existence	The Volatility Framework must be integrated within the system	High	High
F-12	Registry Plugin existence	The Registry Library must be integrated within the system	Medium	Medium
F-13	Timeline Plugin existence	The Plugin Library must be integrated within the system	Medium	Low
F-14	Timeline Generation	The system must be able to generate a timeline by means of the use of F-13	Medium	Low

Table 4: Functional Requirements Table

3.7.2. Interface Requirements

The following table exposes the interface requirements to take into account in the development of the present project.

Interface Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
F-15	Framework loading	The system shall be included in user plugins as a Python library.	High	High
I-01	Volatility Inclusion	The Volatility Framework [F-11] must be included as a Python Library	Medium-High	High
I-02	Registry Inclusion	The Registry Library [F-12] must be included as a Python Library	Medium	High
I-03	File system Mounting	The mounting of disk images will be done by means of OS-dependent system calls	High	High
I-04	File system dump format	The disk dump files must be in an uncompressed format (e.g. RAW)	High	High
I-05	Memory dump reading	The reading of memory dumps will be done by means of OS-dependent tools	High	High
I-06	Memory dump management	The memory dumps analyzed must be in RAW format	High	High
I-07	Persistence format	The persistence described in F-09 will be done by means of an XML file	Medium	Medium

Table 5: Interface Requirements Table

3.7.3. Operational Requirements

The following table exposes the operational requirements to take into account in the development of the present project.

Operational Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
OP-01	CML Run option	The Framework must be able to be run from command line	High	Medium
OP-02	System Command Syntax	The command-line [OP-01] syntax of the arguments used in order to call the system is defined as: [command] [-m <memory dump> or -f <disk dump>] -p <Persistence Path>	High	High
OP-03	GUI Existence	The Framework must include a GUI in order for the user to execute it	High	Medium
OP-04	GUI Run option	The GUI [OP-03] must include an option to launch the analysis F-01	High	Medium
OP-05	GUI Loading Options	The GUI [OP-03] must differentiate the plugin types specified in F-08	Medium	Medium
OP-06	Plugin load on GUI	The GUI [OP-03] must include a mechanism for the user in order to specify which plugin the user want to execute [F-07]	Medium	Medium
OP-07	Evidence Source load on GUI	The GUI [OP-03] must include a mechanism for the user in order to specify the path within the evidence element to analyze,	Medium	Medium
OP-08	GUI Loading of persisted analysis	The GUI [OP-03] must include a mechanism to load the persisted analysis coming from F-09	Medium	Medium

Operational Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
OP-09	Analysis Summary GUI Generation	The GUI must provide a summary of the analysis [F-01] evidences [F-05] once it has finished	High	Medium
OP-10	Timeline GUI Generation	The GUI must provide a way to visualize the results from F-13	High	Medium
OP-11	Volatility GUI choosing	The GUI must provide a way to choose between the different Volatility [F-11] Profiles to use in the analysis F-01	High	Medium
OP-12	GUI-CML mapping	The command-line [OP-01] syntax of the arguments will be constructed from the GUI by means of the parameters received from OP-06, OP-05, OP-07	High	High

Table 6: Operational Requirements Table

3.7.4. Security Requirements

The following table exposes the Security requirements to take into account in the development of the present project.

Verification Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
S-01	Mounting of evidences mode	The file system evidences mounted [F-04] must be read-only to prevent data corruption	High	High
S-02	Automatic unmounting	The evidence targets [F-03, F-04] must be automatically unmounted when the scan F-01 is finished	High	High
S-03	Multiple hashing of evidences	The evidence automatic management defined in F-06, which implements evidence hashing, must perform more than one type of hashing for each evidence	High	High
S-04	MD5 usage	MD5 hashes must be computed for each evidence as part of S-06	High	High
S-05	SHA512 usage	SHA512 hashes must be computed for each evidence as part of S-06	High	High
S-06	Multiple hashing of evidences	The evidence automatic management defined in F-06, which implements evidence hashing, must perform more than one type of hashing for each evidence	High	High
S-07	Version number in report	The persistent report generated in F-09 must include the program version for court validation of the code.	High	High

Table 7: Security Requirements Table

3.7.5. Portability Requirements

The following table exposes the Portability requirements to take into account in the development of the present project.

Portability Software Requirements				
ID	**Name**	**Description**	**Need**	**Priority**
P-01	Mounting dependency	The mounting tool used will be platform-dependent	High	High
P-02	Windows Mounting support	A Windows-native tool for mounting will be supported in P-01	High	High
P-03	Mac OSX Mounting support	A Mac OSX-native tool for mounting will be supported in P-01	High	High
P-04	Linux Mounting support	A Linux-native tool for mounting will be supported in P-01	High	High
P-05	Programming API	Except from the aforementioned, all operations will use Python Language [F-15] APIs or Libraries	High	High

Table 8: Portability Requirements Table

SECTION 3.8

Software Use Cases

This section will describe the different use cases to be considered within the present project. This use case generation will help in the analysis procedure by eliciting new requirements and verifying the existent ones against the case flows described beneath.

The use cases description will follow the Unified Modeling Language (UML) use-case specifications [53], and will be hierarchically organized according to this very same specification, as shown on Figure 5.

3.8.1. Software Use Cases diagram

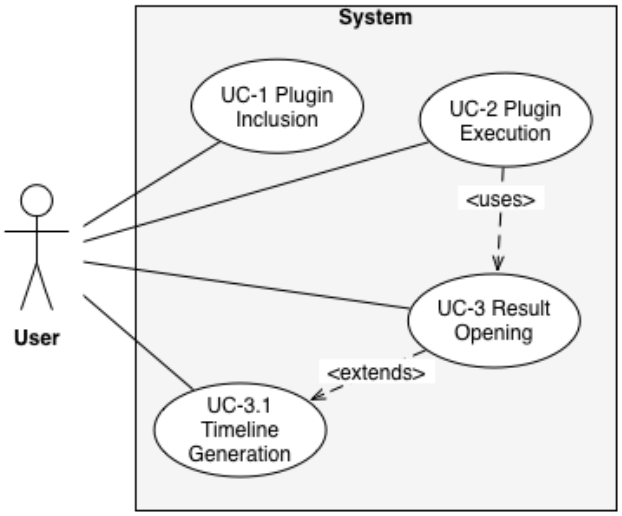

Figure 5: Use cases diagram diagram

Figure 5 shows the identified use cases according in Unified Modeling Language Format. Four different main functionallities can be identified at a first glance, some of them being related to others either by extension (i.e. the use case can be derived from the extended use case) or by usage (i.e. the use case used is comprehended inside the execution flow of the using use case).

Use case UC-2 refer to the process of loading a new user plugin into the framework. Whether use cases UC-3, UC-4 and UC-4.1 refer to the execution and post-execution of the user plugins (i.e. no coding from the user is required here). They do describe the actions performed

by the user and the expected reactions of the framework to such actions.

Subsubsection 3.8.2 further describes each use case in detail, along with its action flow, actors and elements interacting within the system.

3.8.2. Detailed Use Cases description

This section provides further description on each of the use cases presented in subsubsection 3.8.1. For each of them, several parameters are stated, which are defined as follows:

- **Name**. Human-readable identifier of the use case.
- **Identifier**. Unique identifier of the use case. Used for referencing. Use cases listed below represents the ones whose identifiers are depicted in Figure 5.
- **Description**. Extension of the name. It summarizes the operations of the use case in one or two concise sentences.
- **Actors**. Actors present and/or interacting with the software in the given use case. Only the user is defined for the use cases defined below.
- **Preconditions**. Set of constraints which are to be met in order for the use case to be executed without incidents.
- **Postconditions**. Results of the use case after the *operation flow* is completed and if the *preconditions* are met.
- **Operation Flow**. Chain of events and interactions composing the use case as such.

Use Case UC-1	
Name	**Plugin Inclusion**
Identifier	**UC-1**
Description	To include properly the module within the structure of the framework in order for it to be recognized.
Actors	• User

Use Case UC-1 (cont.)	
Preconditions	• The framework must be installed in the system along with its dependencies. • The user posses a valid plugin for the Framework (i.e. with a proper entry point, libraries and .py extension).
Postconditions	• The framework will properly load and read the newly include module.
Operation Flow	1. The user will locate the file containing the plugin. 2. The user will navigate to the installation folder of the Framework and will locate the *modules* folder, where the different plugins will be. There will be separated folders for Disk and Memory modules. 3. The user will move the file mentioned in step 1 to either the Disk or the Memory folders, depending on the target type of the new plugin. The folder in which the plugin resides will tell Monocle whether the plugin is a Disk or a Memory module. 4. On startup, the framework will load the plugin and the name of the file will appear in the correspondent section in the GUI.

Table 9: Use case UC-1 Table

Table 9 shows the first contemplated use case. This use case refers to the inclusion of a module already created to the framework module list. The framework will be configured to automatically recognize .py files in the specified folders and to load them dynamically (i.e. as described by requirement F-07, in subsubsection 3.7.1). Although there are a subtly difference in the inclusion of a Disk and a Memory module, two different use cases were no considered due to the fact that this difference only consists in the target of the final .py file.

Table 10 shows the second use case elicited, namely the actual execution of the plugin previously included as in UC-1. This plugin will yield a set of results depending on the evidences found on the target, which will be stored by the framework in an XML file as described

in requirement F-09, in subsubsection 3.7.1. This generated file will be also depicted in a graphical manner through the GUI of the framework itself.

Use Case UC-2	
Name	**Plugin Execution**
Identifier	**UC-2**
Description	To run a given module obtaining the analysis results
Actors	• User
Preconditions	• The framework must be installed in the system along with its dependencies. • The framework must be running. • The user posses a valid plugin for the Framework (i.e. with a proper entry point, libraries and .py extension). • The plugin is properly inserted into the available plugin list, as specified in UC-1.
Postconditions	• The framework will run the given module and will provide an analysis report.

Use Case UC-2 (cont.)	
Operation Flow	1. The user will select either a Disk module or a Memory module from the list of available plugins to use. (a) The user will select the appropriate Volatility Framework profile if the selected plugin is of *Memory* type and it does make use of the Volatility Framework. 2. The user will select the target object to analyze (i.e. A memory/disk RAW dump file) 3. The user will start the analysis. 4. A window with feedback on the analysis status will appear until it finishes. 5. The results will be automatically opened and a graphical report will be generated for the user to check out.

Table 10: Use case UC-2 Table

Table 11 describes the third use case encountered, which is the opening of an already existent result persisted in an XML file and previously created by a plugin within the present framework. The user will select the persisted XML file, and the analysis results will be showed in a graphical manner (i.e. in the same way as it does after performing the analysis, but without having to repeat it).

Use Case UC-3	
Name	**Result Opening**
Identifier	**UC-3**
Description	To open an existent analysis result XML coming from a previously performed analysis.
Actors	• User

Use Case UC-3 (cont.)	
Preconditions	• The framework must be installed in the system along with its dependencies. • The framework must be running. • A valid XML file created by the same process as the one described in UC-2.
Postconditions	• The framework will show the analysis summary generated from the XML file.
Operation Flow	1. The user will select the option to open a file in the main window. 2. The user will select the file where the analysis results are stored 3. The framework will generate the corresponding GUI analysis summary window.

Table 11: Use case UC-3 Table

Table 12 shows the last use case, in which the user wants to create a timeline from the results generated. This is treated as an extension to UC-4, as this option will be available from the graphical summary described in such a use case. A new window with an easy-to-read timeline will be created in order for the user to interact with it.

Use Case UC-3.1	
Name	**Timeline Generation**
Identifier	**UC-3.1**
Description	To generate a timeline from an existing set of results coming from an analysis as described in UC-3.
Actors	• User

Use Case UC-3.1 (cont.)	
Preconditions	• The framework must be installed in the system along with its dependencies. • The framework must be running. • A valid XML file created by the same process as the one described in UC-2.
Postconditions	• The framework will show a timeline coming from the data present in the aforementioned XML file.
Operation Flow	1. The user will load the results from a file as described in UC-4 2. The user will choose the option of generating a timeline in the analysis result window. 3. The framework will create the appropriate timeline from the data of the analysis.

Table 12: Use case UC-4.1

SECTION 3.9

Acceptance Tests Design

This section is intended to model the different acceptance tests which will be executed in order to verify the requirements stated in subsubsection 3.7.1

Once the general design for the system has been created, it is possible to define a set of acceptance tests which allow the developers to know if the software is performing properly for the stated requirements. It is quite important for an acceptance test to be checked against the requirements, leaving apart any other non-fixed feature.

The following table depicts the acceptance tests to be performed in order to verify the present software, as well as the requirements which are verified for each test. For each one, the inputs and the outputs which shall be obtained after performing it will be stated. A correct output given the required input will result in a successful test, whether any other output will count as a failed one. A failed test will require the revision of all the checked components in order to ensure their fulfillment.

Acceptance tests design			
ID	**Requirements Tested**	**Input description**	**Output Description**
AT-01	F-07, F-08, OP-03, OP-05, P-05	To load a plugin in the framework of type Memory or Disk by placing it in the corresponding folder within the framework installation	The name of the plugin shall appear in the framework window once the plugin starts
AT-02	F-01, F-02, F-03, I-06, I-07, OP-03, OP-04, OP-09, S-07, OP-12	To run an empty script properly loaded within the Memory section. The input file shall be a .raw memory file.	The plugin must return with no errors and must show an empty result page. A result folder must be created with no content apart from an XML file with the framework data (i.e. as there were no results).
AT-03	F-01, F-02, F-04, I-03, I-04, I-07, OP-03, OP-04, OP-09, I-03, S-07, OP-12	To run an empty script properly loaded within the Disk section. The input file shall be a .raw disk dump.	The plugin must return with no errors and must show an empty result page. A result folder must be created with no content apart from an XML file with the framework data (i.e. as there are no results).
AT-04	Apart from the ones in AT-02: F-05, F-06, F-09, S-04, S-05	To run a script properly loaded within the Memory section and targeting existing elements in the memory. The input file shall be a .raw memory file containing such evidences.	The plugin must return with no errors and must show a result page containing the targeted evidences. A result folder must be created with a XML file containing the framework data, as well as data containing the information of the retrieved evidences. Due its extension, retrieved data is defined further in the evaluation section (see subsection 6.2 and subsection 6.3)

Acceptance tests design (cont.)			
ID	**Requirements Tested**	**Input description**	**Output Description**
AT-05	Apart from the ones in AT-03: F-05, F-06, F-09, S-04, S-05	To run an script properly loaded within the Disk section which targets a set of files in a hard drive. The input file shall be a .raw disk dump containing such evidences.	The plugin must return with no errors and must show a result page containing the targeted evidences. A result folder must be created with a XML file containing the framework data, as well as data containing the information of the retrieved evidences. Due its extension, retrieved data is defined further in the evaluation section (see subsection 6.2 and subsection 6.3)
AT-06	F-09, I-07, OP-03, OP-08, OP-09	To open a XML file coming from a successful analysis by means of the open function in the framework GUI.	A result window containing the data stored in the XML file. No additional data must be created.
AT-07	Apart from the ones in AT-04: F-10, F-11, I-01, OP-11	To import the Volatility Framework feature from the framework and extract data from a memory dump by using one of the Volatility profiles in a script. The input memory file must be in .raw format and must contain the evidences to look for. The OS such memory dump comes from must be supported by Volatility and selected in the GUI prior analysis	The plugin must return with no errors and must show a result page containing the targeted evidences. A result folder must be created with a XML file with the framework data, as well as data containing the information of the retrieved evidences.

Acceptance tests design (cont.)			
ID	**Requirements Tested**	**Input description**	**Output Description**
AT-08	Apart from the ones in AT-05: F-12, I-02	To import the Registry tool feature from the framework and extract known data from a registry hive existing inside a disk dump. The disk dump must come from a Windows system. The input memory file must be in .raw format and must contain the evidences to look for.	The plugin must return with no errors and must show a result page containing targeted evidences. A result folder must be created with a XML file containing the framework data, as well as data containing the information of the retrieved evidences.
AT-09	F-10, F-13, F-14, OP-03, OP-10	To generate a Timeline by opening a valid result window (i.e. either coming from an analysis execution or a persisted analysis result) and selecting the option to generate the timeline	A new window containing the timeline for the retrieved data regarding the analysis selected must be displayed without errors.
AT-10	Apart from the ones in AT-02 (but OP-03, OP-04 and OP-09): OP-01, OP-02	To run an analysis in the same way than AT-02, but using the command-line instead of the GUI	The framework must return with no errors and must finish. A result folder must be created with a XML file containing the framework data, as well as data containing the information of the retrieved evidences.

Table 13: Acceptance Tests

Chapter 4

Design

SECTION 4.1

Software Final High level decomposition

By having into account the technologies described and selected in subsection 3.6, and by applying them to the previous high level decomposition proposed in Figure 4, a new High Level Decomposition is proposed in more detail, which considers the aforementioned selected technologies and which gives a better overview of the final general system.

Figure 6 shows the new final high level decomposition diagram. It is appreciable that some elements changed since the previous High Level Decomposition Diagram in Figure 4.

The first noticeable change is the fact that all the external libraries and elements to be included within the framework (i.e. namely *Monocle Framework*) are included in this diagram, along with their respective caller components.

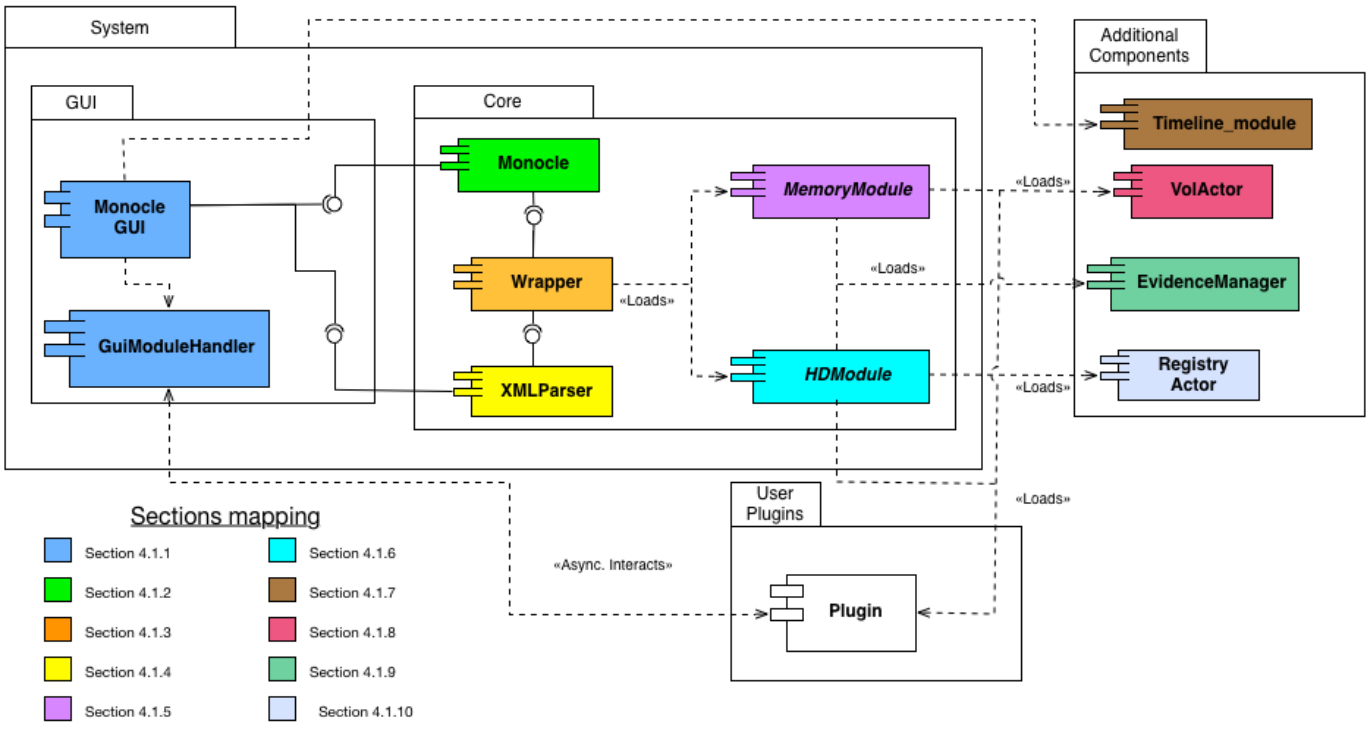

Figure 6: Final High level decomposition diagram

The new elements included in this final design are further described below.

- **Wrapper**. This core element will be the one in charge of configuring the Monocle Framework according to the parameters defined in the Monocle component (i.e. main entry to the system). It will either instantiate a *MemoryModule* or a *HDModule*, as well as perform the persistence tasks through the *XMLParser*.
- **Interface to load Memory/Disk modules**. As described in subsection 3.1, page 40, the modules can be divided into Disk and Memory modules attending to their target within the analyzed system. Two different interfaces are to be defined as they do differ into their mode of operation.
  - *MemoryModule*. Interface defining the elements required in order to perform a memory analysis. It is loaded by the wrapper and can instantiate a *VolActor* (i.e. described below) on user demand. An *EvidenceManager* will be loaded always.
  - *HDModule*. Interface defining the elements required in order to perform a hard drive analysis. It is loaded by the wrapper and can instantiate a *RegistryActor* (i.e. described below) on user demand. An *EvidenceManager* will be loaded always.
- **Timeline_module**. Library in charge of plotting a timeline representation from the data recovered in the analysis.
- **XMLParser**. Library in charge of making the analysis results persistent within a XML file in the evidence output directory.
- **VolActor**. Component in charge of loading and automatically configuring the Volatility Framework into the Monocle Framework. This component can be loaded on demand by a user plugin, and it will provide auxiliar functions with methods to call the different Volatility Plugins included in the Volatility installation framework.
- **EvidenceManager**. The *EvidenceManager* component is the main responsible in the evidence management procedure. It exposes functions in order to hash and store the evidences within a location in the analyst machine.
- **RegistryActor**. The *RegistryActor* is in charge of provide auxiliary functions in order to run and operate over the *Registry* module (i.e. introduced in subsection 3.6).

SECTION 4.2

Software Design

In this section, the different components defined during the analysis phase (i.e. section 3) and extended in the previous subsection will be further explained in detail. There will be an individual description of each of them, along with the associated sub components which might eventually appear when designing the functionality above stated.

Summarizing what it was presented in the previous section, the final component diagram will be the one introduced in subsection 4.1. This very same diagram will be used as point of reference when writing further subsections in the design process of this report.

The description of each component will be represented by a class diagram in UML notation. This will give a better understanding on the internals of the framework as well as it will reveal possible problems which can appear further in the coding phase. Convenience functions which are used internally will not be described in this section for the sake of clarity, as they do not really influence the final outcome of the design phase and are irrelevant in the final result of the framework.

Although some components are third-party libraries (i.e. they are not developed within the scope of this project), their interfacing elements might be described in further sections, as this process of interfacing is indeed in the scope of the system (i.e. integration phase). Elements in grey are not detailed since they do have their correspondent subsection within this document. For the sake of simplicity, only components connected immediately among them will be considered for each class diagram.

4.2.1. GUI Component

The GUI component (i.e. *Graphical User Interface*) is in charge of all the graphical interaction with the user. This component is divided into two differentiated parts in the analysis phase, as two different situations can be inferred from its usage, being namely:

- GUI elements which are inherent to the Framework.
- GUI elements which are created on the User Plugin demand.

This two cases are not equal in terms of design, and they are going to be implemented differently.

4.2.1.1 Monocle GUI

This element is in charge of managing the framework graphical elements (i.e. does not change among different user plugins). The main GUI window will be intended to provide the user with the different options available in order to perform a complete analysis. The user will be able to choose which plugin to run, which evidence source to use, and additional parameters which might depend on the plugin type, such as the Volatility Framework profile to use. The technology proposed for this component is Tkinter, a Python Library very well suited for simple graphical interfaces, as described in subsubsection 3.5.4. Tkinter is in charge to create additional frames for file path selection and OS-related activities.

The GUI component will be in charge of launching the analysis on user selection by calling the *Monocle* component. This will launch a command-line order to the aforementioned component in order to start the computation in a separated process. The GUI will be informed when this process is finished in order to graphically notify the user about it.

Figure 7 depicts the components and relations with other components of the MonocleGUI component.

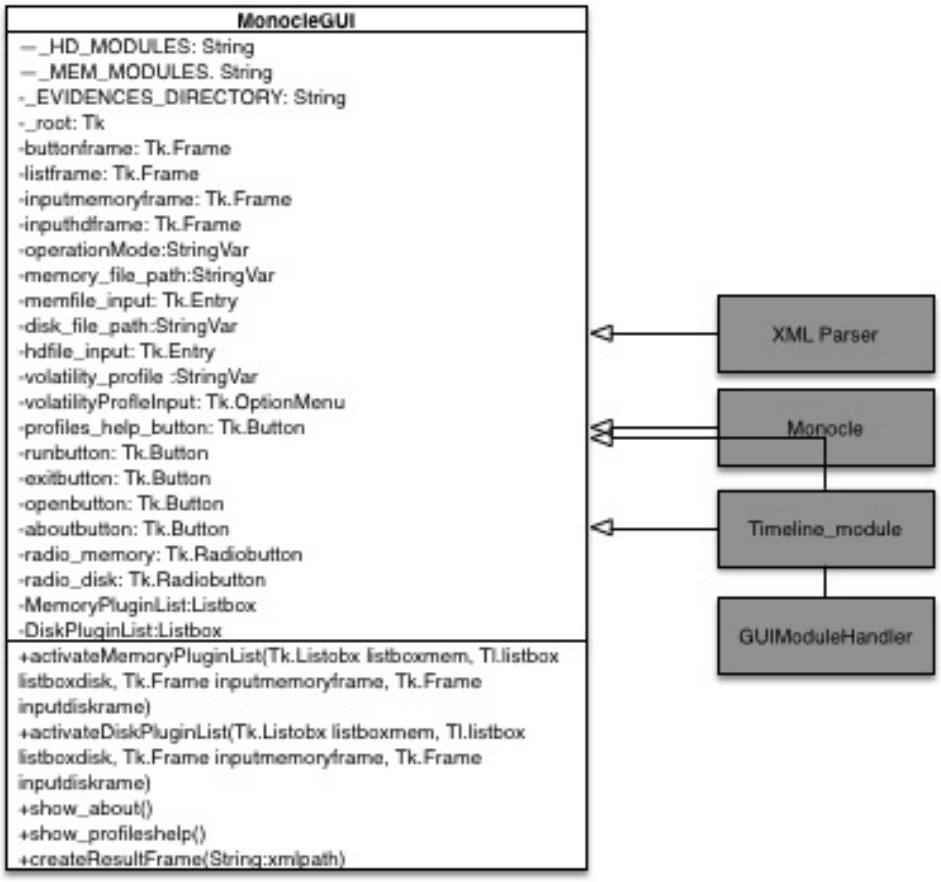

Figure 7: MonocleGUI Class Diagram

4.2.1.2 GUIModuleHandler

This sub component is the one in charge of managing the GUI requests which do depend on the user plugin being executed. As GUI components usually have their own dedicated thread of execution [54], this component will be managed by means of an asynchronous process queue in order to be able to manage requests coming from the user plugins in a reactive way. Each new window to create will do have an associated ID (i.e. represented with an integer). This ID will trigger the corresponding creation function, which must be included in the framework from beforehand. In this way, there is an isolation between the main, immutable graphical components and the user-dependent ones.

Figure 8 represents the class diagram of the aforementioned component and the relationship it does have with other classes.

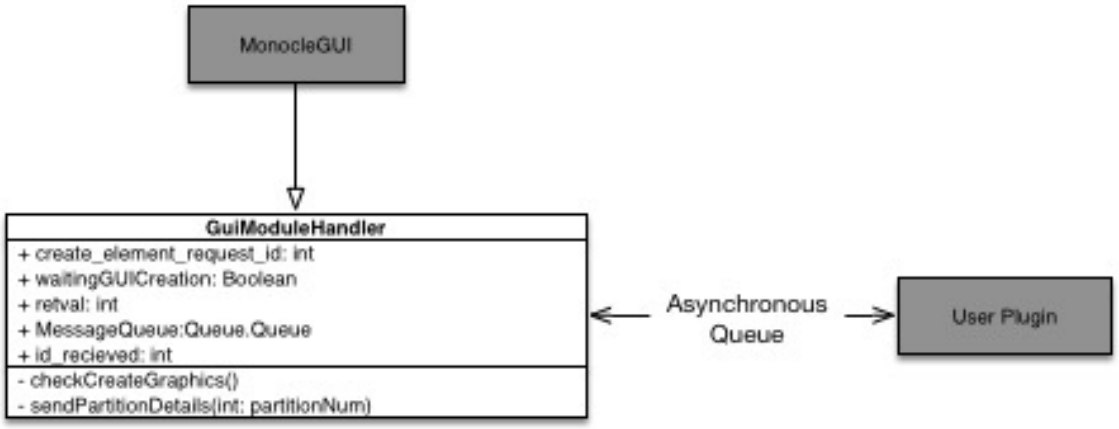

Figure 8: GUIModuleHandler Class Diagram

4.2.2. Monocle Component

The Monocle Component is the main entry point to the Framework itself. It is in charge of managing the different parameters coming from the GUI which will model the analysis behavior to be launched (i.e. which module is to be used, where to store the persistent data coming from the analysis and so on). This component will receive the parameters selected by the user in the GUI and launch the appropriate operations. This component is called from the command line with the proper parameters (i.e. and usually from the MonocleGUI component). Notice that this implies **the framework can be launched from command line without the usage of any GUI component**. This component will also include a parser for checking the arguments used, which will be considered as a different class in order to simplify the design and to enhance modularity.

Figure 9 shows the UML class diagram of the Monocle component, along with its interactions with other components and the parameters required to do so.

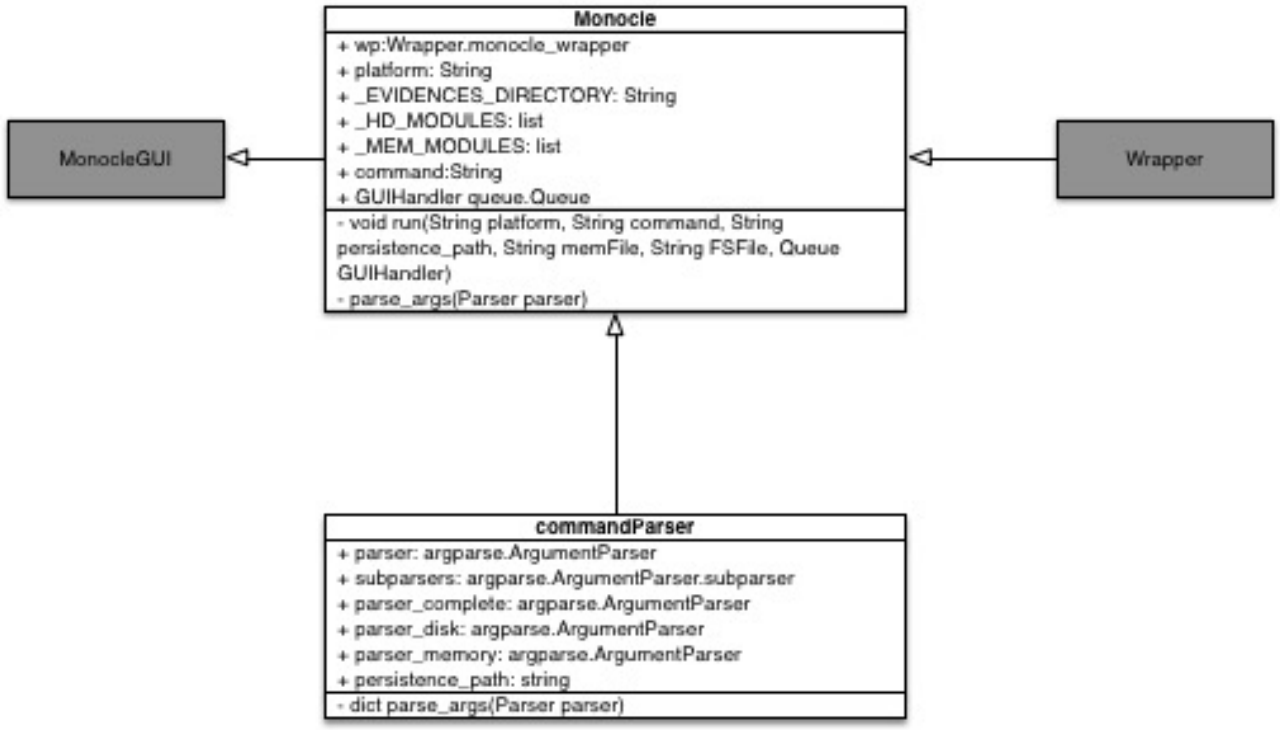

Figure 9: Monocle Class Diagram

4.2.3. Wrapper Component

The wrapper is the component in charge of instantiating and loading the different user plugins to be used, and to start the actual program flow once the analysis target and the plugin are chosen. It will initialize the different tools and components depending on the analysis type (i.e. memory or disk), and it will collect the evidences yielded during the plugin execution in order to pass them to the XML parser for persistence. Figure 10 shows the UML Class diagram corresponding to the aforementioned component.
As a design decision, this component will differentiate whether the plugin to execute is of type disk or memory by the plugin location. Disk modules will be searched in *diskModules* folder, whether memory ones will be under *memoryModules*.

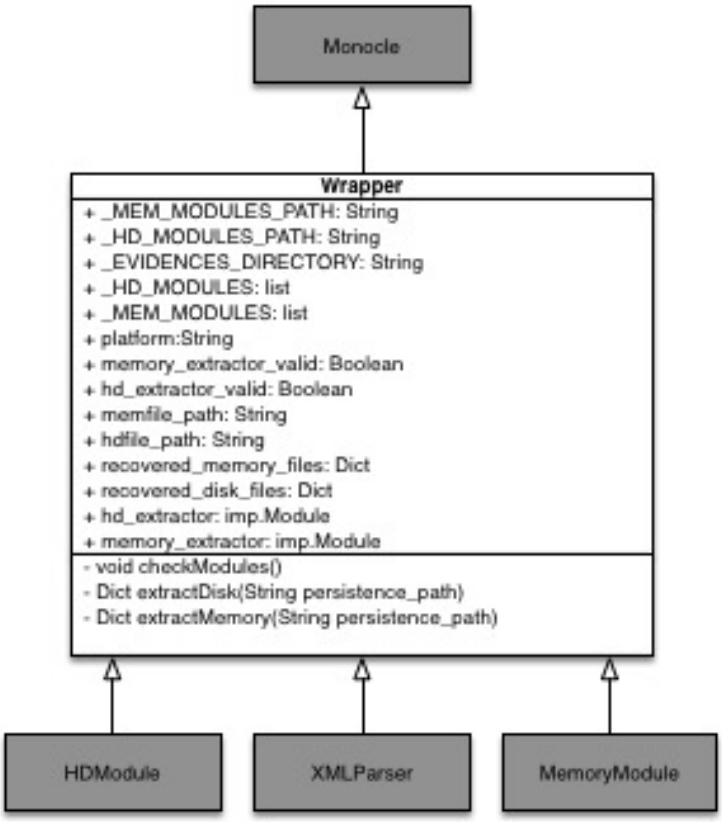

Figure 10: Wrapper Class Diagram

4.2.4. XML Parser Component

The XML parser component is the component in charge of implementing the persistance of the results retrieved from the analysis. This results will be yielded by a user module and will be gathered by the wrapper. The wrapper will call the XML Parser in order to generate an appropriate XML stream, which will be then stored in a file within the previously specified evidence destination path. Figure 11 shows the UML Class diagram corresponding to the aforementioned component.

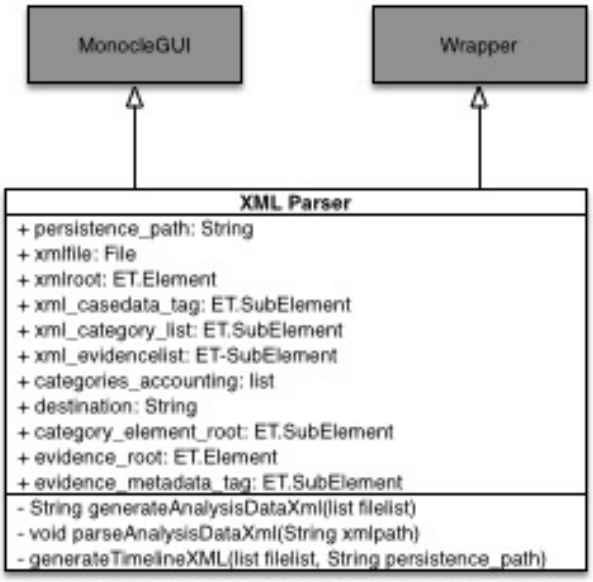

Figure 11: XML Parser Class Diagram

4.2.5. MemoryModule Component

The *Memory Module* component provides the system with an interface for loading and executing memory-targeted user plugins. This class acts as an abstract class, which provides the interface for loading memory-related elements and tools, such as the *VolActor* component. Figure 12 shows the UML Class diagram corresponding to the MemoryModule component.

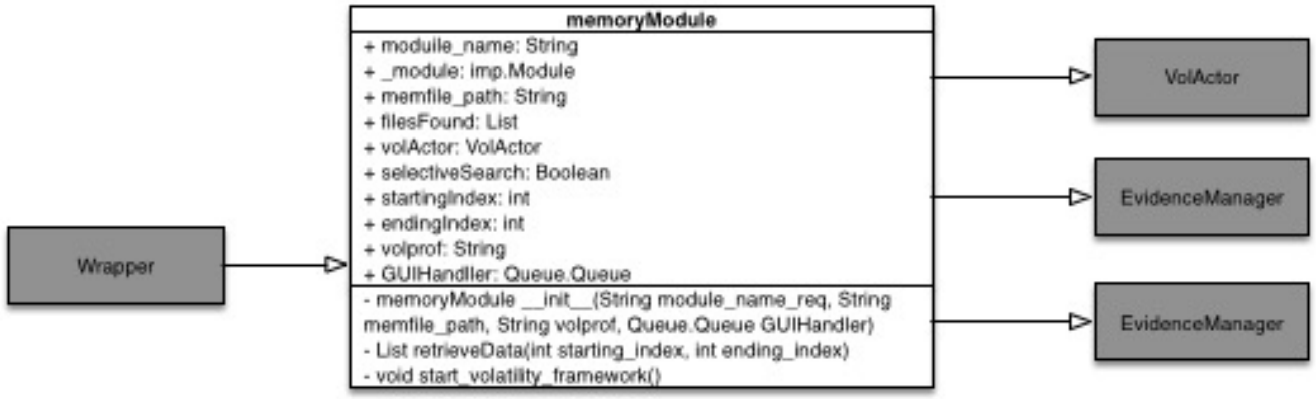

Figure 12: MemoryModule Class Diagram

4.2.6. HDModule Component

The *HD Module* component provides the system with an interface for loading and executing disk-targeted user plugins. This class acts as an abstract class, which provides the interface for loading disk-related elements and tools, such as the *RegistryActor* component. Figure 13 shows the UML Class diagram corresponding to the HDModule component.

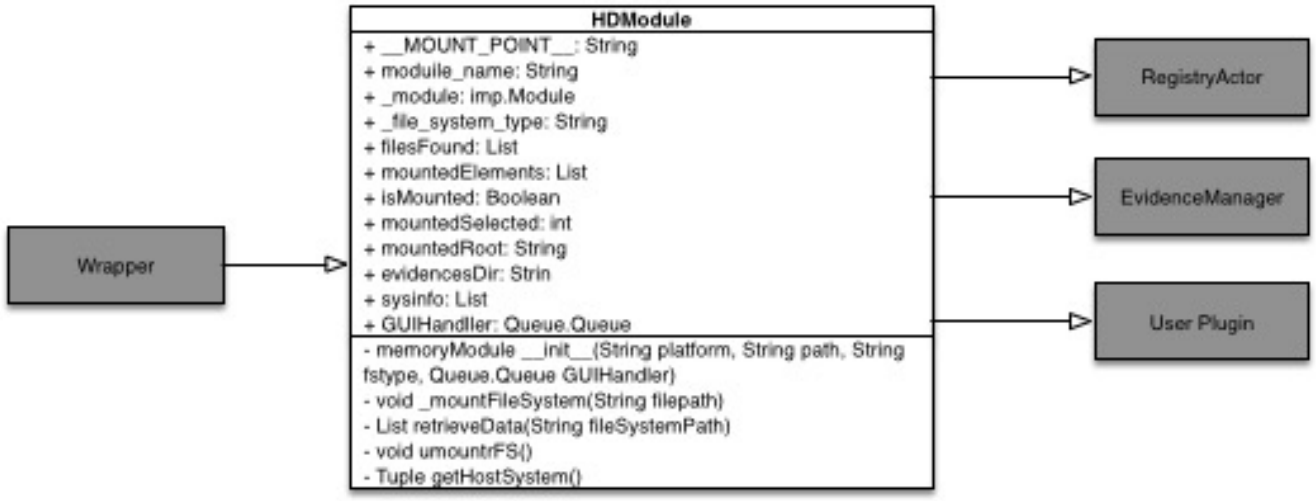

Figure 13: HDModule Class Diagram

4.2.7. TimelineModule Component

The *TimelineModule* component is the element in charge of displaying and launching the result of the analysis as a timeline graphical diagram. This tool interpret an special file type (i.e. generated by the XML parser component), and displays the aforementioned timeline. This component is composed mainly of the TimeLine python module (i.e. third party element), which will not be explained here as it is out of the design scope for the present project. Figure 14 shows the UML Class diagram corresponding to the TimeLine component.

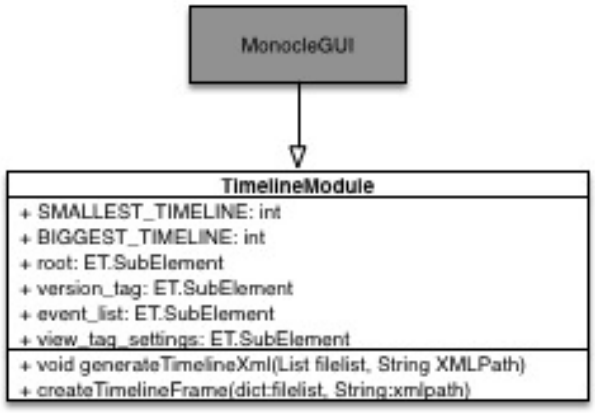

Figure 14: TimelineModule Class Diagram

4.2.8. VolActor Component

The *VolActor* component is the element in charge of interface, configure, execute and initialize the Volatility Framework within the system. This component provides several helper functions to deal with the aforementioned framework, and it does manage all the process of loading new Volatility plugins. This component is also responsible of executing the Volatility Framework *calculate* and *renderText* functions and to retrieve the result of those computations to the user plugin for further usage within the analysis procedure. This allows the user to employ the Volatility Framework with a minimum effort, as long as it is installed in the system. Figure 15 shows the UML Class diagram corresponding to the aforementioned component.

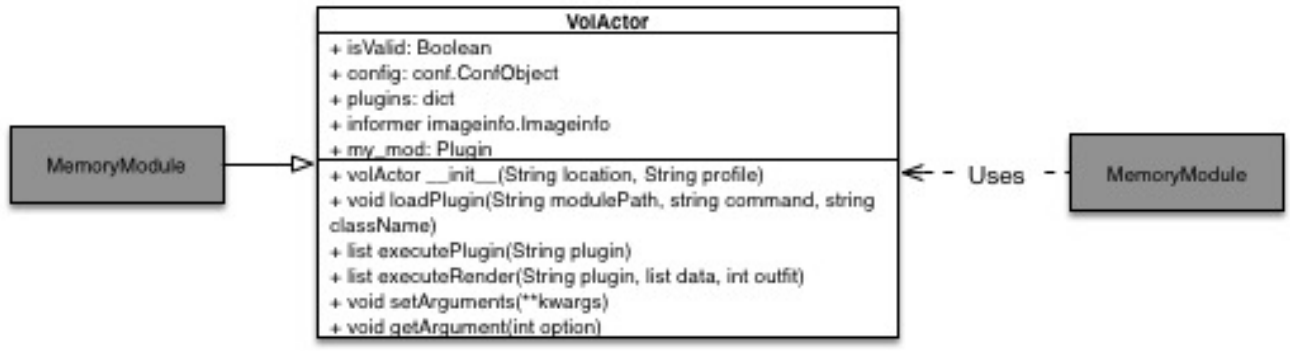

Figure 15: VolActor Class Diagram

4.2.9. EvidenceManager Component

The *EvidenceManager* component is the element in charge of dealing with the evidences yielded from a user analysis. It exposes functions in order to automatically hash and store the evidences within a location in the analyst machine. It is used by user plugins by creating Evidence instances for each evidence found during the analysis. Figure 16 shows the UML Class diagram corresponding to the aforementioned component.

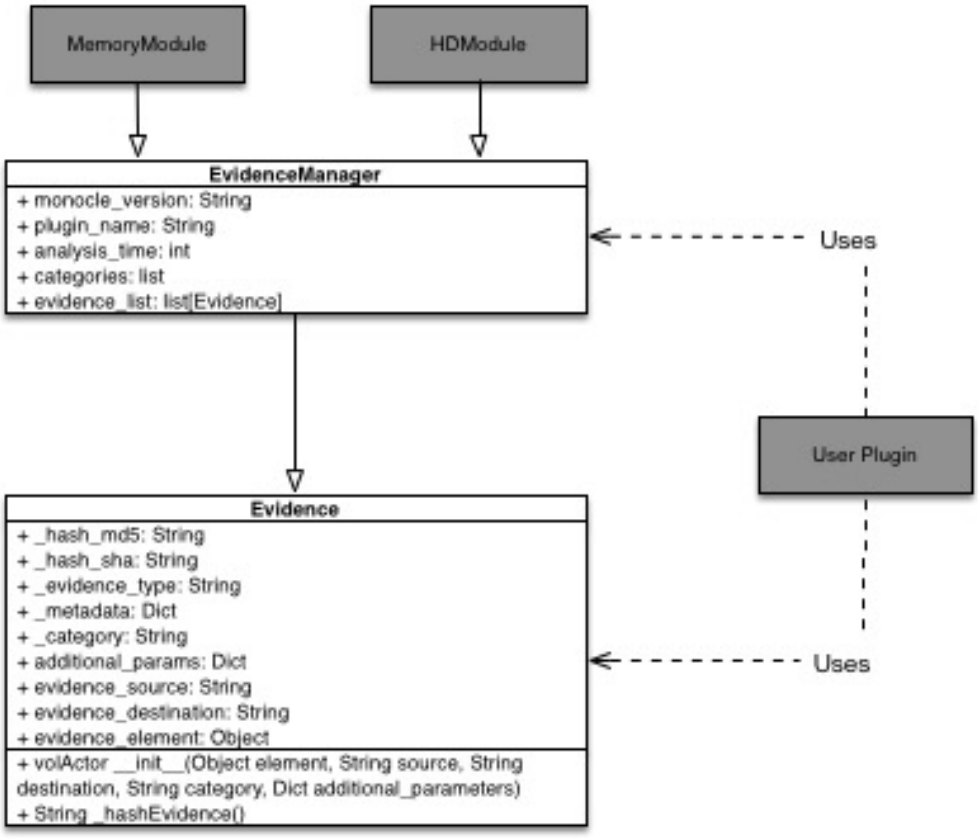

Figure 16: EvidenceManager Class Diagram

4.2.10. RegistryActor Component

The *RegistryActor* component is an element which provides an interface for the usage of the *Registry* Python plugin written by Will Balenthin. This component exposes several functions in order to read Windows registry hives, which can be imported by a user plugin in order to use those results in the analysis. Figure 17 shows the UML Class diagram corresponding to the RegistryActor component.

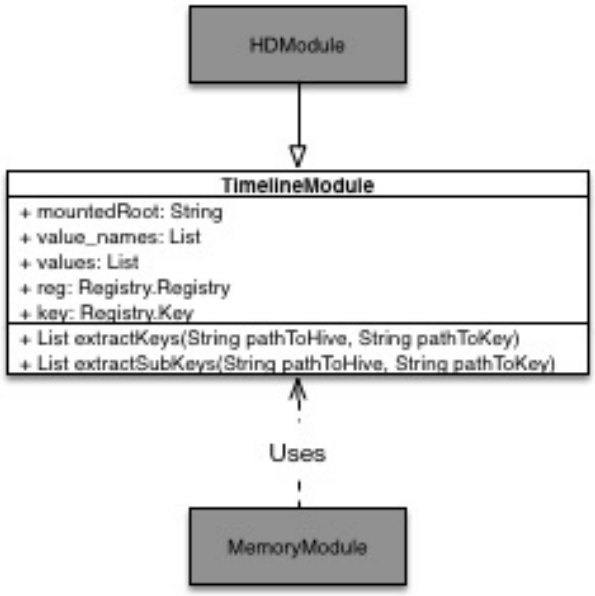

Figure 17: RegistryActor Class Diagram

SECTION 4.3

Sequence Diagrams Definition

This section is intended to the definition and formalization of the sequence diagrams related to each of the use cases proposed in subsection 3.8, page 65.

These diagrams will show properly the behavior the system must follow when interacting with the user in different ways, as well as it will make more clear the relationships among classes.

The UML notation is again the chosen one in order to represent such a scenario. UML Sequence diagrams are to be created in order to define the control flow sequence of the whole process.

The upper elements represent the different interacting objects. The actor will interact with such elements. Such interactions are defined by means of arrows. The following diagrams are to be read upside-down as the time dimension is represented in the vertical axis (i.e. the first action is the one placed closer to the top of the diagram).

4.3.1. UC-1 Plugin Inclusion

Use case 1 refers to the inclusion of a new plugin within the framework in order to detect it. Although the initial sequence of this use case is not interacting directly within the framework, it do so when the system has to recognize the plugin. Figure 18 shows the sequence diagram associated to this use case in particular.

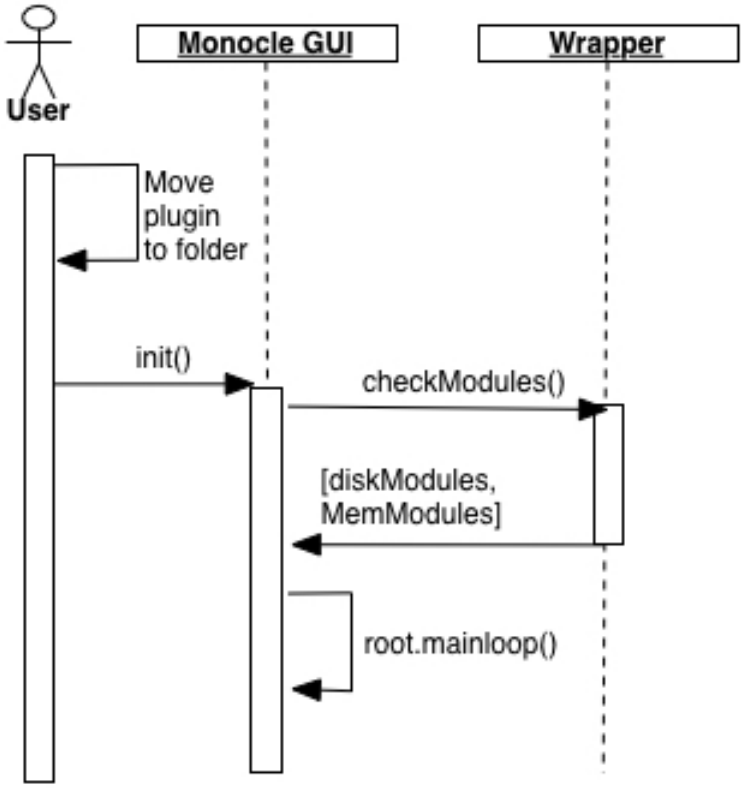

Figure 18: UC-1 Sequence Diagram

4.3.2. UC-2 Plugin Execution

Use case 2 refers to the execution of a plugin from scratch. This includes the GUI loading of the user plugins, as well as the dynamic loading of the plugin to be executed and the results generation from the analysis evidences. It has to be noticed that additional flows might appear depending on the nature of the plugin (i.e. either disk or memory) and the resources and tools the user plugin employs (e.g. volActor). Figure 19 shows the sequence diagram associated to this use case in particular.

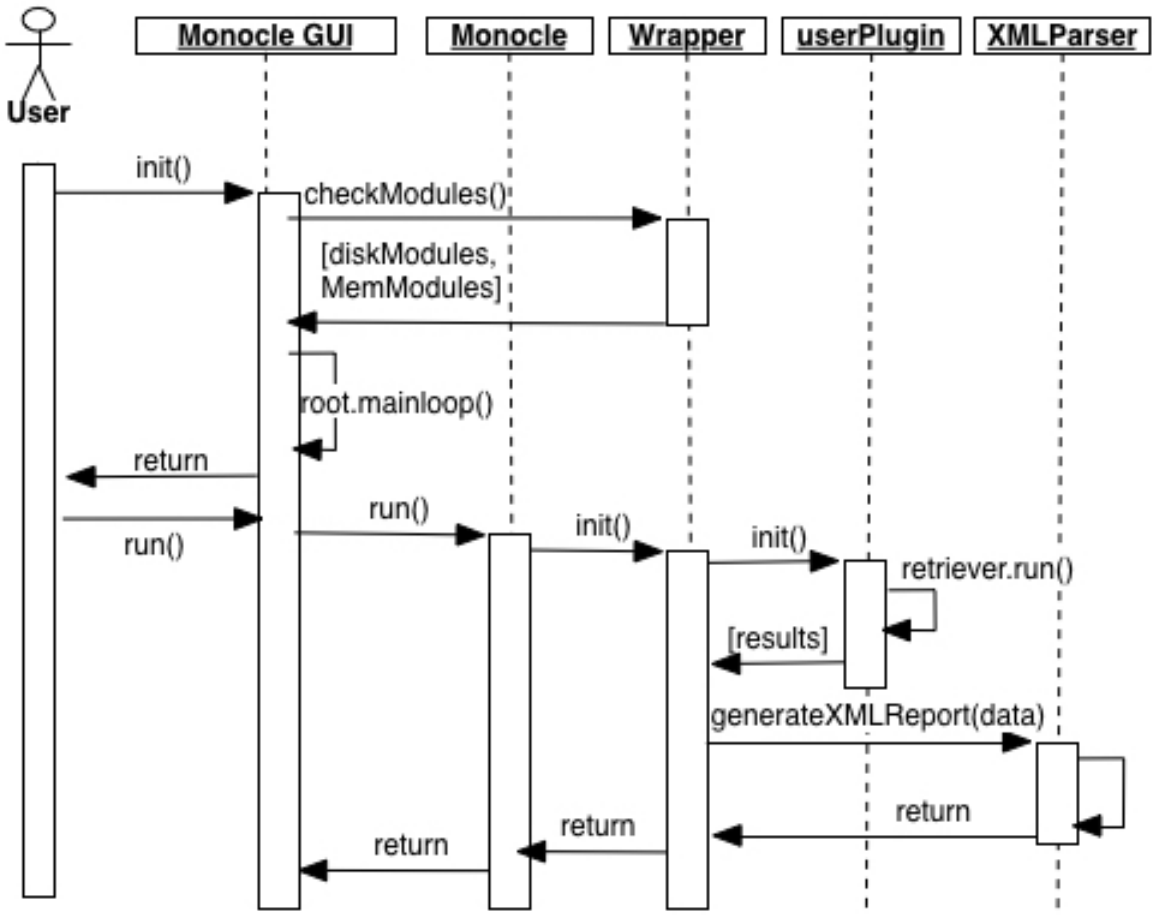

Figure 19: UC-2 Sequence Diagram

4.3.3. UC-3 Result Opening

Use case 3 refers to the loading of the persistent results coming from the XML file generated during the process in UC-2. This XML file is composed of data regarding the analysis (e.g. analysis time, framework version) as well as regarding the evidences found.

This use case is triggered automatically (i.e. without the user interaction) whenever an analysis is performed in order to open the XML file and show the user the results. This use case can however be started at any moment via the *open* button in the main GUI and specifying the appropriate XML result file.

Figure 20 shows the sequence diagram associated to this use case in particular, and it do start where UC-2 flow diagram in subsubsection 4.3.2 finishes (i.e. return from the *Monocle* component).

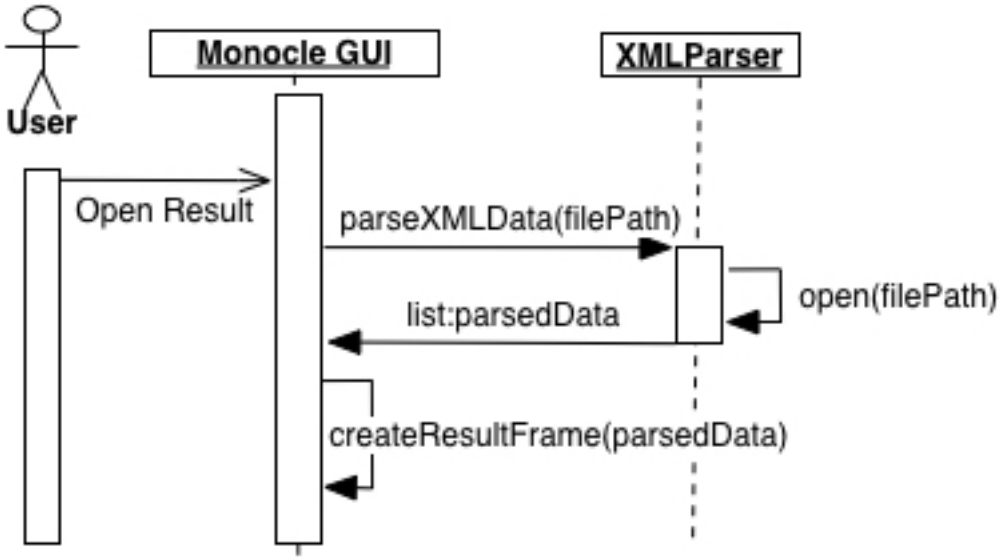

Figure 20: UC-3 Sequence Diagram

4.3.4. UC-3.1 Timeline generation

Use case 3.1 refers to the generation of the timeline window in case the user requires it. This process is accomplished by parsing the analysis data obtained from the XML to the XML parser again, in order for it to create an XML-like file which can be used by the *timeline* plugin. This XML file defines the area comprised by this timeline, as well as all the events shown on it.
Figure 21 shows the sequence diagram associated to this use case in particular, and it do start where UC-3 does in flow diagram in subsubsection 4.3.2 finishes (i.e. after creating the result frame).

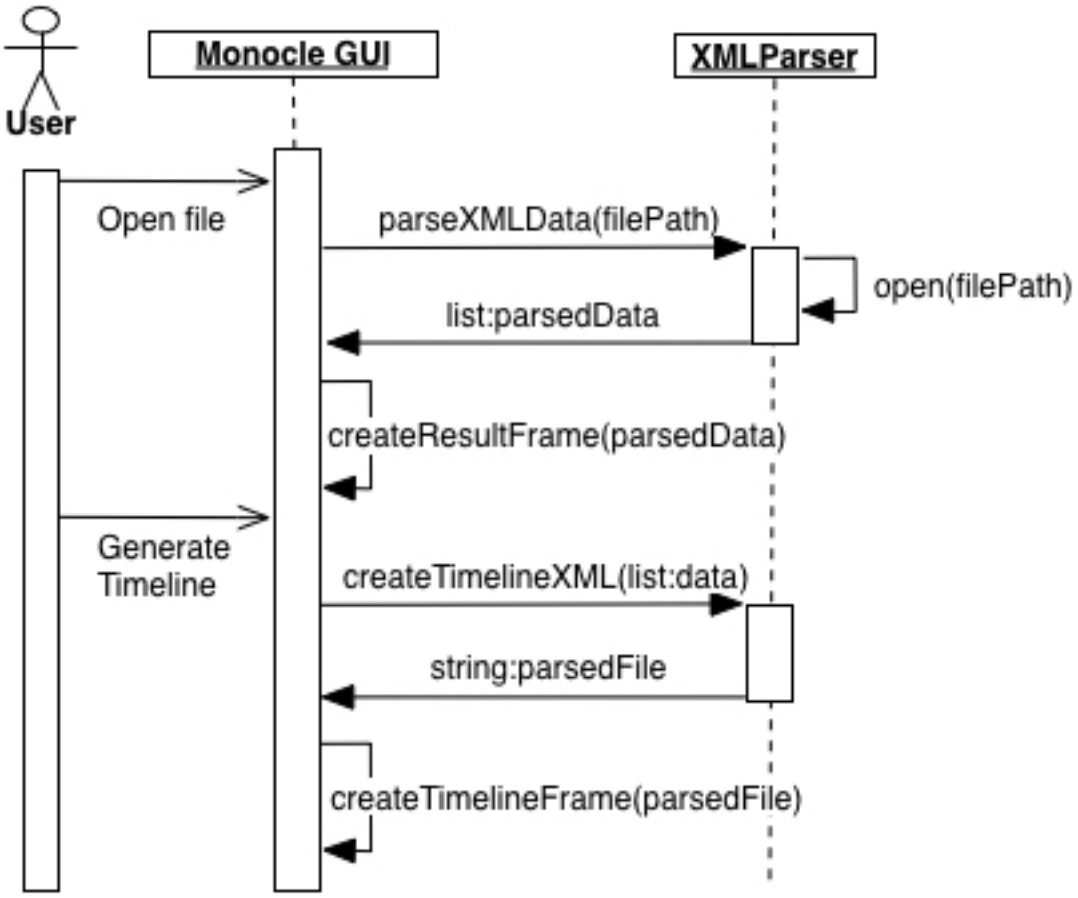

Figure 21: UC-3.1 Sequence Diagram

Chapter 5

Software Implementation

SECTION 5.1

Software Implementation Decisions

This objective of this section is to further explain the decisions taken during the development of the present software in terms of coding decisions after the analysis and the design has been performed. This decisions have a technical component intended to help the deployment of the project and the overall maintainability and functioning of the components integrated within the software.

5.1.1. Isolation between the GUI component and the core framework

The isolation existing among the GUI component and the core component is intended to serve a dual purpose within the development of the overall project.

Firstly and foremost, the total separation among this two components allows modularity. This means that the GUI and the core might change independently of the other element in order to enhance their performance or to add new capabilities without having to modify the other component. This modifications are not going to affect the relationship among the elements as long as the interface connecting it (i.e. the XML file with the persistent report) is not modified.

Secondly, this isolation among components allows the creation of a command-line tool which can be used without the need of a GUI. This is useful in case that this framework is included within a third-party application (i.e. as the Volatility Framework was included in the present project). This way we can provide a method for future work to integrate the existent functionality into a more complex software.

Figure 22 represents graphically the different interactions and the isolation existing between the aforementioned elements.

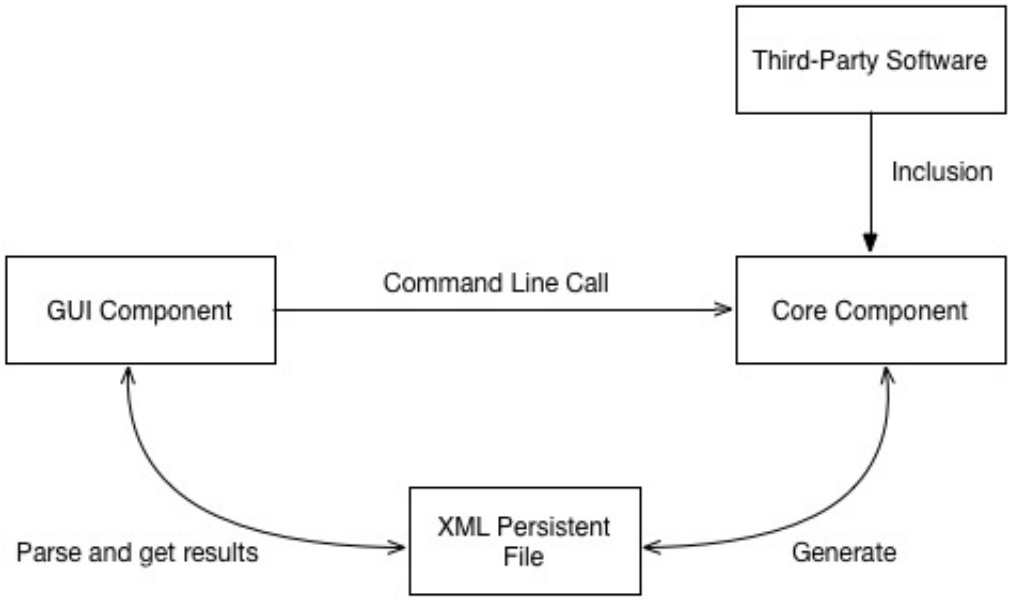

Figure 22: Isolation GUI-Core

5.1.2. Asynchronous message queue GUI-Module

In order to provide some extensibility to the GUI module with regards to the module being used, there must be a way for this two processes to communicate. This problem arises as the GUI components usually need a dedicated process of execution in order to run due to the fact that the main algorithm for GUI creation and update uses busy waiting in order to wait for change orders.

By implementing a queue between processes, it is possible for the GUI to check this queue periodically in order to detect if there are changes to be made. The user module can use this queue in order to request changes in the display and the behavior of the graphical display. This modifications must be implemented in the GUI component, and they shall have an ID number associated (i.e. the module uses a number in order to request the operation, and the GUI component behaves accordingly).

5.1.3. Auto scanning of Volatility Framework

The Volatility Framework is integrated (i.e. as described in subsubsection 4.2.8) within the VolActor component. As explained in the aforementioned section, this component will be in charge of initializing the environment necessary for the Volatility Framework to run. This include the profile to be used within the analysis, as well as the target of such operations.

An implementation decision was made here as there is **mandatory** for the initialization of such environment to specify a Volatility profile to run over. Thus, the user must introduce the corresponding volatility profile when calling the module (i.e. from the proper selection box in the GUI component). However, the user might not know which Volatility profile to use.

If the user do not know or do not include a volatility profile when calling the core component, the VolActor object will **automatically** launch a kdbg scan over the provided memory image in order to try to identify the type of image and the profile associated to it. The kdbgscan module of Volatility tries to do so by analyzing the kernel debugger block in mapped into memory and looking different attributes which are characteristic of a specific operating system and version.

SECTION 5.2

Software Integration Decisions

This section is intended to specify the decisions taken during the integration process of the different modules composing the final software. During this section, the different ways in which those components have been included will be described, as well as the different reasoning leading to the chosen procedure.

5.2.1. Integration of the Volatility Framework

The integration of the Volatility Framework is one of the most difficult yet more important additions to the present software. Due to this fact, it is necessary to be specially careful when deciding which components to expose and how to do so.

As mentioned in subsubsection 5.1.3, this integration will be done in the VolActor object, which will properly initialize the different settings required to run the environment properly. A decision must be taken here in order to study how the different Volatility Framework functions are going to be available for the user plugins.

Due to simplicity and for the sake of control within the actions performed by the user modules, it has been decided to interface the operations which interact with the Volatility Framework, as well as the process of modifying the current parameters operating over the Volatility environment.

The interfacing over the Volatility functions is done by means of the creation of wrapper functions which act over the *render* and *calculate* functions of Volatility. This functions are in charge of executing the different modules, and they need to know which module to launch and which parameters to use to do so. These wrapper functions are intended to be used by means of strings (i.e. the user will specify just the name of the plugin to be launched). These wrapper functions will provide also a way for inputting sets of data (i.e. user dependent, usually arrays of data) and the output file descriptor (i.e. by default the standard system output). This way the module does not need to deal with the plugins themselves, but only with the data sets to analyze.

The interfacing of the settings will be done by two different wrapper functions. One of them will be in charge of loading new plugins in the memory space of the Volatility framework.

The user will only need to know the name of the plugin and where is it located within the Volatility namespace (i.e. the category of the plugin and its name) in order to load the module. The other function will be composed of a setter and a getter wrapper, which allows the user to set parameters as key-value pairs, and to retrieve the value of an already existing parameter by knowing its key.

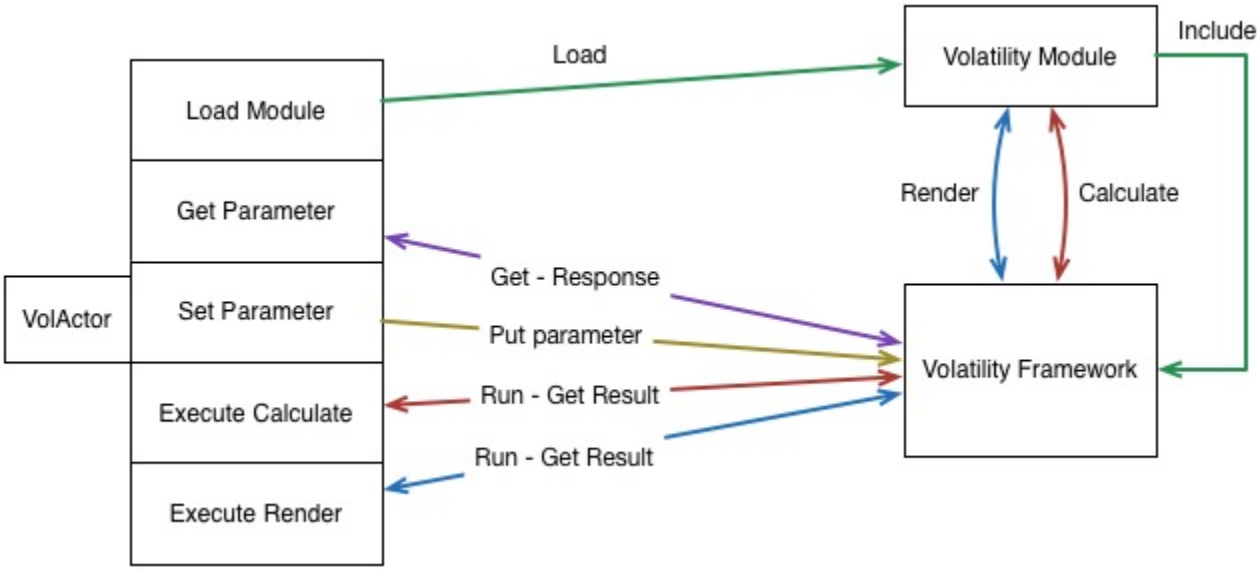

Figure 23: VolActor Wrapper relationship diagram

5.2.2. Integration of the Python Registry module

The Python Registry module is integrated within the HDModule component by means of the RegistryActor component as to provide easy access and parsing for the user to the Windows Registry files. This element is integrated as a Python library, and it has an already defined API. There is thus a design decision which must be taken here in order to decide how to expose this functionality to the user module.

The module provides functions to open (i.e. implementing the parsing) and to extract registry keys and subkeys. It has been decided however to wrap these functions into helper methods in order to avoid effort to the user and to provide transparent access within the analyzed target disk file (i.e. to parse a path relative to the disk to the proper absolute one with the evidence element mounted into the local file system).

This is done by means of *extractKeys* and *extractSubKeys* functions, which receive the path to a given Windows registry hive and the key to analyze. These functions automatically parse the path to a relative path whose root is the root of the mounted evidence disk. They also do open the registry hives and iterate over the results in order to return to the user and easy-to-use Python dictionary.

5.2.3. Integration Timeline module

The timeline module is in charge of generating a graphical representation of the results obtained by means of a user analysis module execution. The incorporation of this module is necessary in order to provide further visual feedback to the analyst. This feature is also desirable in order to further explain the results achieved in front of a court as the judge is usually not specialist in digital investigation procedures.

An special XML file is needed in order for this module to read from it (i.e. it do has a well-defined syntax). It has been however decided to use this module instead of creating the feature from scratch as it is easier to create such an XML special file than to implement a processor which generates the timeline based on the analysis persisted results. Thus, an special parser has to be created in order to translate the results in the XML persistent result file to the special file the timeline module needs in order to operate.

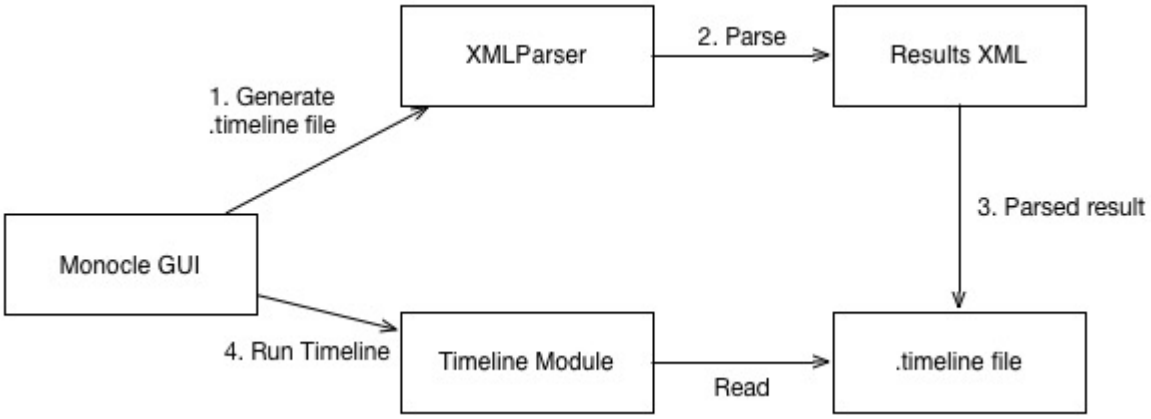

Figure 24: Timeline Usage procedure

SECTION 5.3

Acceptance Tests results

The present section is intended to show the results of the different tests proposed on subsection 3.9. These tests are intended to ensure the proper behavior of the system according to the requirements stated in subsection 3.7.

The system will be considered valid if the output of every single test is the one described when defining the test case. If the output differs in some way on what it is stated, the test is considered failed. If a test fails, the components being tested by such test must be reviewed in order to trace back the error to its source and to correct them.

After all the tests are executed, the system is either accepted, accepted with corrections to be made, or rejected. The execution of the tests over the final system results in the table presented in Table 14. It is appreciable that all test requirements are fulfilled, which implies the system is **acceptable** within the tests cases proposed and thus, **valid** for the requirements stated.

Acceptance tests result		
Test ID	**Requirements Tested**	**Result**
AT-01	F-07, F-08, OP-03, OP-05, P-05	Passed
AT-02	F-01, F-02, F-03, I-06, I-07, OP-03, OP-04, OP-09, S-07, OP-12	Passed
AT-03	F-01, F-02, F-04, I-03, I-04, I-07, OP-03, OP-04, OP-09, I-03, S-07, OP-12	Passed
AT-04	AT-02 + F-05, F-06, F-09, S-04, S-05	Passed
AT-05	AT-03 + F-05, F-06, F-09, S-04, S-05	Passed
AT-06	F-09, I-07, OP-03, OP-08, OP-09	Passed
AT-07	AT-04 + F-10, F-11, I-01, OP-11	Passed
AT-08	AT-05 + F-12, I-02	Passed
AT-09	F-10, F-13, F-14, OP-03, OP-10	Passed
AT-10	F-01, F-02, F-03, I-06, I-07, S-07, OP-12, OP-01, OP-02	Passed

Table 14: Acceptance Tests Results

Chapter 6

Evaluation in cloud scenarios

SECTION 6.1

User-side cloud scenarios analysis

The present section is intended to test and measure the effectiveness of the proposed framework with respect to a practical case scenario. Due to its interest in terms of research and documentation (i.e. as described on section 2), the scenario proposed will be a user-side cloud scenario in which the interaction of the user within different cloud storage providers will be analyzed.

Two different cloud platforms will be evaluated in order to prove the suitability of the framework with respect to the scenario, namely iCloud [6], from Apple inc. and the cloud service of Box [7]. Two different modules will be developed for each scenario, targeting the RAM memory and the disk remnants for each of the aforementioned platforms.

6.1.1. Definition of the environment

Both analysis scenarios will be conducted under a Windows 7 system as this is currently one of the most widely used operating systems. No additional software will be installed in the machines but the different web browsers to be used in order to test the web clients of the different platforms and the installation of the platform third party client. Accounts for the different platforms will be created outside the virtual machines, limiting the interaction of the user to a single connection to the service in order to either view the files or download and install the client.

The tests will be performed over different dumps obtained from third party machines. In order to reduce costs in equipment, such machines will be emulated by means of Virtual Machines. From each virtual machine, the RAM memory and the virtual hard disk will be analyzed in each of the proposed scenarios. Such dumps will be hashed and securely stored in order to preserve data integrity.
For the proposed scenario, the user has two .jpg images (namely *imagejrc1.jpg* and *imagejrc2.jpg*, as well as a .rtf file, namely *TFG_GUIDELINES.rtf*.

6.1.2. Study of the evidences to be found

A similar procedure as the one used in [25], [27], [26] will be used in order to ensure a proper location of the evidences and the isolation of the systems as not to pollute the evidences to find.

Cloud storage systems on user-side scenarios can be accessed traditionally in two ways, namely by means of a web browser to the provided web interface, and by making use of a local client which will synchronize the contents of our local file system to the ones contained in the remote system.

This exposes a series of evidence sources which can be potentially meaningful for the investigation. In order to categorize such evidences, we will differentiate three different levels of relationship of a user within the cloud system, described as follows:

1. No interaction with the system. The machine is a clean raw machine where no cloud system has ever been used
2. Access to the system by means of a web browser. This is a light relationship with the cloud system as the data remnants which can be found will be contained within the web browser history and RAM memory, among others.
3. Usage of the cloud system sync program. This is a heavy relationship with the cloud system, as the data remnants which can be found will not only be potentially the same ones as the previous case, but also will contain extra software modifications and processes in the disk drive belonging to the cloud system itself (i.e. and not only to the web browser, as in case 2).

By studying the different virtual machines, a set of evidences can be manually found in order to prepare the scripts which are to be created for further analysis of the given systems. The evidences which can be found for each of the stages above will be divided as follows. Notice that higher levels of involvement will also potentially include the previous levels data remnants.

1. No interaction with the system.
    - There will be no potential data to analyze (i.e. the user had no relation with any cloud system).

2. Access to the system by means of a web browser.
    - There might be a web browser running as a process which can contain data in memory like URLs or cached webpages related to the cloud service
    - There might be data stored in the registry hives related to URLs or search terms related to the cloud service.
    - There might be data stored within the temporal data folders of the web browser with the URLs and temporal files contained in the webpage, such as images, text or scripts.
3. Usage of the cloud system synchronization program.
    - There might be data in the registry hives regarding the installation of the sync program.
    - This sort of programs usually have a syncing location where all the synced files are stored.
    - Information regarding the sync program can be found in specific hives of the registry, such as the sync folder location or the installation path.
    - Such programs usually run in the background, so there might be a process still running in memory. If so, this process will comprehend valuable information which can be valuable if analyzed (i.e. files, paths, connections...)
    - The installation path of the client will contain the configuration files regarding the synchronization and other relevant user parameters.

All this potential sources of evidences must be taken in order to develop the plugins which will be used in order to check the framework suitability and to study the aforementioned scenario. The proposed objective of such plugins will be to prove the interaction between the user and the cloud service, and to retrieve as much information as possible regarding the level of involvement (i.e. downloads, visits) as well as the data contained within the framework (e.g. images, text files).

SECTION 6.2

Design of the analysis: Box Cloud

The first platform to analyze will be Box Cloud. The source evidences previously stated will be investigated in order to retrieve as much information as possible with a single script. According to the design of the framework defined in section 4, two different plugins will be designed and implemented in order to perform this analysis, targeting the memory dump and the hard disk evidences respectively.

The present section will depict the procedure followed in order to state where the evidences to look for are located, as well as their format and nature. This procedure will have as target the potential sources identified in subsubsection 6.1.2. A prior design of the plugin will be then stated based on the explicit elements found. In order for the detailed design of such plugins, please refer to section 5.

6.2.1. Memory artifacts

As defined in subsubsection 6.1.2, one of the main elements to be found here are remnants of relevant URLs which might have been visited by the user. The most logical way in order to perform this search is to reconstruct the addressable memory space assigned to the web browser itself, where this information can be found. This process is however dependent on the fact that the process must be running at the time the memory dump was taken. A simpler approach would be to just read the raw memory dump bytes and parsing them to ASCII characters until a relevant URL is spotted. This approach presents the drawback that the RAM memory is highly fragmented and non allocated space cannot be rebuild easily.

As the Volatility framework is to be included among the usable tools of the framework, we can use it in order to analyze the contents of the memory assigned to the browser process in the first case, and we can perform a raw read of bytes in the memory in the second. Box cloud service URLs have a fixed starting pattern of the form $https://app.box.com$. The existence of this sole URL is enough indicator to state that there were a minor relation of the user within the platform. This relation of course will be stronger the higher the number of this sort of URL found when performing the analysis.

This URL can also be found in the registry hives storing the recent URL visited. The extraction of them can be done by means of the Volatility Framework too, as it does include

a plugin to retrieve registry hives mapped into memory.

This URLs are however meaningful by their sole structure. By studying the structure of Box web cloud service, it has been discovered that all files which can be previewed in the service have a linked URL of the form $https://app.box.com/files/0/f/0/1/f\_[ID]$, where ID is a numerical value assigned by Box to the stored files. With this information it is possible to additionally search for this identifiers within the memory dump in order to discover more about such files.

The prior analysis reveals that the file metadata stored within the platform is sent to the browser by means of a JSON structure. This structure has several meaningful fields, whose meaning will be explained in Table 15. By collecting all this metadata and relate it to each file it is possible to extract further useful information which can be combined with the result of further analysis.

Box hosted files metadata	
Metadata Tag	**Description**
user_id	Numerical value assigned by the platform to each registered user. The value represents the user viewing the file at the moment this data is requested.
owner	The owner of the file referenced. This is the user hosting the file within the cloud service.
name	Name of the linked file.
raw_size	Size in bytes of the linked file.
raw_date	The EPOCH date where the file was uploaded or created in the cloud platform
content_type	The kind of data the file represents. Experiments show it is not reliable as it guesses the file by its liked extension and not by means of checking the content
sha1	The SHA1 message digest of the file. Used with integrity purposes.
last_updated_by	Registered name for the last user who edited the file in the cloud platform. It is linked with parameter last_updated_by_user_id.
last_updated_by_user_id	Similar to the *user* field, this field represents the numerical value assigned by the platform to the user who last updated the file.
created_by	Registered name for the user who uploaded/created the file in the cloud platform. It is linked with parameter created_by_user_id.

Box hosted files metadata	
Metadata Tag	**Description**
created_by_user_id	Similar to the *user* field, this field represents the numerical value assigned by the platform to the user who created/uploaded the file.
pic_l	Link of the picture within the cloud platform (i.e. suffix to the root URL).
image	Name of the image within the local temporal folder of the web browser.

Table 15: Metadata related to Box hosted files

With all this metadata, it is possible to complement and to get further knowledge of the relationship of the user within the cloud system. It is also possible to request the cloud platform specific information regarding the user as its identifier and specific interactions are known (i.e. such as the user identifier within the platform).

6.2.2. Hard disk artifacts

Disk artifacts are the biggest potential source of data of user information and interactions with the cloud system. Such artifacts comprehend hosted disk files (i.e. images, text files...) as well as registry hives and temporal cached elements.

The study of the artifacts in memory already uncovered some elements present in the disk dump which are relevant to the investigation. As part of the metadata of the files previously extracted from the memory dump, it is interesting to look at the *image* tag, which is the one representing the name of the temporal file cached in the web browser temporal folder. The concatenation of this name with the file extension will give us a file name whose referred file can be retrieved from the aforementioned temporal folders. This way we can provide not only metadata related to the files found, but also the files themselves.

The registry hives are also a potential source of information here. The main difference between their extraction here or from the memory is the fact that they do not need to be mapped to memory in order to be found. This allows for a more detailed search over the different relevant keys for the investigation as all of them will be available as long as the registry hive is in the disk.

Registry key entries of particular interest for the Box cloud service are the following

- **SOFTWARE/Box/BoxSync/InstallID**. This key will show the existence or not of the synchronization client in the system. If it exists, it will also unveil the ID of the registered application within the system.
- **SOFTWARE/Box/BoxSync/InstallID/InstallPath**. This value will show where the box sync application was installed within the operating system. It will be useful in a future in order to retrieve data from such installation folder.
- **NTUSER/Software/Microsoft Internet Explorer/TypedURLs**. This will contain the URL typed by the user in Internet Explorer. The URL of box might be found here. A similar path can be used in order to access others browser history
- **NTUSER/Software/Microsoft/Windows NT/CurrentVersion/ AppCompatFlags/Compatibility Assistant/Persisted**. This registry hive represents a log of the downloaded files of the browser. The box sync installer can be found here if it has been downloaded.

Another source of information might be found in the Windows log files. Those files (i.e. having the extension .evtx) register the operations performed in the computer in order to track changes and other actions. The studying of these logs will surely reveal the installation or download of certain elements related to the cloud system (e.g. the installation of the synchronization app will surely be registered here).

The program folder whose path was discovered during the analysis of the registry is also a potential source of information. The analysis of such folder unveiled the fact that there are sqlite databases apart from the executable files. Such databases can be easily read and carved in order to discover information like the metadata unveiled in the memory analysis phase (i.e. allows correlation of the data) and other configuration parameters regarding the user account within the cloud system.

This folder also contains log files which Box creates automatically in order to track its functioning. The analysis of this logs is extremely valuable in order to track the dates and other meaningful data (i.e. synchronization dates, connections to the service and more).

Finally, it is also interesting to carve the contents of the synchronization folder as to retrieve the data the user has directly manipulated. The path of this folder can be extracted from many of the previously analyzed sources (e.g. the Box databases or the registry hives). Once the path is known, it is only necessary to acquire the contents of the selected folder in a recursive way.

SECTION 6.3

Design of the analysis: iCloud Drive

The second platform to analyze will be iCloud. The source evidences will be investigated in order to retrieve as much information as possible with a single script. According to the design of the framework defined in section 4, two different plugins will be designed and implemented in order to perform this analysis, targeting the memory dump and the hard disk evidences respectively.

The present section will depict the procedure followed in order to state where the evidences to look for are located, as well as their format and nature. This procedure will have as target the potential sources identified in subsubsection 6.1.2. A prior design of the plugin will be then stated based on the explicit elements found. In order for the detailed design of such plugins, please refer to section 5.

6.3.1. Memory Artifacts

As in the previous case, there is data which can be extracted from the address space assigned in memory to the web browser process in order to find mapped files or URLs related to iCloud service. The same carving approach of reading and parsing the raw bytes might also be followed if the process is not among the ones mapped in the memory dump.

The first thing noticed when analyzing iCloud is that the discovering of the parameters and behavior of the cloud platform is quite more difficult to know than it was for Box. Parameters are usually obfuscated in order to avoid reverse engineering, and there are more security components than the ones appearing in Box.

For example, in order to access elements through the iCloud web interface, an authorization token has to be requested for each single file. This authorization token will refer to a file within the system, and it will last for a given amount of time exposed. After the time expires, the token will be no longer valid and a new authorization will be necessary in order to access the file.

The following table shows the parameters which were found as meaningful when conducting the analysis of the cloud platform. Those parameters and their structure are sufficient in order to identify them within the requests performed

iCloud hosted files metadata	
Metadata Tag	**Description**
hs	Value identifying the user within the cloud service. It is constant for the files belonging to the same iCloud user
k	ID of the file within the iCloud drive service. This ID remains the same for the same file and is used in order to identify it within the cloud service.
e	Time the authorization to see the file on the web link is valid. It lasts for one hour since the request of the file.

Table 16: Metadata related to iCloud hosted files

6.3.2. Hard disk artifacts

Although there were some information in the analysis of memory regarding iCloud drive, this was not as meaningful for the analysis as in the previous case where Box cloud was being analyzed. The biggest amount of information will be located though in the hard drive, as there will be again plenty of elements to be retrieved as evidences of the usage of the cloud service.

The first thing to check is again the Windows registry. This will potentially unveil many elements which are relevant to further searching within the platform, like the existence of iCloud sync app, or the location of executable files and downloads among others. The following are a set of registry keys which are to be checked.

- **SOFTWARE/Classes/AppID/iCloudServices.EXE**. This registry entry provides the registered ID for the iCloud application within the system. Its existence is a proof that this element has been used.
- **SOFTWARE/Classes/iCloudServices.AccountInfo**. Provides further mapping for other iCloud keys within the registry.
- **SOFTWARE/Classes/Wow6432Node/AppID/iCloudServices.EXE**. An alternative path to find the AppID of the iCloud sync program.
- **NTUSER/Software/Microsoft/Internet Explorer/TypedURLs**. As in the previous case for Box drive, this registry key will unveil the typed URLs in Internet Ex-

plorer web browser. Similar registry keys shall be checked depending on the browsers installed within the system.

- **NTUSER/Software/Microsoft/Windows NT/CurrentVersion/AppCompat Flags/Compatibility Assistant/Persisted**. This registry key stores the downloads performed by Internet Explorer and their path. This can show whether the iCloud installer was downloaded or not (i.e. *icloudsetup.exe*).

With the information obtained from the registry, it is possible to go more in deep within the analysis. The synchronization program used by iCloud will be usually located under */Program Files (x86)/Common Files/Apple/Internet Services*. The existence of this program is a strong evidence of iCloud usage by the user, and within this path the different libraries and information regarding the program can be found.

The synchronization program also makes use of the AppData (*<user>/AppData/Local/Apple Inc/iCloudDrive/*) folder in order to store configuration parameters and other useful information. It is of particular interest the existence here of the SQLite databases used by iCloud as they do provide different parameters regarding the user account, such as the details of the account or the elements synchronized within the cloud server.

Additionally, the default location for the synchronization folder will be within the */users/<user>/iCloudDrive* folder, being this a good source for file extraction as it does contain all the files which were synchronized with the cloud platform at the moment of the disk dump acquisition.

Finally, the Windows Log files can be also retrieved as they might provide information regarding the installation of the program, the web browsing history and other elements which can be either discovered or reaffirmed by comparing them to the evidences in such logs.

SECTION 6.4

Implementation of user-side cloud analysis plugins

The process described in section 6 can be implemented easily by making use of the tools integrated within the present framework. This tools and modules will make the process of coding such analysis much easier and more efficient.

The elements to recover as evidences in the two aforementioned cases (i.e. subsection 6.2 and subsection 6.3) are quite similar with respect to their origin and location (e.g. registry key values, downloaded files, memory-stored URLs...). Due to this fact, the process of implementing the plugins for such scenarios will be treated as the same one for both (i.e. the tools used and how do they behave will be explained).

Notice that there will be necessary however to look for the specific elements as they do appear named and analyzed in section 6, as this section provides the particular elements which are to be found.

As defined in the specifications of the present software, the plugins will be divided in two for each platform, namely the plugins which will analyze the memory and the ones which will target the hard drive. The elements to use will differ depending on the type of module being coded, so both of them will be described within this section.

6.4.1. Implementation of the Memory Modules

As described previously during the analysis of the scenario to be investigated, the memory is one of the most difficult evidence sources to be retrieved as the allocated space related to each process is usually segmented into non-contiguous segments of data. This makes the raw reading of the memory difficult and not as accurate as it would be done over a linearly-allocated memory segment.

Is due to this fact that the memory plugins will make use of the Volatility Framework in order to better analyze and reconstruct the memory. The different plugins of Volatility will thus be available in order to retrieve the information of the memory.

Two approaches are to be followed here depending on the existence or not of a running web browser process (i.e. as explained in section 6).

The first one will be assuming there is a web browser process running. The names of the most common web browsers will be checked against the results coming from the *pscan* Volatility plugin, which lists the processes which were running in the machine when the memory dump was acquired. More precisely, *iexplorer.exe* (Internet Explorer), *firefox.exe* (Mozilla Firefox) and *chrome.exe* (Google Chrome) will be the ones to search for.

Once the processes running are known and filtered, it is the time to acquire the reconstructed memory allocated to such processes by means of Volatility's *MemDump* plugin. This will produce as many memory dump files as processes are running given certain filters (e.g. their name or PID). Once the memory dumps are created, a raw read of the contained bytes translated to UTF-8 encoding will reveal the possible URLs residing within such processes in a linear way. The URLs to search for were defined in section 6.

Additionally, and as the PID of the web-browser processes are known, it is possible to run other sets of analysis within such processes. Volatility comes with an internet explorer plugin (i.e. namely *IEHistory*) which lists the history of Internet Explorer. This can be used as an alternative way to retrieve such data if the Internet Explorer was used in order to access the cloud service. There are additionally two other plugins which can analyze in the same way Google Chrome [55] (i.e. namely *ChromeHistory*) and Mozilla Firefox [56] (i.e. namely *FirefoxHistory*). The inclusion of such plugins was done over the Volatility standard installation (i.e. not within our framework).

Within the files ID found in such histories and dumps, is it possible to retrieve the JSON structures existing in the memory for each file. This includes the elements defined in Table 15. After this data is created, the EvidenceManager (i.e. as defined in section 4) will be used in order to retrieve such elements. As those evidences are not fixed ones (i.e. they are not a file but a collection of key-value elements), it is necessary to create a string buffer to be parsed as evidence containing the data to be saved. The EvidenceManager will take care of storing it properly and to ensure the integrity checks.

The second approach to take would be mostly similar to the first one, but has to be conducted when there is no web browser processes running in the machine and such processes are not found by means of Volatility's PSScan. This will imply that we are not able to reconstruct the memory structures and thus, it is necessary to manually carve the entire memory dump in a byte-based approach in order to find the file IDs for the elements in box and other URLs, as well as the metadata. Once the results are found, the proceeding is the same as in the previous case by parsing such evidences to the EvidenceManager.

6.4.2. Implementation of the Disk Modules

The disk modules for both scenarios will be targeted to the Registry elements and the files/logs which can be found for each scenario as defined in section 6.

The first step to follow will be to mount the evidence element. This is automatically done by the framework in order to avoid further coding on the analyst side. The framework will automatically ask the user for the partition to mount. After this process is completed, it is time for the analysis process to start.

As explained in the aforementioned section, the first step will be to locate and process the Windows registry hives in order to extract as much information as necessary as to conduct the further stages of the evidence retrieval process. The location of the Windows Registry hives in Windows 7 is well known [57], so this registry analysis can be performed in a straightforward manner. There is however the need of retrieve the Users registered within the system (i.e. whose names are also stored in the registry under *SAM/Domains/Account/Users/Names*).

In order to retrieve such data (i.e. and all the other keys defined during subsubsection 6.3.2), the Registry module of the present framework will be used. As explained in section 4, this plugin makes use of the Python Registry library in order to parse the registry hives structures. The framework interfaces the capabilities of such elements in order to provide the user with straightforward functions in order the retrieve the Registry keys by just inputting the path to the registry hive and the desired key.

Once the Registry entries have been analyzed, it is the time to use the EvidenceManager in order to retrieve the elements pointed by such analysis. The path to each file or folder will be given to this tool and it will be properly stored and secured within the analyst machine.

In the cases where there is data to be carved from the recovered files (e.g. SQLite databases), a Python library will be used in order to perform queries over them. This data will be stored afterwards in the same way the memory elements were (i.e. as this evidences are of the key-value form).

SECTION 6.5

Performance Evaluation

Concerning previous scenarios (iCloud and Box), the overhead caused by Monocle at different points of its execution is measured. Monocle execution workflow is divided in three main phases. The first phase (wrapper pre-setup) consists of setting up Monocle's environment and mounting the evidence digital containers. The second phase (module execution) is the execution of the plugin. In the last phase (wrapper post-setup) the EvidenceManager processes evidences found in the second phase and presents the results to the user.

Table 17 shows times for Monocle's iCloud and Box plugins. In addition, a Box memory plugin not using Volatility was created to show Volatility overhead during execution.

Memory Modules	**Wrapper pre-setup**	**Module execution**	**Wrapper post-setup**
Box (w/o Volatility)	0,498s	25,940s	2,40E-05s
Box	0,737s	206,806s	3,20E-05s
iCloud	0,622s	49,048s	1,00E-05s
Disk Modules			
Box	1,227s	0,913s	3,20E-05s
iCloud	0,695s	3,893s	3,80E-05s

*** All results come from a Windows 7 x64 system emulated through a VMWare Virtual Machine memory and disk dumps. Times are the average execution time of three different independent executions of each listed plugin.**

Table 17: Performance Evaluation of plugins

Results show that executed plugins leverage overall running time, which implies Monocle's overhead is almost neglectable regarding execution times. This overhead is more noticeable for disk modules, 57,3% and 15,6% of the overall time for Box and iCloud respectively. Monocle overhead is smaller in memory modules, being 1,8% for Box module running Volatility and 0,6% for the one not using Volatility. Monocle overhead on iCloud memory module is 1,25%. Disk-targeted plugins overhead seems higher than memory ones. This fact is because disk plugins neither perform data carving nor have to search the entire digital container.

Significant differences exist between times of the module execution phase, as this stage

depends directly on executed the plugin. It is interesting to notice the overhead of using Volatility on a memory targeted plugin, e.g. Box plugin not using Volatility is 8 times quicker than the version using Volatility to reconstruct the memory space.

Chapter 7

Conclusions and future work

SECTION 7.1

Results of the project

This project created a tool (Monocle) which allows forensic analysis on a modular-based execution. This allows results of forensic analysis over well-known structures to be analyzed in a fast way and avoids the user to deal with the low level details of the implementation, as the analysis is perform behind the scenes. The framework also is extensible in order to allow the incorporation of new plugins created by the user. In addition, the framework provides a series of helper tools so as to ease the development and capabilities of the designed plugins, thus reducing the time and effort needed.

As part of the evaluation of Monocle, several plugins targeted client-side cloud scenarios were created. The objective was to prove the effectiveness of Monocle and to study such scenarios themselves. The results showed that client-side cloud scenarios analysis are a valid approach in order to analyze cloud storage scenarios based on SaaS cloud service model. Elements such as the identifiers of the user within the service, the elements synchronized with the cloud system and the different interactions of the user within the cloud system can be elicited from the client machine. This simplifies the forensic analysis as there is no need to perform such analysis on the cloud service provider machines, which is time-consuming and implies extra issues.

In addition to the present project, a scientific conference paper was created depicting the tool and the different conclusions of the present project [58]. This paper is currently under revision for its consideration to the VIII edition of CIBSI [59]. This conference is specialized in topics related to IT security, cryptography and digital forensics.

SECTION 7.2

Personal conclusions

One of the main problems encountered during the present project was the fact of develop it at the same time a bachelor in computer science was going on. This however resulted in more productive working session, as the time to work in the project was limited. This required prior planning for everything to be done.

The present work has been developed in the Computer Science Department of University Carlos III de Madrid. More precisely, the COSEC (Computer Security Lab) group laboratory. This allowed to know firsthand the structure and methodology followed by an actual research group. This research character has motivated the present project, adding an extra layer of meaning to the elements described in this report. It was not only about creating software or fulfilling the user needs, but to provide a useful tool for the researching community which can be further enhanced or employed in derivative works.

In addition, this project allowed to gain in-depth knowledge of IT security topics at the same time the knowledge on computer architecture and systems was applied. This resulted into passionate work, which is the key of any good work. It required a huge effort, but it was worth it.

SECTION 7.3

Future work

The tool proposed in this document is currently on its very first development stage. There are thus several improvements which can be applied to both improve its features or to add new ones.

One of the most meaningful improvements might be the inclusion of the TSK Framework within Monocle This tool would provide Monocle with two of the most powerful open source suites for forensic analysis, namely Volatility and TSK, this would allow to excel among open source tools as results coming from both platforms can be merged in order to provide better analysis. The disk carving capabilities of TSK and the automated evidence mounting of corrupted images are features of interest concerning the improvement of disk images plugins among others.

Related with TSK would be the inclusion of Autopsy within Monocle. Although this would require an adaptation of the original software, Autopsy might provide a common GUI to be used by all Monocle's components on a remote HTML base.

The processing of proprietary and compressed evidence sources (e.g. E01) is currently out of the scope of this tool regarding its first version. The inclusion of some kind of support for different formats is thus a future objective for Monocle. This will allow to multiply the range of possible evidence sources to analyze.

The inclusion of plugin-sharing capabilities of the framework by creating a centralized knowledge base where plugins can be uploaded and downloaded by different users is another matter to address in the future. This would reduce time and effort and facilitate collaboration among different analysts. In addition, such plugins can be re-used afterwards so as to create new, more detailed ones.

References

[1] Kamal Dahbur and Bassil Mohammad. The anti-forensics challenge. In *Proceedings of the 2011 International Conference on Intelligent Semantic Web-Services and Applications - ISWSA '11*, pages 1–7, New York, New York, USA, April 2011. ACM Press.

[2] Brian Carrier. The Sleuth kit. Accessed 05/04/2015 at http://www.sleuthkit.org/sleuthkit/.

[3] LM Ericsson. More than 50 Billion Connected Devices. *www.ericsson.com/res/docs/whitepapers/wp-50-billions.pdfâĂŐ*, (February), 2011.

[4] Audrey Guinchard. Cybercrime: The Transformation of Crime in the Information Age, 2008.

[5] AENOR. UNE 71506. Accessed 06/09/2015 at http://www.aenor.es/aenor/normas/normas/fichanorm

[6] Apple Inc. iCloud. Accessed 04/20/2015 at https://www.icloud.com.

[7] Box. Box Cloud. Accessed 20/04/2015 at https://app.box.com.

[8] Guidance Software. Encase Forensic v7. Accessed 03/17/2015, at https://www.guidancesoftware.com/products/Pages/encase-forensic/overview.aspx.

[9] EnCase EnScript programming. Accessed 06/09/2015 at https://www.guidancesoftware.com/training/Pages/courses/classroom/EnCaseÂő-EnScriptÂő-Programming.aspx.

[10] The Sleuth Kit pipeline system. Accessed 06/09/2015 at http://www.sleuthkit.org/sleuthkit/framework.php.

[11] Shams Zawoad and Ragib Hasan. Cloud Forensics: A Meta-Study of Challenges, Approaches, and Open Problems. *arXiv preprint arXiv:1302.6312*, pages 1–15, 2013.

[12] Yashpalsinh Jadeja and Kirit Modi. Cloud computing - Concepts, architecture and challenges. *2012 International Conference on Computing, Electronics and Electrical Technologies, ICCEET 2012*, pages 877–880, 2012.

[13] Brian D. Carrier. Digital forensics works. *IEEE Security and Privacy*, 7(2):26–29, 2009.

[14] Michael Kohn, Martin S Olivier, and Jan HP P Eloff. Framework for a Digital Forensic Investigation. In *ISSA*, pages 1–7, 2006.

[15] J.R. Lyle. NIST CFTT: Testing disk imaging tools. *International Journal of Digital Evidence*, 1(4):1–10, 2003.

[16] Chet Hosmer. Proving the integrity of digital evidence with time. *International Journal of Digital Evidence*, 1(1):1–7, 2002.

[17] Ray Yeager and Ray Yaeger. Criminal computer forensics management. In *Proceedings of the 3rd annual conference on Information security curriculum development InfoSecCD 06*, page 168, 2006.

[18] Karen Kent, Suzanne Chevalier, Tim Grance, and Hung Dang. Guide to integrating forensic techniques into incident response. *NIST Special Publication*, pages 800–886, 2006.

[19] CDESF Working Group. Standardizing digital evidence storage. *Communications of the ACM*, 49(2):67, 2006.

[20] Wei-Tek Tsai, Xin Sun, and Janaka Balasooriya. Service-Oriented Cloud Computing Architecture. *2010 Seventh International Conference on Information Technology: New Generations*, pages 684–689, 2010.

[21] Dominik Birk and Christoph Wegener. Technical issues of forensic investigations in cloud computing environments. In *2011 6th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering, SADFE 2011*, 2011.

[22] J.J. Shah and L.G. Malik. Cloud Forensics: Issues and Challenges. *2013 6th International Conference on Emerging Trends in Engineering and Technology*, pages 138–139, 2013.

[23] Sameera Almulla, Youssef Iraqi, and Andrew Jones. Cloud forensics: A research perspective. In *2013 9th International Conference on Innovations in Information Technology, IIT 2013*, pages 66–71, 2013.

[24] Darren Quick, Ben Martini, and Kim-Kwang Raymond Choo. *Cloud Storage Forensics*. 2014.

[25] Darren Quick and Kim Kwang Raymond Choo. Dropbox analysis: Data remnants on user machines. *Digital Investigation*, 10(1):3–18, 2013.

[26] Darren Quick and Kim Kwang Raymond Choo. Google drive: Forensic analysis of data remnants. *Journal of Network and Computer Applications*, 40:179–193, 2014.

[27] Darren Quick and Kim Kwang Raymond Choo. Digital droplets: Microsoft SkyDrive forensic data remnants. *Future Generation Computer Systems*, 29(6):1378–1394, 2013.

[28] Ben Martini and Kim Kwang Raymond Choo. Cloud storage forensics: OwnCloud as a case study. *Digital Investigation*, 10(4):287–299, 2013.

[29] Anandabrata Pal and Nasir Memon. The evolution of file carving. *IEEE Signal Processing Magazine*, 26(2):59–71, 2009.

[30] R. B. van Baar, W. Alink, and A. R. van Ballegooij. Forensic memory analysis: Files mapped in memory. *Digital Investigation*, 5(SUPPL.), 2008.

[31] The Volatility Framework project. Accessed 20/03/2015 at http://www.volatilityfoundation.org/#!24/c12wa.

[32] Brian Carrier and Eh Spafford. Getting physical with the digital investigation process. *International Journal of Digital Evidence*, 2(2):1–20, 2003.

[33] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys and Tutorials*, 16(2):961–987, 2014.

[34] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 2014.

[35] España, Ley 1/2000, de 7 de enero, de Enjuiciamiento Civil. BOE núm. 7, de 8 de enero de 2000, páginas 575 a 728.

[36] España, Sentencia 292/2000, de 30 de noviembre de 2000 del Tribunal Constitucional. R.IN. arts. 21.1 y 24.1 y 2 de la Ley Orgánica 15/1999, de 13 de diciembre, de Protecioón de Datos de Carácter Personal.

[37] España. Ley Orgánica 15/1999, de 13 de diciembre, de Protección de Datos de Carácter Personal. BOE núm. 298, de 14/12/1999.

[38] España. Real Decreto 1720/2007, de 21 de diciembre, por el que se aprueba el Reglamento de desarrollo de la Ley Orgánica 15/1999, de 13 de diciembre, de protección de datos de carácter personal.

[39] España. Ley 25/2007, de 18 de octubre, de conservación de datos relativos a las comunicaciones electrónicas y a las redes públicas de comunicaciones. BOE núm. 251, de 19/10/2007.

[40] España. Ley Orgánica 10/1995, de 23 de noviembre, del Código Penal. BOE núm. 281, de 24 de noviembre de 1995, páginas 33987 a 34058.

[41] Harlan Carvey. The Windows Registry as a forensic resource. *Digital Investigation*, 2:201–205, 2005.

[42] Bruce Eckel. *Thinking in Java 4th Edition*, volume 27. 2011.

[43] Alex Martelli, David Ascher, and Anna Ravenscroft. *Python Cookbook*, volume XXV. 2005.

[44] Brian W Kernighan and Dennis M Ritchie. *The C programming language*, volume 78. 1988.

[45] T J O'Connor. Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers, 1st edition. *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers, 1st edition*, 2012.

[46] Guido Van Rossum and Fred L Drake. The Python Library Reference. *October*, pages 1–1144, 2010.

[47] Guido Van Rossum and Fred L Drake. Using Python. *October*, page 49, 2009.

[48] Will Ballenthin. Python-Registry. Accessed 04/10/2015 at http://www.williballenthin.com/registry/, 2014.

[49] SourceForge. The timeline project. Accessed 03/10/2015 at http://thetimelineproj.sourceforge.net.

[50] https://docs.python.org/2/library/xml.etree.elementtree.html Accessed 04/20/2015.

[51] IEEE. IEEE Recommended Practice for Software Requirements Specifications. *Practice*, 1998:37, 1998.

[52] Esa. ESA Software Engineering Standards. *ESA Publications Division*, ESA PSS-05(2), 1994.

[53] By Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell*, volume 12. 2005.

[54] WJ Chun. *Core python programming*. 2006.

[55] Superponible. http://blog.superponible.com/2014/08/31/volatility-plugin-chrome-history/ Accessed 04/30/2015.

[56] Superponible. http://blog.superponible.com/2014/08/31/volatility-plugin-firefox-history/ Accessed 04/30/2015.

[57] KA Alghafli, Andrew Jones, and TA Martin. Forensic Analysis of the Windows 7 Registry. *Journal of Digital . . .*, page 17, 2010.

[58] Jorge Rodriguez-Canseco, J M De Fuentes, Lorena Gonzalez-Manzano, and A Ribagorda. MONOCLE - Extensible open - source forensic tool applied to cloud storage cases. 2015.

[59] CIBSI - Congreso Iberoamericano de Seguridad Informática. Accessed 2/06/2015 at http://cibsi.espe.edu.ec/index.html.

[60] Bases y tipos de cotizacion 2015. Accessed 20/05/2015 at http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm.

Appendix A

Project Management

SECTION A.1

Planification of the project

This section is intended to mark the project management proposed and followed during MONOCLE project. An initial estimation of the project phases, sequence and timing is described as it was planned prior to the project starting. An updated schedule is then presented with the different changes made during the project life cycle.

A.1.1. Initial Schedule

This section shows the initial planning calculated for the present project. The different phases of the project can be noticed and each one was assigned an estimated amount of time. Several tasks are dependent on the previous ones, as it will be showed further in this section.

The project was scheduled to start on September 8, 2014, and to finish on June 12, 2015. This project lasted for 228 days, and was carried at the same time as the 4th course of the Bachelor in Computer Science and Engineering of the Carlos III University. Days have been divided into 5 hours of work each. Holidays and week-ends are not considered in the following computations as no work was performed during such periods.

Only one person was working in the project. This fact is reflected in the fact that overlapping tasks cannot be performed in parallel and last longer than they should. This was initially considered in the initial planning.

Table 18 shows the expected starting and ending dates of the tasks planned for the project. Figure 25 plots such tasks in a Gantt chart in order to facilitate their understanding. Arrows within different tasks of the Gantt chart show the dependencies existing within the project management. These dependencies imply tasks not being able to be started when others still under development. It has to be noticed that minor corrections performed in the documentation after the end of the project are not considered in the present schedule due to their small workload.

Task name	Starting Date	Ending Date	Duration (days)
Planning	**9/8/14**	**9/9/14**	**2**
State of the Art study	**9/10/14**	**9/22/14**	**9**
Forensics and tools	9/10/14	9/15/14	4
Cloud forensics	9/16/14	9/22/14	5
User-side cloud forensics	9/17/14	9/22/14	4
Analysis	**9/23/14**	**10/9/14**	**13**
Requirements elicitation	9/23/14	9/25/14	3
Technological Study	9/24/14	9/29/14	4
Use Case definition	9/30/14	10/2/14	3
System Architecture	10/3/14	10/9/14	5
Design	**10/10/14**	**10/17/14**	**6**
Software Design	10/10/14	10/15/14	4
Sequence Diagrams	10/16/14	10/17/14	2
Implementation	10/20/14	4/1/15	**118**
Framework Implementation	10/20/14	1/30/15	75
Plugin implementation	3/5/15	4/1/15	20
Testing	**2/2/15**	**4/13/15**	**51**
Framework testing	2/2/15	2/11/15	8
Plugin testing	4/2/15	4/13/15	8
Evaluation	**2/12/15**	**3/4/15**	**15**
Cloud services study	2/12/15	3/4/15	15
Documentation	**4/14/15**	**6/11/15**	**43**
Project Documentation	4/14/15	6/11/15	43
Paper writing	4/30/15	6/3/15	25
Project Total	**9/8/14**	**6/11/15**	**257**

Table 18: Initial time estimation for the project

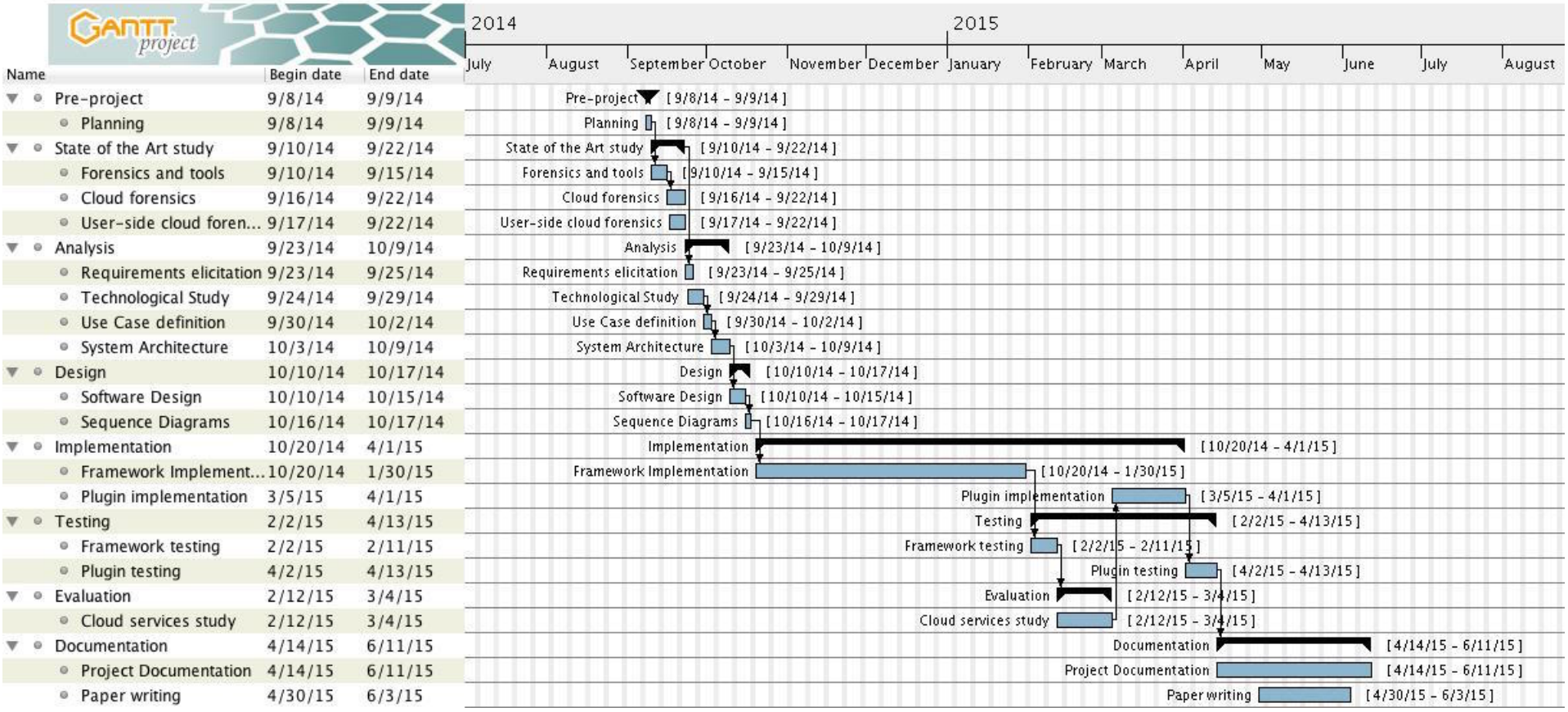

Figure 25: Expected schedule Gantt diagram

The proposed planning shows two different phases of implementation and testing. This is due to the nature of the present project. This project is a framework to be used with plugins, so different implementation and testing shall be performed in order to ensure both the software and plugins are working. As the latter cannot be done until the framework is working, it is necessary to divide the implementation phase in two different steps.

A.1.2. Definition of critical tasks

Critical tasks conform the *critical path* of the project development. A *critical path* is defined as the longest path which can be followed on a project scheduling having into account overlapping tasks. Thus, if there are two tasks executing at the same time, the critical task will be the one ending last.

The present project follows a waterfall methodology. This implies that each stage of the development cannot start until the previous ones have finished. Stages of the project comprehend the planning, state of the art study, analysis, design, implementation, testing, evaluation and documentation. Waterfall methodology does not actually benefits from the definition of a critical path. The present project however has two different implementation and testing phases for two different software pieces (namely the framework and the plugins used to test it), which makes the definition of a critical path important. Critical tasks are defined in Figure 25 as tasks which have an arrow pointing to the next task. By following such arrows it is possible to construct the critical path, which lasts (as expected), the same amount of time as the project (i.e. 257 days).

A.1.3. Real schedule

As the project was going on, differences between the original planning appeared. Different issues appearing for some tasks modified the overall time to complete them. Differences are however small with respect to the original schedule. Table 19 shows the updated schedule after changes made during the development cycle. Due to the differences in the time elapsed for each task, the ending date of the project changed for the better. The original project was expected to be finished on June 11, 2015, whether the actual project ended on June 8, 2015. This implies the project ended 3 days before expected. The updated Gantt diagram is presented on Figure 26. Besides, Table 20 shows the percentage difference of the different tasks regarding their initial planning and final development.

Task name	Starting Date	Ending Date	Duration (days)
Planning	**9/8/14**	**9/9/14**	**2**
State of the Art study	**9/10/14**	**9/24/14**	**11**
Forensics and tools	9/10/14	9/17/14	6
Cloud forensics	9/18/14	9/24/14	5
User-side cloud forensics	9/19/14	9/24/14	4
Analysis	**9/25/14**	**10/16/14**	**16**
Requirements elicitation	9/25/14	9/29/14	3
Technological Study	9/30/14	10/2/14	3
Use Case definition	10/3/14	10/7/14	3
System Architecture	10/8/14	10/16/14	7
Design	**10/17/14**	**10/24/14**	**6**
Software Design	10/17/14	10/22/14	4
Sequence Diagrams	10/23/14	10/24/14	2
Implementation	**10/27/14**	**4/2/15**	**114**
Framework Implementation	10/27/14	2/2/15	71
Plugin implementation	3/6/15	4/2/15	20
Testing	**2/3/15**	**4/14/15**	**51**
Framework testing	2/3/15 2/12/15	8	
Plugin testing	4/3/15	4/14/15	8
Evaluation	**2/13/15**	**3/5/15**	**15**
Cloud services study	2/13/15	3/5/15	15
Documentation	**4/15/15**	**6/8/15**	**39**
Project Documentation	4/15/15	6/8/15	39
Paper writing	4/24/15	5/27/15	24
Project Total	**9/8/14**	**6/8/15**	**254**

Table 19: Real time elapsed in the project

Task name	Expected	Real	Difference	Percentage change
Planning	**2**	**2**	0	0.00%
State of the Art study	**9**	**11**	+2	+22.22%
Forensics and tools	4	6	+2	+50.00%
Cloud forensics	5	5	0	0.00%
User-side cloud forensics	4	4	0	0.00%
Analysis	**13**	**16**	+3	+23.08%
Requirements elicitation	3	3	0	0.00%
Technological Study	4	3	-1	-25.00%
Use Case definition	3	3	0	0.00%
System Architecture	5	7	+2	40.00%
Design	**6**	**6**	0	0.00%
Software Design	4	4	0	0.00%
Sequence Diagrams	2	2	0	0.00%
Implementation	**118**	**114**	-4	-3.39%
Framework Implementation	75	71	-4	-5.33%
Plugin implementation	20	20	0	0.00%
Testing	**51**	**51**	0	0.00%
Framework testing	8	8	0	0.00%
Plugin testing	8	8	0	0.00%
Evaluation	**15**	**15**	0	0.00%
Cloud services study	15	15	0	0.00%
Documentation	**43**	**39**	-4	-9.30%
Project Documentation	43	39	-4	-9.30%
Paper writing	25	24	-1	-4.00%
Project Total	**257**	**254**	-3	-1.17%

Table 20: Difference between expected and final project schedules

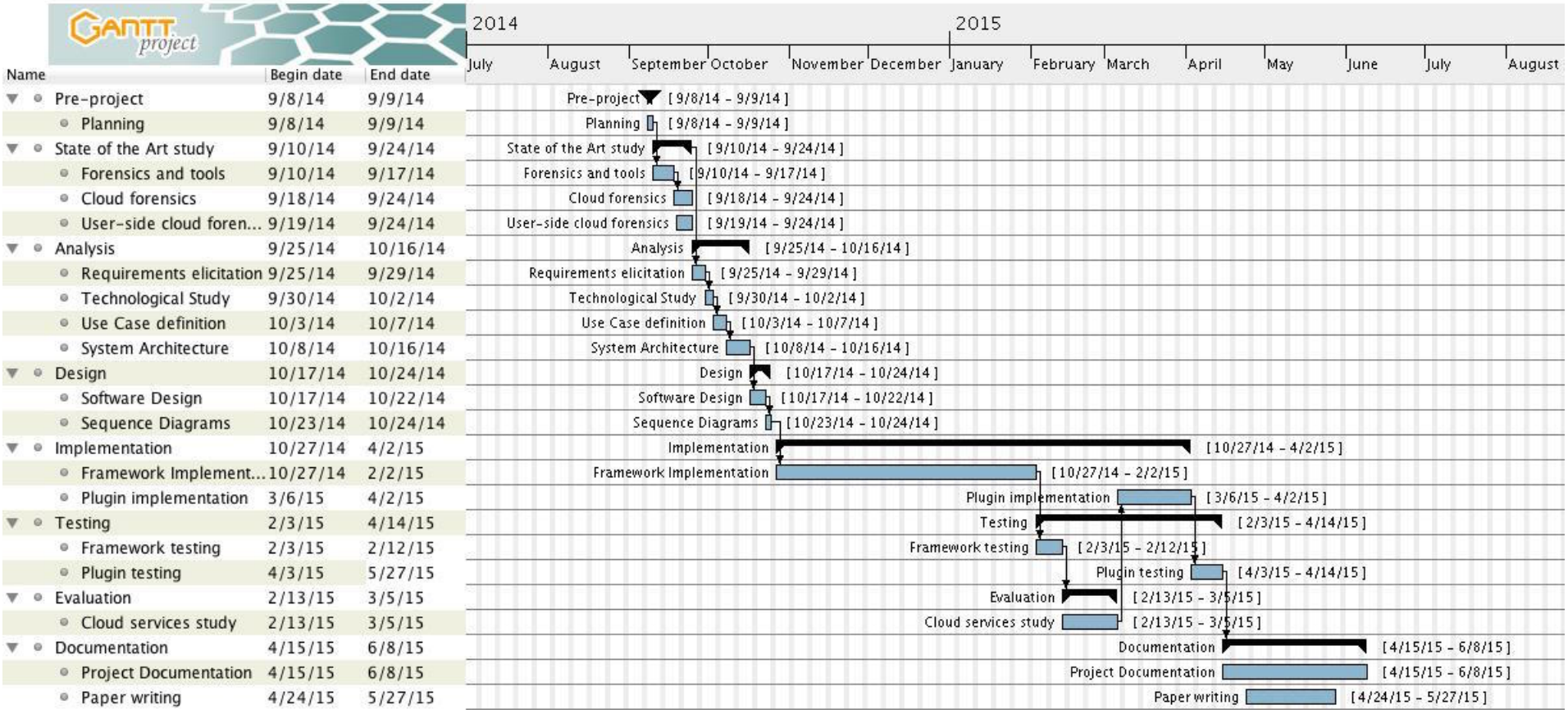

Figure 26: Final Gantt diagram

SECTION A.2

Economical Analysis

This section is intended to provide an analysis of the project cost from an economic perspective. This analysis will include an initial estimated budget, the client budget and the real cost of the project.

A.2.1. Methodology

All these budgets are calculated including direct and indirect costs inherent to the project. The project was expected to last 257 days, which will be considered in costs directly related with the time.

Direct costs are related with elements directly related with the project development. These concepts derived from objects and personnel, as well as other quantitative elements which are directly assigned to the project itself. Prices detailed here do not take into account any discount. Displacement costs are accounted in terms of distance traveled

Indirect costs are composed of costs deriving of the usage of the assets counted as direct costs. This include maintenance costs, power, and other assets which are required in order for the elements used in the development of the project to work. Spanish 18% VAT is accounted at the end of the costs calculation project in order to provide a final budget.

Recall that a comma separator (,) is used in order to express decimals in the costs below, and a dot separator (.) is used to separate numbers into groups of three, e.g. thousands and millions.

A.2.2. Initial Budget

This section details the initial budget of the project. Such budget is composed of personnel costs, equipment, software, consumable items, traveling and food costs. **??** comprehend all these assets. For the sake of clarity, each asset is defined and pre-calculated as stated below.

- **Personnel costs**. This cost was obtained from the workload of a single engineer during the project. 5 hours of work from Monday to Friday. This cost was re-calculated in order to add the 28,3% of taxes regarding Spanish Seguridad Social [60]. Costs regarding the present project supervisors are **not** considered, as their management role is variably split into several projects. Personnel costs are depicted in Table 21.

Concept	**Cost/u(€)**	Units (h)	**Raw cost**	**Indirect cost**	**Total**
Engineer (1)	27€	1.285h	34.69€	9.818,68€	44.513,68€
				Total	**44.513,68€**

Table 21: Personnel costs

- **Equipment costs**. These costs are derived from the equipment used during the project. Such equipment was composed of a laptop, an additional monitor screen and a printer. Equipment costs are depicted in Table 22.

Concept	**Cost (€)**	**Dedication**	**Deprecation period (Months)**	**Applicable cost**
MacBook Pro 15 13-inch 2,9 GHz Intel Core i7 8 GB 1600 MHz DDR3	2.193,00 €	8,56 Months	36 months	521,44€
Asus VS197DE (Monitor)	105,78€	8,56 Months	36 months	19,01€
HP Officejet 6500A (Printer)	79.95€	8,56 Months	36 months	25,15€
			Total	**565,60€**

Table 22: Equipment costs

- **Software costs**. Derived from licensing of software used during the project. The operating system and the development environment, as well as different tools used for documenting compose this set of assets. Software costs are depicted in Table 23.

Concept	Cost (€)	Dedication	Deprecation period (Months)	Applicable cost
Apple OSX 10.10 Yosemite	0,00€	8,56 Months	36 months	0,00€
JetBrains PyCharm	199,00€	8,56 Months	36 months	47,31€
TexMaker v4.3	0,00€	8,56 Months	36 months	0,00€
OmniGraffle v6.2.3	0,00€	8,56 Months	36 months	0,00€
			Total	**47,31€**

Table 23: Software costs

- **Consumables costs**. This covers consumable items used during the development, such as office material and printer cartridges. Office material comprehends notebooks, pens, pencils, stapler and paper sheets. Consumables costs are depicted in Table 24.

Concept	Cost (€)	Units	Applicable cost
Office material	33,50€	1	33,50€
Printer Cartridge	89,99€	1	89,99€
		Total	**123,49€**

Table 24: Consumables costs

- **Traveling and Food costs**. Expenses incurred in the displacements to the workplace, namely Leganés) (2,9 km) and for food (one per day) at work. Travel and food costs are depicted in Table 25.

Concept	Cost (€)	Units	Applicable cost
Travel	0,12€/Km	1	189,30€
Food	6,00€/day	257 days	1.542,00€
		Total	**1.731,30€**

Table 25: Travel and Food costs

A.2.3. Direct Costs

This section describes direct costs. Direct costs are calculated by summing up all the partial costs calculated for assets in subsubsection A.2.2. Table 26 shows the direct costs associated with the project. VAT is not included as it will be added afterwards in subsubsection A.2.5.

Concept	**Applicable cost**
Personnel costs	44.513,68€
Equipment costs	565,60€
Software costs	47,31€
Consumable costs	123,49€
Traveling and Food costs	1.731,30€
Total	**46.981,38€**

Table 26: Direct costs calculation

A.2.4. Indirect Costs

Indirect costs include elements which are not accounted directly. Some examples contemplated in the indirect costs are security costs, administration costs, electricity and other utilities (although some of then might depend on the industry the project is related to).

Indirect costs are to be calculated as previously stated on subsubsection A.2.1. As a fix expense is difficult to state for indirect costs, they are usually computed as a percentage of the direct costs. In this case, indirect costs represent a 22% of the direct costs, thus yielding 10.335,90€. This is to be included in the following section when estimating the overall costs.

A.2.5. Cost estimation

Table 27 details costs both direct and indirect. It also includes Spanish VAT of 18% applicable to the project. These costs will be used afterwards in order to state a client budget and a business proposal.

Concept	**Applicable cost**
Direct costs	46.981,38€
Indirect costs	10.335,90€
Total costs no VAT	57.317,28€
VAT (18%)	10.317,11€
Total costs with VAT	67.634,39€

Table 27: Estimated Costs

A.2.6. Client Proposal

This section accounts for the calculations made in order to create the Client proposal. This proposal has into account different extra parameters, such as the benefit to be obtained and a calculation of the risks of the proposal. No additional income is to be received as Monocle is to be distributed under an open source license.

Risks usually associated to software tools in similar projects represented 10% of the overall costs (no VAT). The present project maintains such a percentage in order to calculate the risk value associated

Regarding benefits, they will represent a 14% of the total final price (i.e. after applying the risk percentage and with no VAT). This value has been chosen due to the fact that similar projects do ask for a 15% of benefit. By reducing this in 1% it is possible to better-position our proposal among similar ones.

Table 28 depicts the different expenses and added values, and shows the final project proposal to the client.

Concept	**Applicable cost**
Direct costs	46.981,38€
Indirect costs	10.335,90€
Total costs no Risk	57.317,28€
Risk (10%)	5.731,72€
Total costs no Benefits	63.049,00€
Benefits (14%)	8.826,86€
Proposal no VAT	71.875,86€
VAT (18%)	12.937,65€
Total costs with VAT	**84813,51€**

Table 28: Client proposal

Appendix B

Monocle User Manual

SECTION B.1

About the Manual

The present annex is intended to provide an detailed description of the installation process of the software presented in this report, as well as a guided walk-through the contents and capabilities the software is capable of.

B.1.1. Manual Structure

The first section of this manual will be intended to the installation of the required elements in order to run Monocle on a computer. Such elements will be generalized as much as possible in order to make this manual applicable to different operating systems and architectures.

The second section of this manual will focus on the usage of plugins and it will enumerate where the elements are to be created once a plugin is run on the system. It will also specify the main features of the present software which can be triggered by the user during the execution of a given plugin.

The third and final section of this manual will be focused in the creation and inclusion of new plugins within the framework. This section will contain more technical nomenclature as it is intended for developers who want to extend the existing functionality of the software and to contribute with the community in order to enhance the present framework.

B.1.2. Legal Disclaimer

Monocle is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

Monocle is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with Monocle. If not, see $< http://www.gnu.org/licenses/ >$.

SECTION B.2

Installation of the software

Monocle has been developed in the Python language. This language has the advantage of offering multi-platform support, so the process of installation do not differ much from one computer to another. This section will highlight the software requirements prior to installation, as well as a recommended way to install the present software.

B.2.1. Software dependencies

This section states the elements which must be installed in the computer prior the installation of Monocle in order for it to work properly. A link to such resources will be given if possible. Notice that all of the resources listed here are released under open-source licenses, and you shall not pay any extra fee for them as stated on their respective licenses.

The dependencies of Monocle prior to its installation are namely:

- **Python v2.7** (http://www.python.org).
- **Volatility Framework**. Version (2.4 or above). https://github.com /volatilityfoundation/volatility
- **Python Registry**, from Will Ballenthin. http://www.williballenthin.com /registry/index.html
- **Timeline Module** http://thetimelineproj.sourceforge.net/about.html.

Once such dependencies are installed within the system, it is possible to commence the set-up of Monocle.

B.2.2. Monocle setup

Setup of Monocle is a quite straightforward process once the dependencies are installed. Shall you place your copy of monocle within a directory of your choosing, Monocle requires of no installation. By invoking Python over MonocleGUI.py file within the Monocle folder is enough for Monocle to load its dependencies and set up its environment.

SECTION B.3

Monocle Usage

Monocle can be started by invoking Python over MonocleGUI.py file. This will load the initial environment of Monocle, and will lead to Monocle's main interface (Figure 27). It is possible to execute the framework on its command-line version by means of the execution of Python over the Monocle.py file with the required arguments. Such arguments are further explained in subsubsection B.3.2.

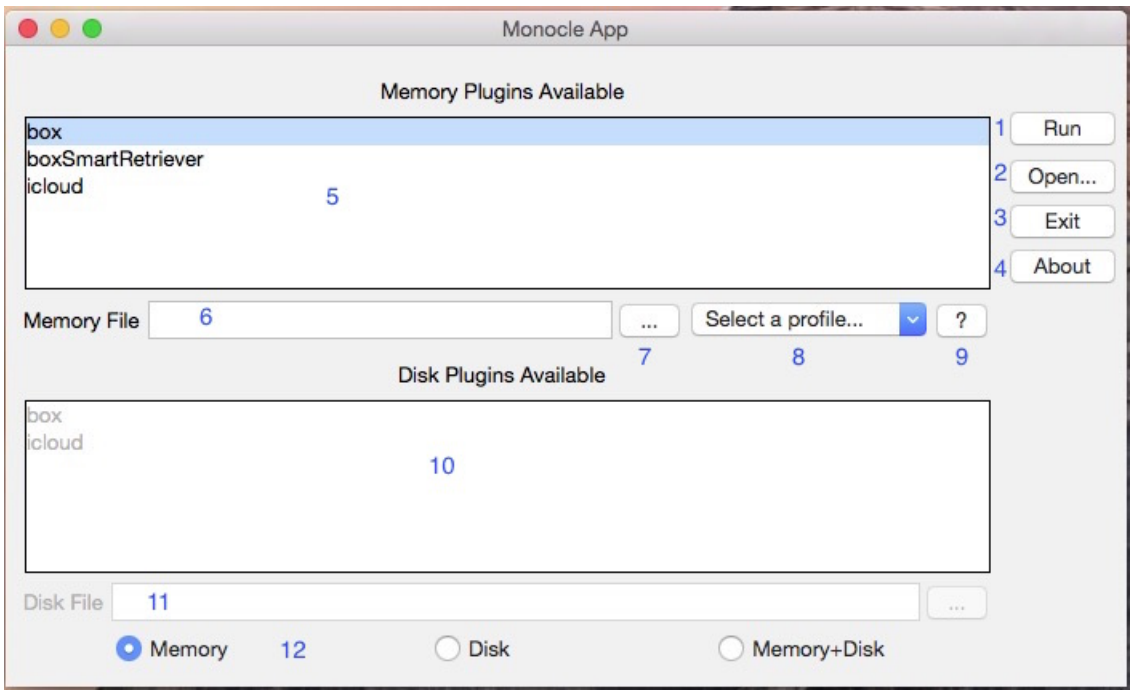

Figure 27: Monocle's Main interface

From Monocle's main interface, it is possible to operate within Monocle and all its components by selecting user plugins, evidence sources and different additional options which are available.

B.3.1. Main Interface elements

Numbers on Figure 27 stand for the different features to interact with, which are defined as follows:

1. **Run Button**. This button will invoke the framework over the selected parameters in 12, 5, 6/11 and 8 (if applicable). Monocle will star computation until the analysis performed by the user plugin is finished. Once finished, it will show a summary window with the different evidences found.

2. **Open Button**. This button is used in order to load previously realized analysis so as not to repeat them. This will load the generated summary provided at the end of the analysis as if it was just executed.

3. **Exit Button**. This button will terminate the program.

4. **About Button**. This button provides further information regarding Monocle. (e.g. Version number, authors...)

5. **Memory Plugin Selection List**. This box provides a list of all the loaded plugins within the framework targeting Memory. The selected plugin will be the one executed when the Run (1) button is pressed.

6. **Memory analysis source box**. This text field receives the path within the system of the digital container where the target element to analyze is located. RAW memory dumps are the only accepted.

7. **Path browsing**. This button provides a browsing window in order to choose the file to analyze so as to ease its location. The path of the selected item will be placed in 6.

8. **Volatility Profile selection**. This dropbox contains a list of all the supported Volatility plugins. If the plugin being executed makes use of Volatility, it might be necessary to provide a profile for it to run. If no profile is selected and the plugin does need one, Monocle will try to automatically detect the proper profile to use.

9. **Help on Volatility Profiles**. This button pops up a dialog box containing information regarding the available Volatility plugins as well as their specifications. This information is retrieved from the Volatility Framework documentation.

10. **Disk Plugin Selection List**. Similarly to 5, this list provides the set of currently loaded disk plugins which are available to run and are targeting Disk dumps. The selected plugin will be executed when the Run (1) button is pressed.

11. **Disk analysis source box**. Similarly to 6, this box receives the path within the system of the digital container where the target element to analyze is located. RAW disk images are the only accepted.

12. **Mode execution selection**. This radio button selector allows the user to choose either to run a Disk module, a Memory module or both in the same analysis. Depending on the chosen option, the different elements in the interface regarding each one of the targets will become available or unavailable.

B.3.2. Monocle command-line execution parameters

Apart from the possibility of executing Monocle from a graphical user interface, it is possible to run the tool directly from command line by providing the required parameters in the calling line. The general syntax of the command line execution is as follows:

python Monocle.py [command] [-m <memory dump> | -f <disk dump>] -p <Persistence Path>

Where *command* is either *disk* or *memory*, -f and -m are the path to the disk or memory dump, respectively, and -p is the path used as destination for the analysis results and evidences.

No analysis result window is created when the analysis runs on command-line mode. The results can however be accessed by means of the *Open* button on Monocle's main GUI.

B.3.3. The Execution Process

The execution process takes place after the user clicks the Run button in the main window of the framework. At this point, the different options selected in such window will be parsed to Monocle's core. The selected plugin will start and, in case the mode selected were disk, the user will be asked for the partition to mount, as depicted in Figure 28.

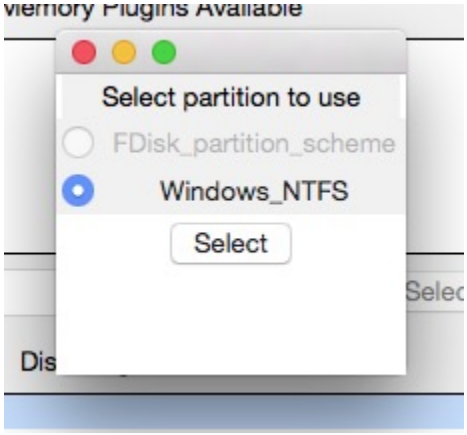

Figure 28: Partition Selection Window

Once the environment is ready, Monocle will start executing the different actions stated by the user plugin. The time spent on this phase depends on each plugin, so it is difficult to precise. During this phase, a *Processing* message will remain in the screen, as Figure 29 depicts. Please notice that the exiting Monocle in this phase might result into unexpected results, as the management of the evidences has to be performed during the whole program lifecycle so as to provide integrity.

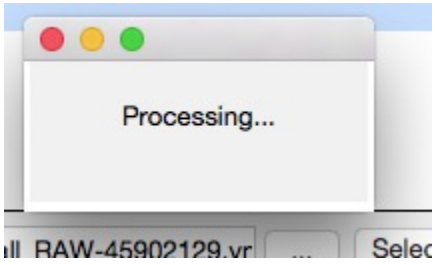

Figure 29: Plugin Execution Phase

Once the process is finished, a window with a result summary will be created (Figure 30).

This window provides further information regarding the evidences found, as well as the data the analyst might have created. This data is organized according to the parameters stated in the plugin code, so as for the plugin coder to state the relations he might consider proper depending on the nature of the retrieved evidences.

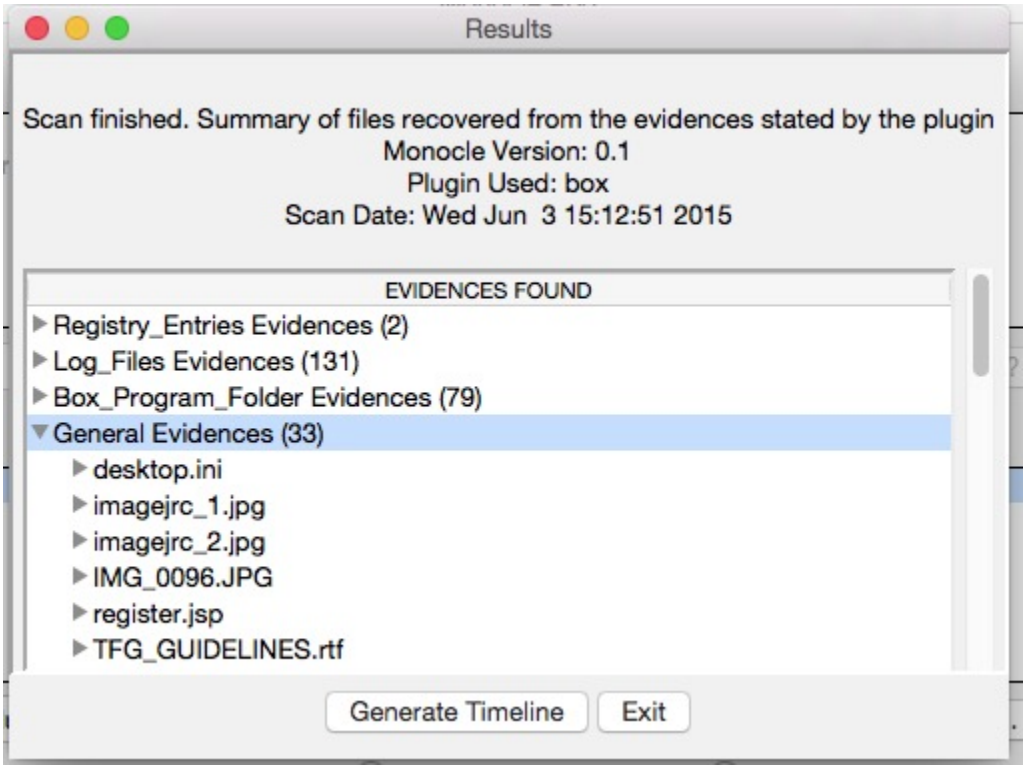

Figure 30: Monocle Summary Window

Evidences are organized in this window in a hierarchical way, and each entry contains all the information of interest retrieved for each single evidence. These evidences can be found within the output folder (i.e. as explained below). Automatic hashing and storing of evidences is provided so as to provide integrity.

This scan summary is actually being loaded from an XML file created in the destination path provided when the analysis started. This implies that the summary window can be opened in a future by means of the *Open* function without the need of re-executing the whole analysis.

Within the results window, there is the possibility to create a timeline with the evidences. Such timeline will organize the found evidences in a graphical way depending on the analyst statements. If no directives were provided during the analysis plugin creation, the acces time residing in the file metadata will be used as organizing directive. Figure 31 shows the timeline associated with the previous analysis in Figure 30. Categories being the same as

the ones appearing in the summary window can be displayed or hidden by means of the checkboxes in the upper left corner of the window.

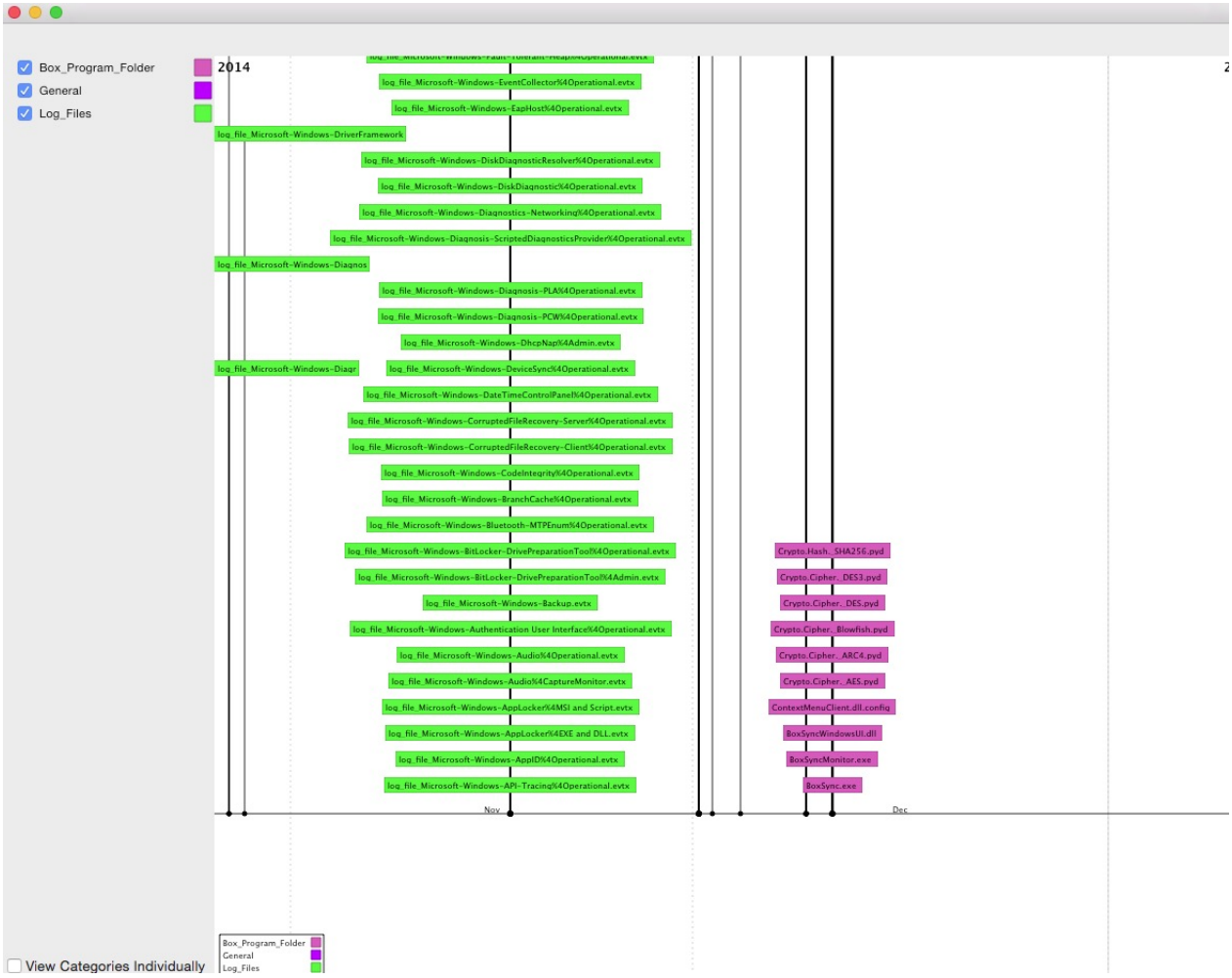

Figure 31: Monocle Timeline Window

The retrieved evidences during the analysis and their respective integrity and metadata will be stored within the local file system in the output path designated (or the local Monocle folder if no path was provided, namely *analysisResults*. This folder will contain in addition the generated files both created to store the analysis summary and the timeline (*analysisResults.xml and timeline.timeline, respectively*. Figure 32 depicts an example of the general structure of this folder.

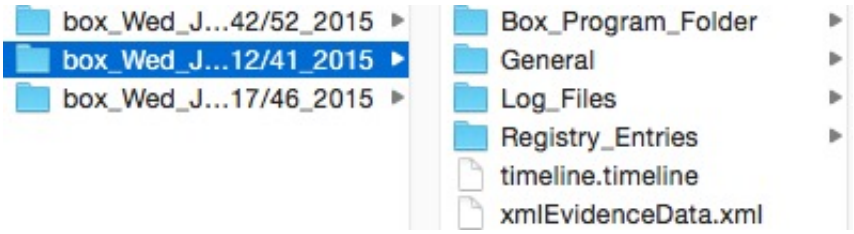

Figure 32: Output folder example

B.3.4. Volatility Profile Selection

The Volatility Framework is able to reconstruct the memory space of a system by means of predefined structures which are defined in the so called profiles. Each profile is associated with a specific operating system, and contains the inherent memory structures and organizations which are used by such system in order to read RAM memory.

The selection of the appropriate profile is thus of the outermost importance, as an incorrect choosing will result in a wrongly reconstructed memory space address. In case the user does not know the operating system the image comes from, it is possible to retrieve such data by means of Volatility via *kdbgscan*.

Kdbgscan is a Volatility plugin which identifies the operating system based on kdbg debugger flags present in memory structures. This analysis provide a relatively accurate inference of which operating system the memory belongs to.

Kdbgscan is automatically executed by Monocle if no memory profile is selected so as to provide a faithful reconstruction of the memory space.

SECTION B.4

Monocle Plugin design

User plugins are used by Monocle in order to provide functionality over the evidence sources. Plugins provide capabilities the analyst can work with, such as the usage of Monocle built-in tools or the capabilities provided by means of the Python programming language.

This section is intended to provide further knowledge as well as a guide in how to create new plugins which work with Monocle.

B.4.1. Monocle Plugin format

In order for Monocle to recognize a user-made plugin, it is necessary to follow a set of steps and a given format in order for the program to properly detect the elements in the plugin.

Monocle plugins are Python files. This implies that the extension of the file must be of type *.py* for the framework to recognize the plugin.

Such plugins are dynamically loaded within the framework. This implies that they must be placed at a special directory and must follow a determined schema in order for them to work normally and benefit from the built-in tools Monocle provides.

```
from modules.HDModule import hdModule

class retriever(hdModule):

    def retrieveData(self):
        """Entry function for Monocle app"""
```

(a) Disk-targeted modules

```
from modules.MemoryModule import memoryModule

class retriever(memoryModule):

    def retrieveData(self):
        """Entry function for Monocle app"""
```

(b) Memory-targeted modules

Figure 33: Code inclusion structure in Monocle Plugins

The basic schema for a Monocle Plugin only requires some code inclusion within the plugin code. This code inclusion is used basically to create a *retriever* object. This object is used by Monocle as the entry point for the program. More specifically, a function namely *retrieveData* within the object is to be used as Main entry to the program. Plugins schema varies depending on the target of the analysis as the *retriever* object is different whether the

target is memory or disk. Figure 33a and Figure 33b show the code import necessary for disk and memory, respectively.

Once this code is included, the analyst can start making use of the tools included within monocle and its own means in order to execute an automated analysis of the evidence sources. Once the plugin is finished, it must be included within the Monocle framework plugin directory. Such directory contains two different folders where memory-targeted and disk-targeted plugins are located. Monocle will search in these folders for new plugins each time the framework is started. In order to include a new plugin, it is enough to copy the plugin to the appropriate folder (Figure 34), and it will detected automatically.

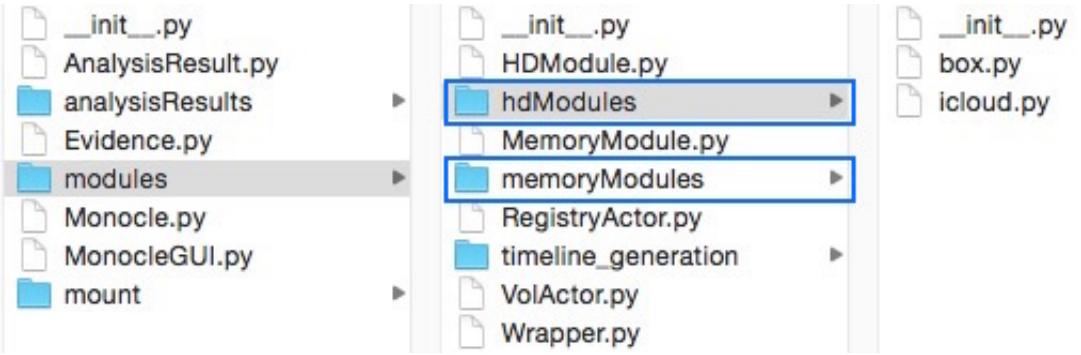

Figure 34: Plugin directory example.

If the plugin structure is correct and the plugin is placed in the proper directory, it will appear in the main Framework GUI (Figure 35). Notice that in case the plugin is detected but the structure is not the proper one, an error might occur at execution time. This error will occur however prior to the execution of the plugin (i.e. in Monocle's internal setup phase), so there is no real risk for the digital evidences integrity.

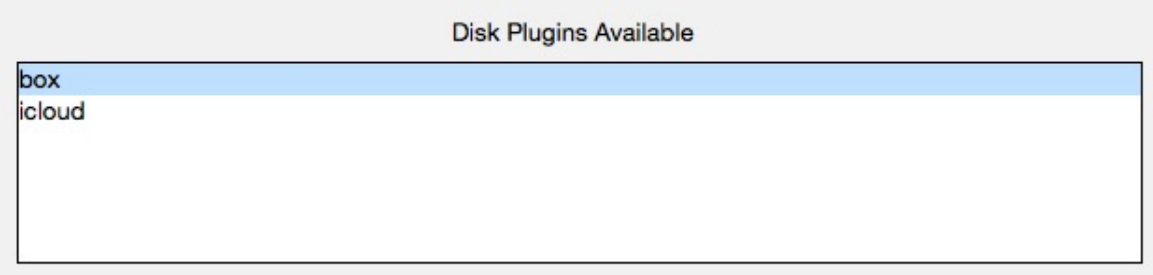

Figure 35: Properly detected files of Figure 34.

SECTION B.5

Interacting with the framework

The fact that the plugin makes use of the Python language allows the user both to make use of Monocle's own features and to create its own algorithms by means of the Python API. Whether the design of analysis algorithms by means of Python is out of the scope of the present document, this section presents the interaction of the plugin with the evidence sources and the different built-in tools.

B.5.1. Interaction within the digital containers

The analysis will be performed over the digital container specified by the user in Monocle's main GUI (i.e. or by means of the command-line argument). This digital container will be already managed by the Framework when the user script starts to execute, and the path within the local file system can be retrieved by means of *self.memfile_path* attribute of the *retriever* object (i.e. in case of a memory-targeted plugin). In case the plugin target were disk, this will be already mounted within the local file system in read-only mode, being it accessible by means of the *self.mountedRoot* attribute of the *retriever* object.

By doing so, it is possible to interact directly within the evidence source digital container. Monocle comes with a set of tools a user plugins can make use of in order to ease the analysis process and to speed up the coding phase of the plugin. Although this version of Monocle is an early version and only includes few tools, there are one available for each plugin type (disk and memory). Such elements are VolActor (i.e. interfacing the Volatility Framework) and the RegistryActor (i.e. interfacing the Python Registry plugin).

B.5.2. VolActor usage

The VolActor tool allows the analyst to use all the power of the Volatility framework inside of its own plugins. By interfacing those features, Monocle is able to seamlessly provide the different functions with a minimum effort in the analyst coding phase. Monocle will setup, deploy, detect and load the different options volatility requires in order to perform its different functions, and will return a data list with the results found (i.e. this allows the user

plugin to work directly with such set instead of parsing Volatility's text output).

In order for the user plugin to use the VolActor object, the plugin type must be targeted to memory (i.e. the retriever must belong to the MemoryModule package). If so, the retriever object will already contain a non-initialized VolActor object. This object can be initialized with the *start_volatility_framework* function existing in the retriever object. This function will make use of the different parameters coming from the user in the GUI in order to start the Volatility environment properly (e.g. the Volatility profile and the evidence target). If no profile was selected, Monocle will try to guess the appropriate one.

Once the VolActor is initialized, its different functions will be available. These functions are defined in Table 29, as well as their respective attributes and return values.

Function	**Description**
loadPlugin (modulePath, command, className)	Loads the volatility plugin located in package 'module' 'command'. This is mapped as 'from 'module' import 'command'. This function will add the loaded plugin to the dictionary of plugins in volActor object. calculate() and related volatility functions can be called from there. **Arguments** • modulePath: The Volatility module path in dotted notation. Ex: volatility.plugins.filescan • pluginName: The pluginName to be loaded from the module selected. Ex: PSScan • className: The class containing the calculate() function within the plugin.
Returns • None executePlugin(plugin)	Executes the calculate() function residing in the selected 'plugin' of the volatility framework. **Arguments** • plugin: The plugin to be executed **Returns** • Iterator resulting from the calculate() function.

Function	Description
executeRender(plugin, data, outfd= sys.stdout)	This function will execute the render_text function of the selected volatility plugin. This is useful for some plugins (i.e. Dump plugins) where the actual dump over the data is performed in this step. **Arguments** • plugin: The name of the plugin to be executed as stated in the load_plugin function. • data: The data to parse to the render_text function in order to operate over it. Usually, the result of the calculate function of the same plugin (ex: a= executePlugin(pslist); executeRender(pslist, a). • outfd: Optional parameter in order to specify the output file descriptor to be used. By default it will be stdin. **Returns** • None
setArguments(**kwargs)	Adds a variable number of arguments to the current configuration context of volatility. This functions accepts an arbitrary numbers of key-value pairs (ex. setArgument(PID=4, DUMP_DIR="/usr/local"). Notice that key must be the one specified in the target volatility plugin to be used **Arguments** • kwargs: List of key-value pairs of arguments to pass to Volatility. **Returns** • None
getArgument(option)	Returns the value of the current configuration argument specified by option in the volatility framework configuration **Arguments** • option: the key of the given option (Ex: PID) **Returns** • Value associated to the option passed as argument.

Table 29: VolActor interface functions

The user plugin can call any of these functions in order to prepare and execute Volatility analysis calls over the evidence. Figure 36 shows an example of a call to the IEHistory plugin, identified by *iehistory* command and located under the volatility.plugins package of Volatility.

```
self.start_volatility_framework()
self.volActor.loadPlugin("volatility.plugins", 'iehistory', 'IEHistory')
resultList = self.volActor.executePlugin(plugin[1])
```

Figure 36: Sample execution of Volatility Plugin.

NOTE: Please notice that Volatility takes its time in order to perform its computations. Expect the execution time for your plugin to be increased considerably if using Volatility.

B.5.3. RegistryActor usage

The RegistryActor is a tool built-in inside Monocle's core which allows the user plugins to automatically parse Windows registry key hives. Such hives are found in a proprietary format from Microsoft, and cannot be interpreted directly without parsing. RegistryActor component uses Python Registry module from Will Ballenthin in order to provide its features.

The functioning of the RegistryActor is somehow similar to the functioning of the VolActor component, with the main difference that it does not need to be initialized in order to work (i.e. as there is no need of prior configuration for the tool to work).

The interface for the tool consists basically of two functions which are sufficient in order to explore the registry. Such functions are used to retrieve registry keys and registry subkeys, respectively. Each function has two different arguments, being the first the path to the registry hive location within the digital container (i.e. the local path, as Monocle provides automatic translation), and the second the registry key/subkey name to look for. Notice that whether the hive path address specification is Unix-like notation, registry path within the hive is Windows-like one.

Table 30 shows the specifications for such two functions, and Figure 37 shows an example call to the API. More precisely, Figure 37 depicts the SAM/Domains/Account/Users/Names from the SAM registry hive, which contains the different user names the target system has registered.

```
self.userList = self.registryActor.extractSubKeys("/Windows/System32/config/SAM", "SAM\\Domains\\Account\\Users\\Names")
```

Figure 37: Sample execution of Registry Plugin.

Function	Description
extractKeys (pathToHive, pathToKey)	Extracts the registry keys specified by pathToKey. Returns a list of tuples with structure [valueName, valueValue] or None if the key or the registry hive were not found. An empty tuple implies that the key has not values associated. **Arguments** • pathToHive: The path to the registry hive to open • pathToKey: The key to search for within the hive **Returns** • A list with the key values. None on error (e.g. the key does not exists).
extractSubKeys (pathToHive, pathToKey)	Extracts the registry subkeys path belonging to specified by pathToKey. Returns a list of tuples with structure [valueName, valueValue] or None if the key or the registry hive were not found. An empty tuple implies that the key has not values associated. **Arguments** • pathToHive: The path to the registry hive to open • pathToKey: The key to search for within the hive **Returns** • A list with the key values. None on error.

Table 30: RegistryActor interface functions

B.5.4. Evidence Manager usage

The evidence manager is one of the most useful tools built into Monocle. It provides automation over some of the most important tasks regarding the forensic process on digital evidence management, namely:

- **Digital-feasible documentation of the evidence.** This includes the data recovered from the evidence itself, as its location within the digital container, the metadata of the evidence (if any), and the destination where this evidence will be stored after analysis.
- **Recovery of evidence data.** Which is referred to the fact of gather the evidence file or evidence elements, and to store the interpretations, which can be optionally given by the investigator depending on the context. This also includes organizing the data for further ease of access.
- **Securing of evidence data.** Implying the insurance of consistency by means of hash functions. This guarantees the consistency of the files and the evidence tree generated, as well as the generated reports and elements of the analysis.

The evidence manager is the way the user plugins tell Monocle where the evidences are located. In this way, it is possible for the framework to collect them and securely store them within the local analyst's file system.

In order for the user to call the Evidence Manager, it is enough with creating a new Evidence object. This automatically triggers the Evidence Manager. The Evidence Manager will take care of the different attributes specified for the Evidence in order to gather the necessary elements and store them securely within the analyst hard drive. Additional comments can be specified to each evidence, as well as a parameter to use as value for timeline ordering.Table 31 shows an example call to this evidenceManager.

Element	Description
Evidence (element, source, destination, category= ”General”, additional_parameters= None)	Class representing an evidence object recovered during the analysis. **Arguments** • element: File or StringIO containing the evidence. StringIO implemented in order to be able to record evidences not related to files, such as registry keys. • source: Specification on where the evidence was found. • destination: The file used for store the evidence element in the Evidence folder. The path will be calculated automatically having as root a folder with the category name. • category: Category in which to classify the evidence. It must be a string containing a valid path name (i.e. without /). • additional_parameters: Dictionary containing optional values to be added to the evidence. Default is . **Returns** • True on success. *AttributeError* or *TypeError* on failure (input invalid).

Table 31: Evidence definition interface

SECTION B.6

Creating a Plugin

The present section presents a walktrough of the process followed in order to develop a real plugin for monocle. It provides a detailed step by step list in order to show the different elements needed in a real case.

B.6.1. Defining the plugin. Functionality and objectives

The first thing to consider when coding a new plugin is the target and objective of such plugin. It is necessary to clearly state the elements to yield by the plugin and their meaning. It is also important to bear in mind the fact that the existence of the elements to yield (e.g. files, photos, registry keys) might not be meaningful by their own existence but depending on the context they exist on.

During this walkthrough a plugin to extract all urls existing within running web browser processes is going to be developed. Thus, the main objective of the plugin is to return all existing URLs. In order to do so, it is necessary to reconstruct the browsers memory space so as to be able to carve URLs from it.

Volatility provides a way to reconstruct processes memory address by specifying the PID of the process. It also provides a way to list all processes running on a memory dump and to obtain all the URLs associated to them.

B.6.2. Coding the plugin. Initial setup

Once the process to follow is known, it is time to write some actual Python code to see it in action. As mentioned previously in this manual, the first thing to do is to create a Python file (extension .py) in the corresponding folder within Monocle installation. As the plugin we are coding is targeting memory, this folder is the *memoryModules* folder. This folder can be found inside Monocle's folders under *modules/memoryModules*. We will place our plugin there, and will give it the name *urlRetriever.py*, as shown in Figure 38.

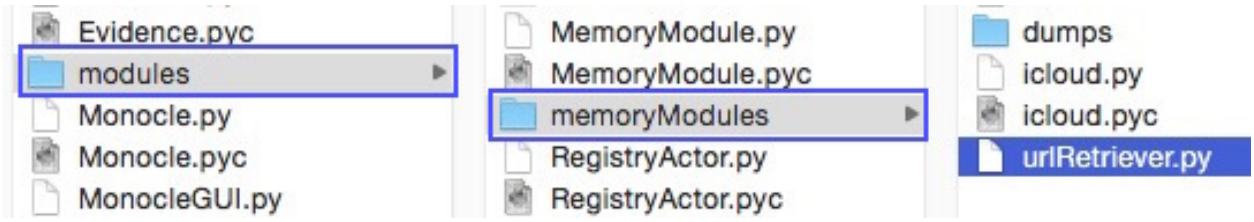

Figure 38: Placing *urlRetriever.py* within Monocle folders.

Once the file is placed, it is time to write some code. In order to make the plugin compliant with Monocle interface the first thing to do is to import the proper packages. These packages are the module, which will be used in order to create a retriever object (Monocle's entry point) and the EvidenceManager, used to mark elements as evidences. To do so, only the following two lines are necessary

```
from modules.MemoryModule import memoryModule
from Evidence import Evidence
```

Once these packages are imported, it is necessary to state where our plugin is going to start. When executing a plugin, Monocle will automatically search for a *retriever* object. This object is persent in the *memoryModule* and in the *hdModule* classes, containing target-specific tools for memory and disk, respectively. In our case, and as we are creating a memoryModule (i.e. imported the memoryModule package), we will create a retriever object inheriting from *memoryModule*. In addition, it is necessary to override the *retrieveData* function, which is the function Monocle will call when first running the plugin. The following code creates a retriever object inherinting from *memoryModule* and provides a *retrieveData* function for Monocle to execute. This function is going to contain our plugin's functionality.

```
class retriever(memoryModule):

    def retrieveData(self):
            """This function will be the one executing when
            Monocle starts. Plugin code goes here."""
```

With this, the plugin is ready to execute. The code contained in *retrieveData* function. The plugin however **does not work yet**, as we are not returning any evidence. Monocle expects the returning value of the *retrieveData* function to be a collection of Evidence elements.

We can test our plugin by passing a dummy Evidence element. The evidence parsing is performed by means of the Evidence Manager. This manager is called when instancing Evidence elements, so let's create an Evidence element and return it. Notice that Monocle expects a list of Evidences to be returned. The retriever class we previously created already contains an attribute namely *evidencesFound*, which is a list to store these evidences. So in order to parse our dummy evidence, we will use the *append()* function of the *evidencesFound* attribute in order to include the new *Evidence*.

The plugin so far looks as follows.

```
from modules.MemoryModule import memoryModule
from Evidence import Evidence

class retriever(memoryModule):

    def retrieveData(self):
        self.filesFound.append(Evidence('Hello World',
        'DummyEvidenceSource','DummyEvidenceDestination'))

        return self.filesFound
```

If we execute Monocle, this plugin will appear under the Memory tab. If the plugin is executed (it is necessary to specify a dump memory file although we are not actually accessing it), the Dummy evidence we added will appear in the result tab as expected (see Figure 39).

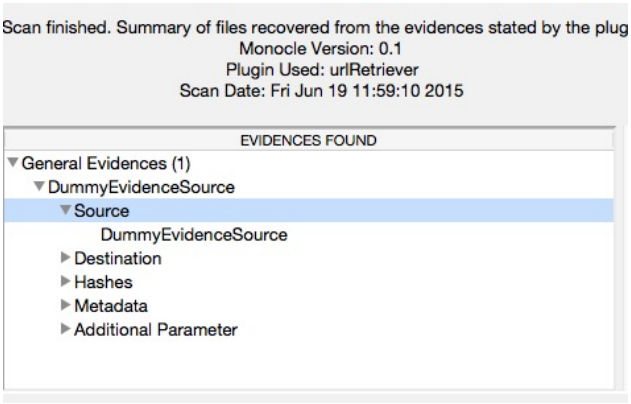

Figure 39: Dummy Evidence showing in the result window.

B.6.3. Coding the plugin. Obtaining running processes

We have already a plugin integrated within Monocle so it is possible now to access the evidence and perform operations over it. The evidence element can be accessed in this case by means of the *memfile_fd* attribute, which gives a read-only file descriptor to the digital container stated in Monocle's main page. Additionally, the path of this digital container can be accessed through *memfile_path* attribute.

Our *urlRetriever* needs to reconstruct the web browser processes in order to perform the search. This can be done by means of Volatility as described above. The *VolActor* component of the retriever is in charge of managing all Volatility elements in order to run the framework. Volatility adds several overhead to the execution, so it's usage has to be explicitly stated in the plugin. It is enough to add the following line in order for the *VolActor* to setup the necessary environment.

```
self.start_volatility_framework()
```

Once the *VolActor* is settled, it is possible to execute Volatility within our framework. The plugin needed in order to obtain the list of processes running in the machine is called PSScan, and the plugin to dump the memory address space of a process is called MemDump. Such plugins have to be loaded into the VolActor. This is done by means of the *loadPlugin* function of the *VolActor*, whose arguments are the package the plugin belong to, the class of plugin it is and the name of the plugin itself, in such an order. Information regarding volatility plugins and their characteristics is available in Volatility web site [31].

In order thus to include the PSScan and the MemDump plugins in our VolActor, the following lines are needed.

```
self.volActor.loadPlugin("volatility.plugins", 'filescan', 'PSScan')
self.volActor.loadPlugin("volatility.plugins", 'taskmods', 'MemDump')
```

Volatility provides two ways of executing its modules. Either the *calculate()* function or the *renderText()* function. This is useful for some plugins (i.e. Dump plugins) where the actual dump over the data is performed in this step. Such functions are interfaced in the VolActor by means of the *executePlugin()* and the *executeRender()* respectively. Let's execute the PSScan plugin to retrieve all the processes that were running in the system when the memory dump was obtained. The following line is enough to do so.

```
process_list = self.volActor.executePlugin('PSScan')
```

This will return a set of elements with the attributes defined in PSScan plugin, e.g. the PID and the process name. It is necessary to notice that it is necessary to access to the __str__ parameter of elements returned by Volatility in order to get the Python string. Now we have the list of processes in the system. If we were just interested in such elements, we can just set them as evidences and we would have a plugin which returns the running processes in the system. This plugin would look as following, and will have the results depicted in Figure 40.

```
from modules.MemoryModule import memoryModule
from Evidence import Evidence

class retriever(memoryModule):

    def retrieveData(self):

        self.start_volatility_framework()
        self.volActor.loadPlugin("volatility.plugins", 'filescan',
        'PSScan')
        self.volActor.loadPlugin("volatility.plugins", 'taskmods',
        'MemDump')

        process_list = self.volActor.executePlugin('PSScan')

        for x in process_list:
            process = x.ImageFileName.__str__()
            pid = int(x.UniqueProcessId)
            self.filesFound.append(Evidence(process, process, "out",
            additional_parameters={"PID" : pid}))


        return self.filesFound
```

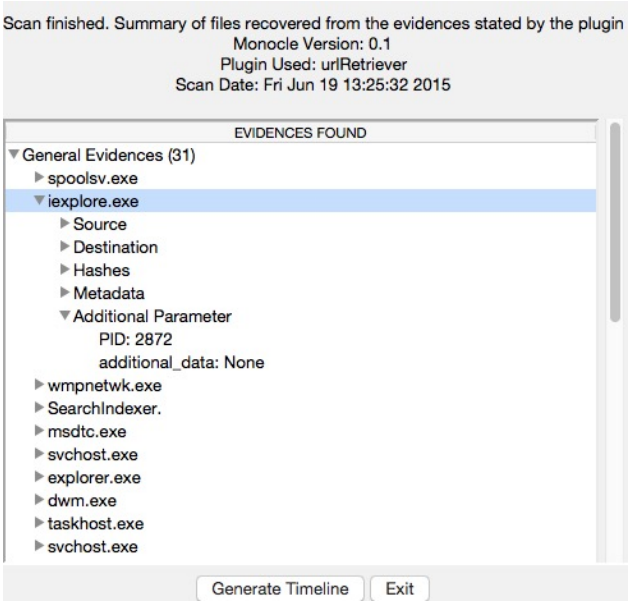

Figure 40: Processes running in the memory dump.

B.6.4. Coding the plugin. Dumping selected processes

Now it is time to select the processes we are interested in and to dump their memory address in order for it to be reconstructed and properly scanned. This is done by checking the list obtained in the previous section against the processes names we are interested in. For this example, we will only focus in Internet Explorer (iexplore.exe), Mozilla Firefox (firefox.exe) and Google Chrome (chrome.exe). By checking the list obtained we can obtain the PID of each and thus, invoke *MemDump* plugin of Volatility to dump the memory address.

MemDump plugin is a special kind of plugin which requires both of the *calculate()* and the *render_text()* functions to work. Once it has been executed, it is necessary to specify certain parameters which are used by the plugin during execution. Those are the *DUMP_DIR* and the *PID*. In order to specify such parameters, the VolActor has a function named *setArguments*, which allows to include an arbitrary set of parameters to be loaded. The inclusion of the following lines of code are enough to dump a process memory address once its PID is known.

```
dataDump = self.volActor.executePlugin('MemDump')
self.volActor.setArguments(DUMP_DIR= dump_name, PID=pid)
self.volActor.executeRender('MemDump', dataDump)
```

Notice that in order to dump the processes we are interested in, it is enough to add extra constraints for name checking and to include the above code within the script. The result will be as follows, and results the appropiate results as Figure 41 show.

```
from modules.MemoryModule import memoryModule
from Evidence import Evidence
import os # Used to get the local path

class retriever(memoryModule):

    def retrieveData(self):

        self.start_volatility_framework()
        self.volActor.loadPlugin("volatility.plugins", 'filescan',
        'PSScan')
        self.volActor.loadPlugin("volatility.plugins", 'taskmods',
        'MemDump')

        process_list = self.volActor.executePlugin('PSScan')

        processNames = [ 'iexplore.exe', 'firefox.exe', 'chrome.exe']
        # This path is an arbitrary one
        dump_name = os.path.dirname(os.path.realpath(__file__))

        for x in process_list:

            process = x.ImageFileName.__str__()
            pid = int(x.UniqueProcessId)

            if process in processNames:
                dataDump = self.volActor.executePlugin('MemDump')
                #Setting parameters
                self.volActor.setArguments(DUMP_DIR= dump_name, PID=pid)
                #Actual dump
                self.volActor.executeRender('MemDump', dataDump)

                    self.filesFound.append(Evidence(process, process, "out",
                    additional_parameters={"PID" : pid}))

        return self.filesFound
```

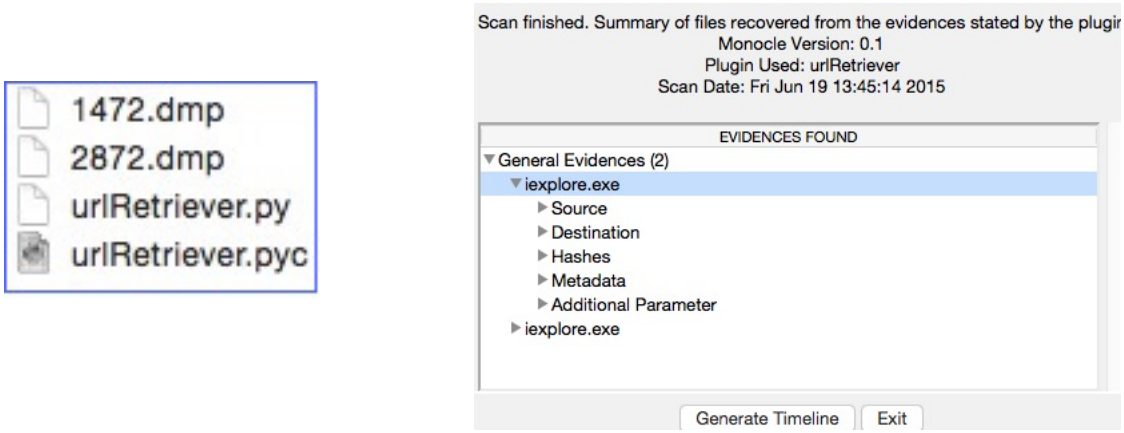

Figure 41: Processes dump files and report.

B.6.5. Coding the plugin. Carving URLs

Finally, the only step remaining is to obtain the actual URLs. As the memory is already reconstructed, this is as easy as to use the *re* library of python in order to construct a regular expression and to check the dumps against such regular expression. The final code of the plugin will be as follows, and Figure 42 shows the results obtained.

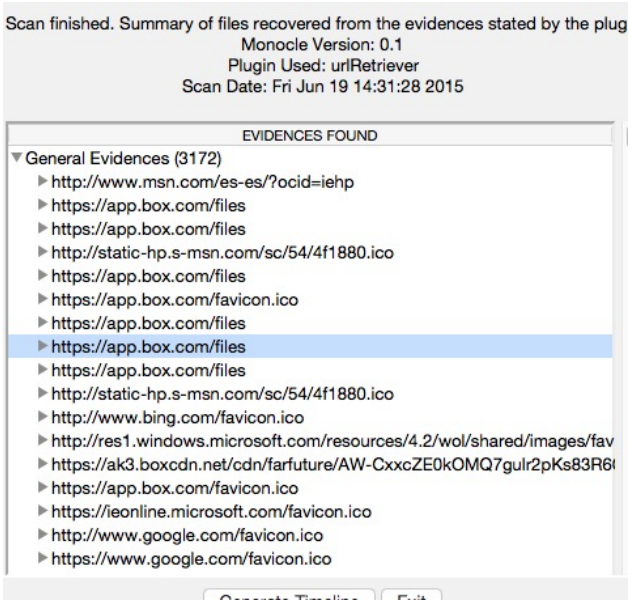

Figure 42: URL obtained with the plugin.

```
from modules.MemoryModule import memoryModule
from Evidence import Evidence
import os # Used to get the local path
import re # Used to find URLS with regular expressionsste

class retriever(memoryModule):

    def retrieveData(self):

        self.start_volatility_framework()
        self.volActor.loadPlugin("volatility.plugins", 'filescan',
        'PSScan')
        self.volActor.loadPlugin("volatility.plugins", 'taskmods',
        'MemDump')

        process_list = self.volActor.executePlugin('PSScan')
        processNames = [ 'iexplore.exe', 'firefox.exe', 'chrome.exe']
        # This path is an arbitrary one
        dump_name = os.path.dirname(os.path.realpath(__file__))

        for x in process_list:

            process = x.ImageFileName.__str__()
            pid = int(x.UniqueProcessId)
            if process in processNames:
                dataDump = self.volActor.executePlugin('MemDump')
                #Setting parameters
                self.volActor.setArguments(DUMP_DIR= dump_name, PID=pid)
                #Actual dump
                self.volActor.executeRender('MemDump', dataDump)

                dump_descriptor = open(dump_name  + "/" + str(pid) + ".dmp")
                for line in dump_descriptor:
                    urls = re.findall('http[s]?://(?:[a-zA-Z]|[0-9]|
                    [$-_@.&+]|[!*\(\),]|(?:%[0-9a-fA-F][0-9a-fA-F]))+',
                     dump_descriptor.read())

                    for url in urls:
                        self.filesFound.append(Evidence(url, url, "out",
                                additional_parameters={"Source":process}))

        return self.filesFound
```