POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

BackToLife - an innovative technique for memory forensics

Supervisors
prof. Antonio Lioy
ing. Andrea Atzeni

Candidates
Luca Doglione
Marco Senno

Academic year 2016-2017

This work is subject to the Creative Commons Licence

Summary

Today advancements in the use of the internet and technologies are leading to an increase in cyber crimes. Digital forensic techniques are now uniformly adopted in investigations for acquiring courtroom evidence. The number of crimes committed using digital devices has increased and information gathered by investigators can be used to prosecute a criminal. The goal of digital forensics is to execute an accurate and in-depth investigation while keeping a trail of evidence to find out exactly what can be found on a computing device. Digital investigation was born in 2001, when the first analysis model was created. The latter was improved in the following years and is now a sequence of well-defined steps and phases. The major improvements can be found in the acquisition and analysis stages. Many techniques have been designed for both phases. In the analysis phase, for example, investigators moved from a static to a live analysis. The latter has as its objective the gathering of evidence from the volatile memory of the target system. At the beginning of the digital investigation, only static analysis was performed, where physical storages were analysed for collecting suspicious data. Nowadays static analysis has been set aside due to increased storage size and disk encryption. Live analysis has spread thanks to the advantages it provides. The latter include, for example, the fact that live analysis can be performed without turning off the system and includes live interrogation and memory imaging. Live analysis is the more powerful technique moreover because it allows investigators to repeat the analysis several times, leaving the system state intact. Using memory imaging analysis, investigators are able to collect evidence proving their origin. Memory imaging minimises the impact on the analysed system, requiring actions be taken only for dumping the physical memory. As the RAM is dumped, the state of the system is captured, allowing the analysis to be repeated. The memory image analysis' major strength is its ability to gather information related also to terminated processes.

Both the tools and techniques of memory forensics have been improved in the last decades. In the investigation field in particular, strategies have tried to respond to the need of analysts and investigators to understand exactly what is performed on a target system, recreating a state similar to that one existing before the memory acquisition. This particular group of techniques includes many strategies presented over the years such as `DSCRETE` or `RETROSCOPE`. Both are able to generate a graphical output just analysing the memory dump. The first renders graphical data used by a target process, while the second is able to generate screenshots of last-used mobile applications. Another technique uses both live and static analysis to recreate the system environment before memory acquisition. With this technique, the system state is manually recreated, becoming similar, but not identical, to the original one.

This thesis presents `BackToLife`, a project which can be collocated with the aforementioned strategies. The objective of `BackToLife` is to restore the target system's previous state, allowing the analyst to interact with it. This thesis presents an innovative technique in the field of memory forensics. The main idea is to resurrect a process on the analyst's computer starting from a target computer's physical memory dump. Using `BackToLife`, the analyst simply needs to select a target

process, wait for its resurrection, and then analyse it, as in a live response analysis with the advantage of repeatability.

The stated objective was achieved through a model which makes use of two existing tools: the CRIU Project and the Volatility Framework. The former is a checkpointing tool that is able to generate a target process checkpoint. In so doing, it allows the user to later restore a target process to the same identical state. This kind of tool is often used for generating a snapshot of a running application that can be used in case of a system failure. The second tool used for the implementation of this thesis is the Volatility Framework. `Volatility` is one of the most commonly used forensic frameworks. It is different from specific tools, which are able to perform a particular and well defined forensic analysis. It is a generic tool that offers to analysts a shell-like interface to use for a physical memory dump, allowing them to thus perform an analysis similar to live interrogation.

Researchers developed an analysis model composed of six stages, which are very similar to the ones presented in other memory forensics strategies. Specifically, memory has to be acquired and analysed in order to list all available processes. After the target process is selected, the analyst can firstly extract it and then restore it to the previous state of memory acquisition. Following the resurrection, the process can be more easily analysed interacting with it. From a development point of view, the workflow is composed of three different phases. Firstly, a `Volatility` plugin has to be generated in order to extract all needed information. Secondly, `CRIU` image files should be created in order to build a valid checkpoint. Thirdly, the checkpoint can be restored using `CRIU`.

Researchers have created a model called *DACA* that has been used for the development of this project. The *DACA* model (Dump, Analysis, Composition, Automation) represents the steps needed for understanding what the missing parts for obtaining the expected result are. By following the model it has been possible to correctly develop scripts and tools for the research in question. This model is composed of several steps. Firstly, researchers choose and begin a target process. They then perform the RAM dump. When the latter is completed the process is checkpointed with `CRIU`. When these phases are completed, researchers start the analysis and extraction phases. Each `CRIU` image file is analysed, and related data are researched in the memory dump using the Volatility Framework. Specifically, the volshell plugin is used for searching data structures directly in the memory dump. After the needed data have been found, they have to be composed in `JSON` structures encoded in image files using the program `CRIT`, a companion tool available in the `CRIU` suite. The last phase of the *DACA* model is the development of an automation process that is able to extract needed memory areas and to compose them for generating a valid checkpoint usable by `CRIU`.

Researchers applied the *DACA* model to various programs of increasing complexity, refining `BackToLife` scripts. At the beginning, it was used as a simple counter program written in C. By checkpointing it with `CRIU`, many image files were created and analysed in order to collect all needed data from the memory dump. A basic `Volatility` plugin called `linux_backtolife` was created. It is the main plugin for automatically extracting data. For each image file generated by `CRIU`, a part of the `linux_backtolife` plugin was written for extracting related data. In some cases a separated plugin was written because developers thought it could be useful also for others analysts. Data collection and extraction were sometimes very difficult due to missing parts in the Volatility Framework, and for each file a good amount of kernel structures and theory were studied. When all image files were analysed and automatically generated, researchers tested the developed scripts and plugins with many programs with the objective of refining the tools. After the generation of the first pool of image files, researchers performed tests with some programs such as multi-thread counters or other Linux common commands, such as the `Nano` editor, the `Python` shell and many others.

The *DACA* model has been applied also with network-based process or programs which make use of unix sockets. Regarding network-based programs, a simple client-server application has been created which uses TCP connections. After the `CRIU` checkpoint, the new generated image files have been analysed to understand how to create the network part. The latter requires the study of Linux kernel internals in order to understand how information about opened TCP connections are stored in the kernel memory. The Volatility Framework has been adapted for supporting some data structures that are not included in the default installation. Those structures are used

by the `linux_backtolife` Volatility plugin. In order to extract needed data, an ad-hoc plugin called `linux_dump_sock` has been implemented for extracting network informations. Both client and server programs have been resurrected from a memory dump. Also, other network-based processes were tested such as `OpenSSH` SSH client or `Lynx` browser. Network processes could be successfully extracted with `BackToLife`, it is possible to perform a resurrection of a TCP based process, but conditions allowing for the complete restoration of the network connection are rare. Many assumptions were made by analysts, for example that processes are able to manage inside the connection, re-establishing it when it is closed. Another assumption concerns the network configuration. A network connection is based on addresses, and when a process is restored, the previous network environment has to be recreated in order to re-establish the connection. Concerning unix sockets, many problems were encountered since `CRIU` does not totally support unix sockets. Researchers developed a `Volatility` plugin called `linux_dump_unix_sock` that is able to extract information about unix socket. Data structures are similar to TCP connection structures, but researchers encountered some problems during the resurrection of processes using unix sockets. The plugin has been tested with a basic client-server application that uses unix sockets. The main problem is caused by the fact that both client and server run on the same machine, so it is not possible to keep the connection alive for one of the peers. Another problem is that it is impossible to resuscitate both the client and the server at the same time. When the client-server protocol is stateful after the resurrection has been performed, the server and the client are in two different states they cannot communicate correctly. Otherwise, when the client-server protocol is stateless, the resurrection works properly.

The last part of this research concerns applications with a graphical user interface. Using their own strategy, researchers tried to understand if it was possible to resurrect this kind of processes. `CRIU` does not support graphical application restores and checkpoints, requiring a particular trick be used for this kind of program. The idea is to use the `Xvnc` server as an `X-server` for running graphical applications. The checkpoint and restore have to be done on the entire process tree containing the `Xvnc` server itself. The idea for resuscitating `X-based` process uses this trick. In particular, researchers tried to understand if it was possible to insert an extracted graphical process inside a previously generated checkpoint which contains a `Xvnc` server. The first step toward the objective was to understand how to insert a process in an existent tree, and how to connect the graphical program with the `Xvnc` server. In order to implement this strategy, a forwarder process was implemented in order to adapt the communication between `X-process` and `X-server`. The forwarder is needed since the environment of the extracted process is different from the environment of the `X-server` on the analyst's machine. This implementation was not sufficient for performing a complete resurrection, because of the complexity of the `X-protocol`. In a future implementation a better approach could be to develop a forwarder that interprets the X-protocol in its completeness.

`BackToLife` is able to revive several kinds of processes starting from a physical memory dump. The analyst simply needs to select a target process. After that, `BackToLife` analyses the memory, and the target process is extracted from the dump and resurrected on the analyst's computer in order to perform the investigation. Several tests were performed with different kinds of processes, and the main kinds of command line programs can be resurrected using this project. The research has gone beyond command-line-based processes resurrection. The study also covered programs with graphical user interface. In this research a model has been developed that aims to resurrect a graphical process into the analyst machine. `BackToLife` is the first project in this field and several developments are possible. A porting on different operating systems could help to further develop such a technique, also in mobile fields. Another possible development regards the possibility to run BackToLife on a machine with different characteristics from the target machine such as the kernel or the libraries' version. The purpose of this research has been to understand the feasibility of resuscitating a process from a memory dump onto another machine with the same kernel. The presented results have shown that it is possible with the right conditions, providing performing memory analysis with a totally different approach.

Contents

Chapter 1

Introduction

1.1 Technological context

Nowadays the advancement in the use of internet and technologies leads to an increase of cyber attacks. Digital forensic has an important role in the investigation field, it is often involved for finding evidences about computer crimes.

Through memory forensics it is possible to acquire malicious evidences which can be presented to the court in order to prosecute a criminal. The goal of digital forensics is to execute an accurate investigation while keeping a chain of evidence to find out exactly what can be found on a computing device. Different techniques and technologies are employed: the older approach is the static forensics which requires that target devices are shut down for the analysis, it is based on analysis of storage media for collecting evidences. Modern techniques propose to analyse volatile memory, specifically it is possible to distinguish two main categories: live response and memory imaging. These techniques permit finding evidences which can not be collected through static analysis [1], which nowadays presents many drawbacks [2][3], as for example the need of a properly executed shutdown, the very big size of storage disk or disk encryption. The other kind of techniques proposes to analyse what is running on machine, with two different approach which lead to different results [3]: memory imaging and live interrogation. In particular one of the most important feature of memory imaging compared to live interrogation, is the possibility to analyse also what was performed in the past on the investigated system [4].

The memory forensics has evolved in last decades and the first investigation model [5], is become today a very solid and well defined process [6] comparable to traditional investigation. The variety of tools increased during the years and in particular they are always more powerful and easy to be used by the community, as for example the Volatility Framework [7]. The evolution of memory analysis tools, in the years, was lead by different fields where memory forensics could be applied, the investigative scenario was the most important, but malware and rootkit detection have also to be considered, and many forensic tools were developed also for finding files, user's input or cryptographic keys.

With the evolution and the increasing of the variety of available tools, also techniques evolved in the years. Memory forensics techniques in the investigation field tried to respond to the need by analysts and investigators to recreating an environment similar to the system state before the memory acquisition [3]. This could be very important in order to really understand what the investigated user was performing and why. Other techniques have as objective the restoration of previous state in particular from a graphical point of view [8][9][10][11], for example extracting data and rendering them in order to generate graphical content easy to be understood by humans. The possibility to restore a previous state, is an important improvement for the investigation field of memory forensics, where investigators have the need of analyse collected data many times in order to gather evidences which can be certainly accepted in a court of law.

1.2 Forensic through resurrection

This thesis presents an innovative technique in the field of memory forensics. The project is about the resurrection of processes starting from a physical memory dump.

In this report is presented BackToLife, a set of scripts and plugins that can be used by investigators in order to perform the resurrection of a process starting from a volatile memory dump. Using BackToLife the analyst have just to select a target process. After that BackToLife has analysed the memory, the process is extracted and isolated from the memory dump file and resurrected on the analyst's computer in order to perform the investigation, as depicted in the Figure 1.1.

Giving the opportunity to resuscitate a process permits to analysts to directly visualise the outputs and to interact with the program. This is a great improvement in the forensic investigation because the analysis turns into live analysis in which it is possible to observe the behaviour of the process. Obtaining the interactive output of a process can be really important in a forensic analysis because it gives to analysts the possibility to perform the investigation directly on a running process. It means that investigators can collect data and directly watch what the user was performing with the investigated process. Secondly, it is possible to perform a repeatable live analysis on a target process which is restored exactly how it was, before that the memory dump was taken.

Figure 1.1. BackToLife model.

Using BackToLife the forensic expert does not need to understand low level data structures in order to perform a volatile memory analysis. The investigator, ideally, have only to decide which is the target process, restore it and perform the analysis at a higher abstraction level.

This project combines memory analysis techniques with check pointing technologies. In particular, the project leverage on Volatility Framework in order to analyse the physical memory dump and `CRIU` project in order to restore processes. Using `Volatility` it is possible to extract required information that have to be manipulated for being usable in the `CRIU` project.

Since the memory acquisition phase is really important in the section 2.2 are described different techniques for dumping physical memory. In section 2.3.2 used tools are described. Firstly, Volatility Framework functionalities are explained and several usage examples are described. Subsequently, it is analysed how `CRIU` works and how it should be used. In chapter 3 several techniques are described that are comparable to BackToLife. The chapter 4 treats the development of BackToLife from the beginning of the project to the analysis of resurrection of processes with graphical user interface. The last section of the chapter 4 describes the Docker container developed that is directly usable by the analyst. In the last chapter 5 limits of the project and possible future development are described.

Chapter 2

Background

2.1 Introduction

2.1.1 Why memory image analysis?

Nowadays digital forensics has an important role in investigation in order to find evidences about computer crimes. Different techniques and technologies are employed, the first approach to memory forensics was the static forensics which requires that target devices are shut down for the analysis. Another category of techniques proposes to analyse what is running on machine, in order to better identify digital evidences not available on disk [1], these are live forensics techniques.

Static forensic is based on analysis of storage media for collecting evidences on saved or deleted (and extracted) files, it requires system shutdown which have to be executed in a proper way in order to guarantee data consistency. Another drawback of static analysis is related to the disk size increasing [2], hard-drive contains now terabytes of data and it causes long times for taking images, it becomes more difficult when investigation requires analysis of RAID arrays, SAN or NAS. Encryption is also a problem for static technique, because it makes almost impossible to access data [3]. Furthermore, dynamic data are very important for digital investigation, but, with static analysis all informations about network connections, kernel modules and machine dynamic state are lost.

Live forensics techniques permit gathering information about dynamic state and machine context directly from volatile memory, in order to obtain a good snapshot of the running system. In particular, it is possible to distinguish two different approaches, live interrogation and memory imaging. Live interrogation was the most common used technique for many years, and it consists in gathering dynamic data directly at runtime from target machine. This approach resolves the main issue of static analysis, however, it reveals some drawbacks. Interrogation consists in using the system for gathering information, this is possible through use of forensics tools, but them could cause changes in machine state. These changes are not admissible because are against accepted principles of digital investigation [3]; moreover, if system state changes, analysis is not repeatable and results can not be verified. Another drawback of this technique derives from terminated processes, their locations are left as unallocated space in memory and unmapped from kernel space, as a consequence it is impossible to retrieve information about terminated process using live interrogation. Memory imaging is the alternative approach, it consists in analysing a physical memory dump in order to find evidences, trying to mitigate some of described drawbacks. This technique reduces the impact on analysed system, it requires only actions for dumping the physical memory, the state of the system is captured as it is, and it can be analysed several times. The investigation process is repeatable without the risk of loosing evidences, this is important for ensuring the possibility to present results in a court of law. Offline analysis permits also to bypass operating system of compromised machines, in particular it is not affected by anti-forensics methods. One another positive aspect of this technique concerns terminated and cached processes, using memory imaging their locations can be mapped and used in the investigation process [4]. Static forensics can not

be compared with this kind of technique, informations which can be extracted are different for the two approaches. It is possible to use both in order to guarantee a global point of view during the investigation process [3]. Memory imaging approach relies on different tools which allows investigators to interact with memory dump like as in live interrogation, there are several tools developed for different operating systems or memory image formats.

2.1.2 History of memory image analysis

Digital forensics techniques and tools have considerably evolved in last decade, and nowadays memory analysis is comparable to traditional investigation. All evidences gathered during an investigation process are now accepted by courts of law; it is possible thanks to a community composed by forensics experts and investigators which studied digital forensics in order to propose rigorous methodologies and techniques. The first important step for digital forensics is the Digital Forensic Research Workshop (DFRWS) in 2001, held in Utica. The audience of the workshop was composed by academic, civilian and forces professional of digital forensics. The goal was to create a community of academic and forensics experts to share their knowledge in this field [5]. In 2005 there was the first challenge concerning the memory image analysis, one year later that Carrier et al. presented Tribble [12], a hardware-based memory acquisition tool based on PCI. The challenge consisted on answer several questions about a compromised Windows memory image, developers had to build tools thinking to new technologies in order to extract informations from memory dump. The challenge led to development of two different tools (Memparser, Kntlist) capable of read and traversing processes structures in kernel space to extract needed informations [13]. Movall et al. presented, in the same year, a tool suite for the analysis of a physical memory dump of a system running a Linux environment, with the purpose of understand its usage by operating system and not with investigative intentions [14]. From 2005 to 2006 many forensic tools were presented, most of them was developed for using address translation. These were all specific tools able to analyse low level data in order to find kernel structures signatures and to gather particular informations like the process list, registry informations and other specific data. In 2006, FATkit framework was presented by Petroni et al. [15]. This tool supports Linux and Windows operating system, it is a modular tool able to perform a comprehensive analysis of low level data reconstructing the address space. It permits investigators to execute an high-level analysis of memory images. FATkit was the first tool to mention using the pagefile [16], this technique permits reaching invalid pages for collecting more evidences. Petroni's team build a first example of a generic tools which offers the possibility to extract different kinds of data. The evolution of FATkit is Volatility [7], it was published for the first time in 2007, and it is now the most used framework in memory forensic area. It is a generic tool which can extract running processes, memory maps for each process, open network sockets or kernel modules. Volatility is developed in Python and it is composed by many plugins, it is a very modular framework which can be easily extended. Afterwards, from 2008, when DFRWS proposed a challenge based on Linux environment, researches started to create and to transfer techniques end methodologies to linux-based system, MAC OS X and FreeBSD. Savold and Gubian in [17], focused on memory pagefile extraction in order to guarantee a more quickly analysis of data. Malware and rootkit detection have driven researchers to investigate in forensic field, MAS [18] and KOP [19] are two examples of kernel integrity check tools based on analysis of a memory dump, both of them require kernel code. From 2009, many researches started to investigate memory forensic in the virtualization field. In particular, different techniques proposed to use virtual machines in the investigation process [20]. One of the main weakness of static analysis is the encryption of devices storage. In 2011 [21] an important work was conduct by Balogh et al. They proposed to find encryption keys from a physical memory dump, deciphering a storage encrypted with TrueCrypt. In 2011, the Volatility framework was forked in order to make the code base more modular, improve performance and increase usability, the resultant product was called Rekall [22]. Recently, researchers of memory forensics field have extended Volatility for adding functionalities. Bianchi et al. built a tool, called Blacksheep [23] containing Volatility and many new plugins to identify rootkit evidences. Ligh, in 2013, introduced a new option for enabling pool tag scanning in order to search in the entire kernel virtual address space [24]. Case et al. in 2014 [25] proposed a new tool for analysing swap files and compressed RAM in Mac OS X and Linux environment.

The evolution of memory analysis tools, in the years, was lead from different fields where memory forensics techniques can be applied. At the beginning, the investigative scenario was the most important, but malware and rootkit detection have also to be considered. Different forensic tools were developed for finding, for example, files, user's input or cryptographic keys; some special researches led to develop tools for rendering data or graphical screenshots of windows operating system. Nowadays researchers are moving on mobile forensic, in particular on Android devices, because data stored on smartphones could contain evidences crucial for resolving a criminal case.

2.2 Memory acquisition

2.2.1 Introduction

This section firstly describes how the volatile memory is organized and managed by the operating system. Then is reported which are the criteria for evaluating an acquisition method. Subsequently, different acquisition methods are presented focusing on differences between them.

2.2.2 Memory management and virtual memory

In order to understand how should work a memory acquisition tool it is important to have a general overview about which are the memory management techniques, how the memory is organized and how is possible to address a memory cell. The memory-management algorithms vary from a basic approach to paging and segmentation strategies. Each approach has its own advantages and disadvantages and the choice between them depends on many factors such as the hardware design of the system. As reported in [26] the Random Access Memory (RAM) is a memory composed by an array of words or bytes, each with its own address. The CPU fetches instructions from memory depending on the value of the program counter (PC). Additional loading or storing to specific address can be caused by the instruction. Main memory and the processor registers are the only storage that the CPU can access directly. CPU registers are accessible within one cycle of the CPU clock but the same cannot be said for a memory cell, which is accessed via a transaction on the memory bus taking more than a CPU clock. In this case, the processor is in sort of stall condition. In order to prevent this situation a fast memory in added between CPU and RAM memory called cache. Furthermore, it is needed to ensure correctness of operation to protect the operating system from access by user processes and, in addition, to protect user processes from one another using hardware support. In order to do this, each process has a separate memory space.

Base and limit technique

It is needed a mechanism to determine the range of legal addresses that the process may access and to ensure that the process can access only these addresses. The simplest method for this kind of protection uses two registers called base and a limit register, as illustrated in Figure 2.1. The base holds the smallest legal physical memory address while the limit specifies the size of the range. CPU must check every memory access generated in user mode in order to be sure it is between base and limit for that user.

Protection of memory space is given by the fact that the CPU compare every address generated in user mode with the registers. Any attempt by a program executing in user mode to access operating-system memory or other users' memory results in a trap to the operating system. The scheme in Figure 2.2 shows the address checking made by the CPU for preventing a user program from modifying the code or data structures of either the operating system or other users. The base and limit registers can be loaded only by the operating system, which uses a special privileged instruction.

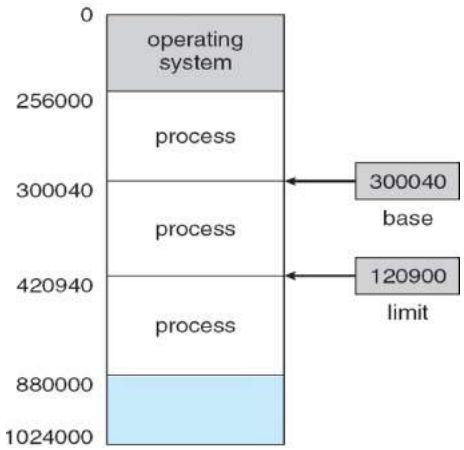

Figure 2.1. Base and limit mechanism (source: [26]).

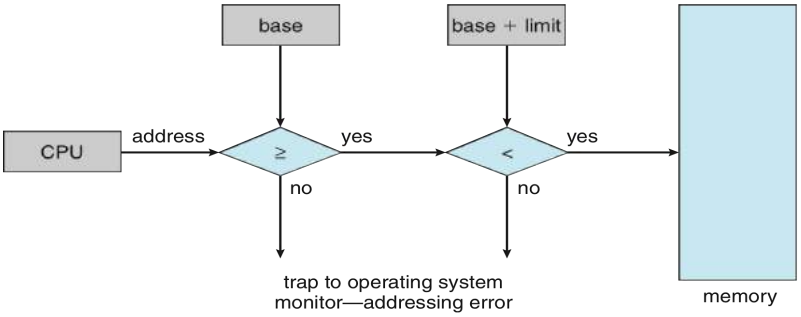

Figure 2.2. Base and Limit address checking (source: [26]).

Logical and physical address space

It is important to distinguishes between logical and physical address. An address generated by the CPU is commonly referred to as a logical address and the whole set of all logical addresses is call logical address space. A physical address is the address seen by the memory unit and the entire set of physical addresses corresponding to the logical addresses generated by a program is called physical address space. A special component in the CPU called the Memory Management Unit (MMU) is responsible of translating logical address into the physical address. The basic scheme is represented in Figure 2.3. The base register is now called a relocation register and the value is added to every address generated by a user process.

Contiguous memory allocation

The memory is divided into two partitions: one for the resident operating system and one for the user processes. We usually want several user processes to reside in memory at the same time because this is a multiprogramming measure. In contiguous memory allocation, each process is contained in a single contiguous section of memory. This method divides memory into several fixed-sized partitions containing one process per partition. The operating system uses a table in order to indicate which parts of memory are available and which are occupied. Several strategies are available for selecting the partition in which allocate the process.

- First fit: the operating system allocate the process into the first hole that is big enough.
- Best fit: the operating system allocate the process into the smallest hole that is big enough.

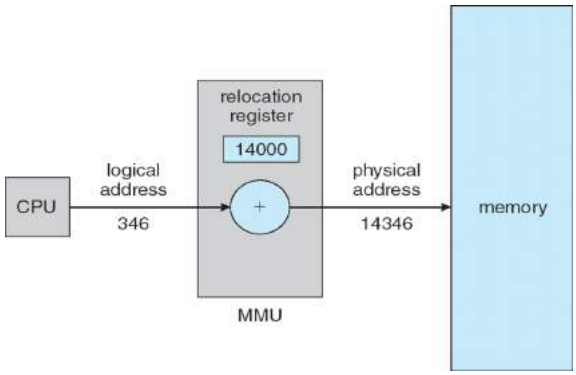

Figure 2.3. MMU scheme (source: [26]).

- Worst fit: the operating system allocate the process into the largest hole.

Both the first-fit and best-fit strategies for memory allocation suffer from external-fragmentation. A solution to the external-fragmentation problem is to permit the logical address space of the processes to be non-contiguous. Two possible techniques are paging and segmentation and they can be combined.

Paging

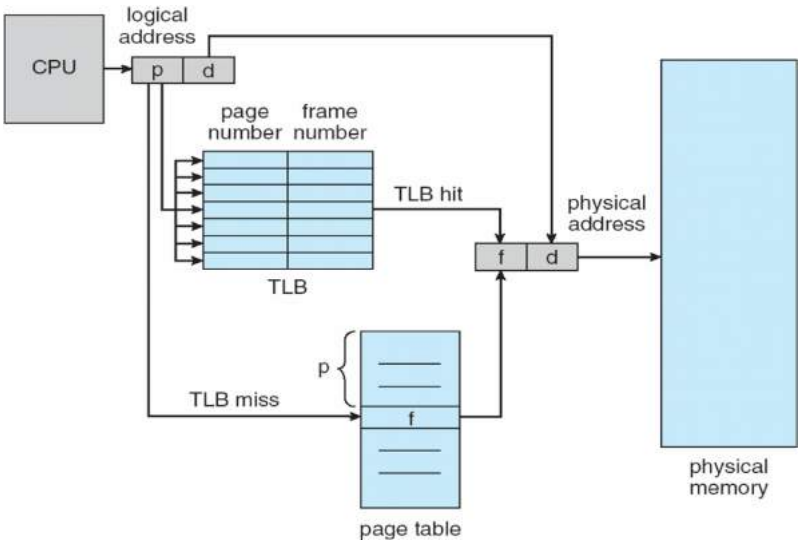

Figure 2.4. Paging mechanism (source: [26]).

Paging is a memory-management scheme that permits the physical address space of a process to be non-contiguous. The basic method divides physical memory into fixed-sized blocks called frames and divides logical memory into blocks of the same size called pages. When a process is to be executed, its pages are loaded into any available memory frames. The hardware support for paging is illustrated in Figure 2.4. Every address generated the CPU is divided into two parts: the first part is the page number and the second part is the page offset. The page number is used as an index into a page table. The page table contains the base address of each page in physical memory. Given that the page table resides in memory is it needed a mechanism for avoiding to access the page tables for every single memory access repeatedly. In order to do that the MMU maintains a cache called Translation Lookaside Buffer (TLB) that is an associative array. When the CPU generate a logical address the MMU try to find the page physical address in the TLB. If it occurs a TLB hit, the physical address is generated joining the page physical address with

the displacement. If it occurs a TLB miss, the page table is accessed in order to get the physical address page.

Segmentation

Segmentation is a memory-management scheme in which the program is divided in logical units such as main program, functions, objects and stack as shown in Figure 2.5. Each logical unit is called segment. Each segment has a name and a length. The addresses specify both the segment name and the offset within the segment.

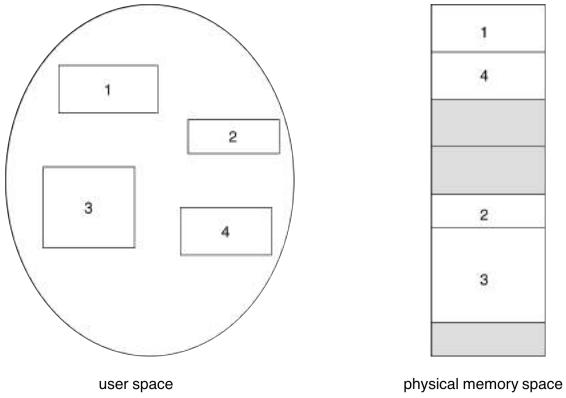

Figure 2.5. Logical view of segmentation (source: [26]).

Using the segmentation mechanism it is necessary a structure called Segment Table in which each entry has a segment base and a segment limit. The segment base contains the starting physical address where the segment resides in memory, and the segment limit specifies the length of the segment. The use of a segment table is shown in Figure 2.6. A logical address contains a segment number and an offset. The segment number is used as an index to the segment table. The offset d of the logical address must be between 0 and the segment limit.

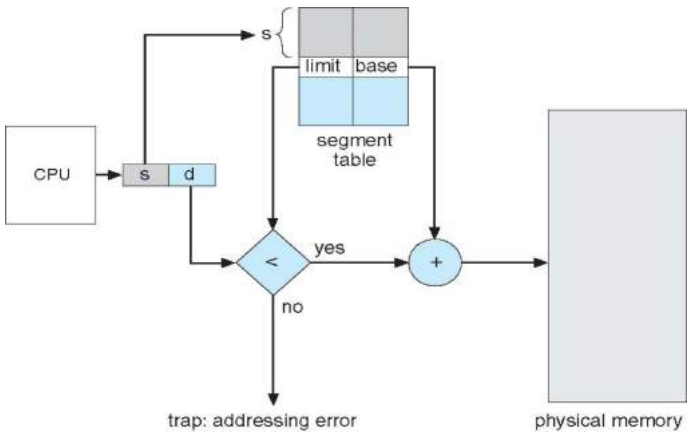

Figure 2.6. Segmentation hardware (source: [26]).

Virtual memory

All the memory management strategies described above efficiently manage processes in memory in order to raise the multiprogramming. As described in [26], they tend to require that an entire process be in memory before it can execute. In order to relax this constraint the concept of virtual memory is introduced. Virtual memory is a mechanism used for allowing the execution

of processes that are not completely loaded in memory. A program can be executed even if it is larger than physical memory. The virtual memory technique is strictly connected to the demand paging mechanism. With demand paging pages are only loaded when they are requested during program execution. If a page is not demanded, it will not be loaded in volatile physical memory. This method speeds the execution of the program start-up phase because the operating system does not load the whole set of the pages. Hardware support is needed to know which pages are in memory and which pages are on the disk in the swapping area. Each entry in the page table has a bit used for this purpose. If a page is accessed but the bit in the page table is set as invalid a page fault is raised causing a trap to the operating system. The operating system will load the page from the swap on a free frame in memory. Then it will update the page table and restart the instruction in order to access the page as shown in Figure 2.7.

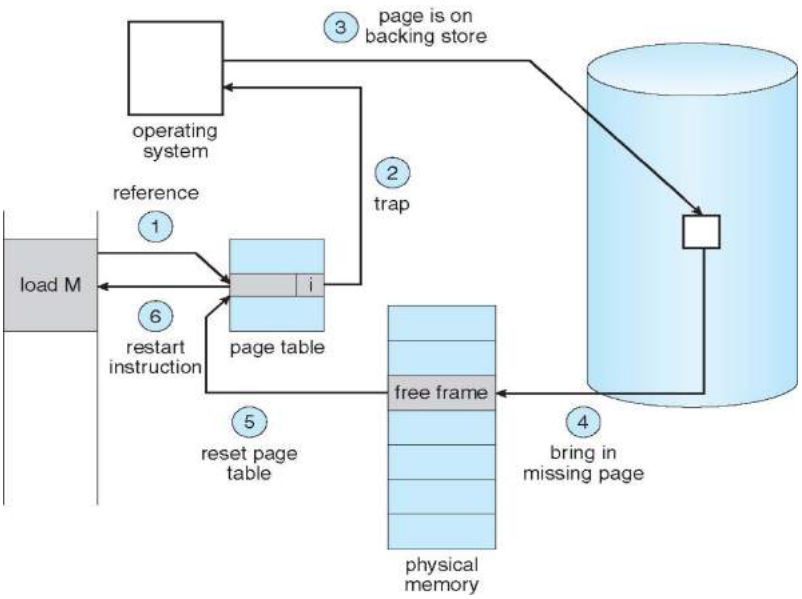

Figure 2.7. Page fault managing (source: [26]).

2.2.3 Memory acquisition overview

As reported in [27] memory acquisition means copying the contents of volatile memory to non-volatile storage. This is one of the most important and critical phase in the memory forensics process in order to obtain a non-corrupted image. All acquisition methods generate a sort of distortion to the digital environment. The analyst should pay attention to those distortions because they could impact the analysis. There are several methods for acquiring the content of volatile memory and each one is different from the other. The analyst before acquire the memory should study the case and the environment in order to understand what is the best method for that particular case. The Figure 2.8 shows a relatively simplistic decision tree based on some of the factors that the analyst should take in consideration. An important factor is whether the target system is a virtual machine (VM) or not, because if the target is a VM, the analyst may use options for acquiring memory provided by the hyper-visor such as pausing, suspending, taking a snapshot, or using introspection. Another important factor is whether the target is currently running. If the target is powered down the analyst cannot directly access the volatile memory but in many cases can find volatile memory files on the hard disk. These files include hibernation file, crash dumps and page files. If the target is running the examiner could dump the volatile memory but this kind of operation in many cases require administrator-level privileges. In this case several methods are available.Otherwise, it is needed to use hardware methods exploiting Direct Memory Access (DMA) using technologies such as Fireware, PCI and Thunderbolt.

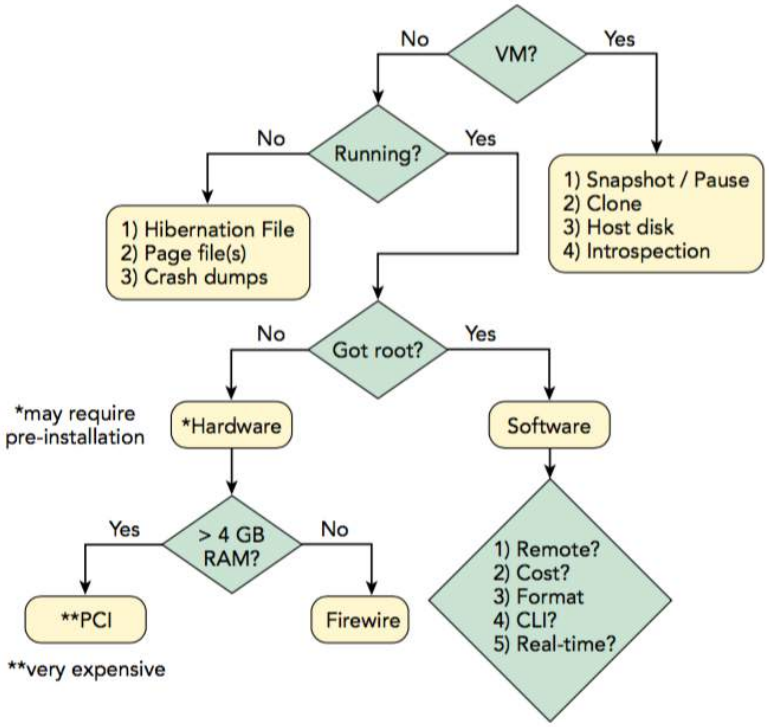

Figure 2.8. Acquisition decision tree. (source: [27]).

Acquisition critical points

Most operating systems do not provide a native mechanism for acquiring physical memory and this could leave the machine in an unpredictable state. Some operating systems provide software interfaces for the user or kernel level programs to read specific areas of memory. However, if the OS itself has already been compromised the OS cannot be trusted to provide the correct RAM contents. This is the reason why J. Wang et al. [28] create a PCI-based solution instead of a software-based solution. The following part describe some of the major reasons why memory acquisition tools can leave the system in an unstable state and why the memory dump could be corrupted. As reported in [27] memory acquisition is not an atomic operation. The acquisition is a process that take time and during the time the content of the RAM constantly change, even during an idle state. It is important to note that acquisition tools try to get the current state of the volatile memory. But the current state could be the time when the acquisition started, ended, or at any moment in between starting and ending. Sometimes could happen to have a corrupted memory dump that analysis tools cannot examine. Another critical point, as explained in [27] is the reading of the device-memory regions. On x86/x64 architectures, the firmware provides a memory mapping to the OS with some regions marked as reserved used by the firmware, by the ISA or PCI busses, or by various motherboard device, as shown in Figure 2.9. As reported in [29] by J. Stuüttgen and M. Cohen the MMU translates memory access from the virtual address space into the physical address space using page tables and the Control Register CR3. The CR3 value is the address for the page tables. The operating system configures hardware DMA buffers within reserved regions for direct access to device memory. Reading these regions can be seriously dangerous for the system stability. Reading DMA mapped regions from the physical address space does not result in a read operation from system memory, rather it activates various hardware devices connected to the system. These read operations may trigger interrupts, system crashes, or even data corruption if not performed according to the device's specific PCI bus protocols. For the memory acquisition tool, it is extremely important to ensure that DMA regions are avoided during acquisition in order to minimize the chance of unrecoverable system crashes. Some physical addresses are mapped to device registers that change the state of the device each time the physical location is read. That change could alter the state of the device sending the system in a freezing

state.

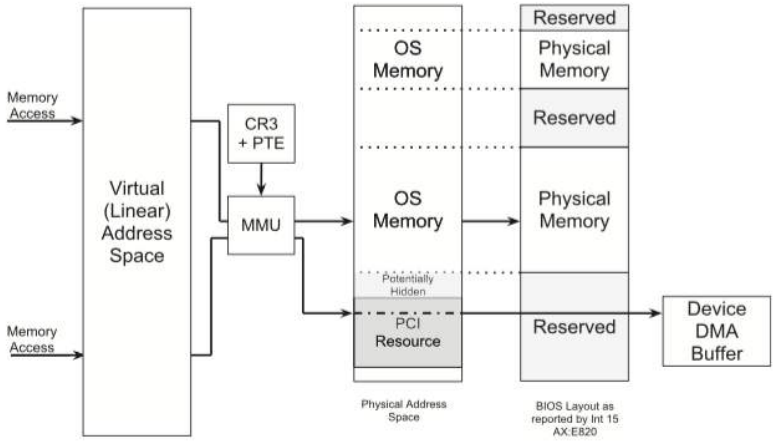

Figure 2.9. Physical Memory utilization on modern architectures (source: [29]).

2.2.4 Evaluation criteria

Acquiring the content of the volatile memory is one of the most important steps in memory forensics but remains an open problem what is the best practice for acquiring memory without disrupting the environment. B. Schatz describes how should be the ideal acquisition tool [30]. As described by B. Schatz in [30] the function of an ideal memory acquisition method is to sequentially access the volatile memory and copies out to an external storage. This copy of the memory should be a precise copy the original host's memory. The ideal memory acquisition method must work on arbitrary computers, and additionally must produce a reliable result, either producing a trustworthy result or none at all. A reported in [31] the definition of B. Schatz implies three different requirements. The copy must be exact, meaning there must not be any kind of error in the produced image. It must be complete, which means all physical memory ranges must be copied and it is not possible to obtain an image with fewer data with respect to RAM memory. And finally, the copy must be created at a specific point in time, which implies the image must be taken at once, not over a long period of time. To be able to measure the quality of a memory acquisition tool and thus the resulting memory image, several authors have proposed criteria for evaluate the quality of the output generated by a memory acquisition tool. In 1993 Afek et al. [32] focus on atomicity criteria and outline a solution for producing atomic snapshots of shared memory in a distributed system model. In 2007 Schatz [30] proposed three different criteria for measuring the quality of an image: fidelity, reliability and availability. Subsequently in 2011 Inoue et al. describe four metrics for estimating the quality of a physical memory snapshot, namely correctness, completeness, speed, and the amount of interference. The most recent criteria are described by Vömel and Freiling in 2012. In order to measure the quality of an acquisition procedure they describe three independent criteria in [33]: correctness, atomicity, and integrity. In the following section a short overview of these criteria is given, focusing in particular on the explanation of Vömel and Freiling's criteria, analysing the differences from criteria proposed by other authors. The purpose of this overview is to help the analyst to comprehend differences between different acquisition methods with respect to the produced output. This evaluation is useful for preferring and choosing an acquisition method instead of another one after a precise measure of quality parameters. Vömel and Freiling (2012) merged the many different notions of forensic soundness in the literature into three criteria for snapshots of volatile memory: correctness, atomicity and integrity. In [33] is reported a definition for memory snapshot, useful for completely comprehend the criteria.

We denote the set of all addressable memory regions by R, the set of all possible values of a given memory region by V, and the set of all timestamps by T.

The function m takes a specific memory region as well as a specific point in time and returns the contents of that memory region. So, for example, $m(x, y)$ refers to the value of memory region

x at time y.

$$m : R \times T \to V$$

A memory snapshot (or simply snapshot) is a vector of tuples that, for every memory region, contains the value of that region together with the point in time the value was retrieved from this region . A snapshot is formalized by the function:

$$s : R \to V \times T$$

from memory regions to tuples (x, y). For any such tuple we denote by $s(r).v$ the first component and by $s(r).t$ the second one. For example, if $s(r) = (x, y)$, then $s(r).v = x$ and $s(r).t = y$.

- Correctness: a memory snapshot is correct if the image contains exactly those values that were stored in memory at the time the snapshot was taken. The degree of correctness is the percentage of memory cells that have been acquired correctly.
- Atomicity: the criterion of atomicity leverage on the fact that the memory image should not be affected by signs of concurrent activity. A not atomic snapshot become unusable by analysis tools in the most of the cases. The degree of atomicity is the percentage of memory regions that satisfy consistency.
- Integrity: the integrity criterion is a measure of the impact of a tool on the memory regions. [34] A snapshot satisfies a high degree of integrity if the impact of a given acquisition approach on a volatile memory is low. By loading an acquisition tool into memory, specific parts of memory are affected and modified raising the degree of system contamination. The degree of contamination is inversely proportional to the integrity measure.

Correctness

Vömel et al. in [33] reported that the correctness means that the snapshot contains only "true" values, i.e., those values that were actually stored in memory when the snapshot was taken. Correctness applies to any memory region contained in the snapshot, i.e., the concept can be applied to both full and partial snapshots. More specifically, *a snapshot is correct with respect to a set of memory regions $R \supseteq R$ for all these regions, the value that is captured in the snapshot matches the value that is stored in this region at this specific point of time.* Formally, a snapshot s is correct if it satisfies the following condition:

$$\forall r \in R : s(r).v = m(r, s(r).t)$$

As reported in [33] a clear example of a code that produce a correct output could be the same in the listing 2.10 where the memory is modelled as an array m of dimension n. The lines show an algorithm that produces a correct full memory snapshot. It cycles through the RAM and copies the values of those regions to the snapshot array. Respectively, in the second part of the Figure 2.10 is possible to see a code that produce an incorrect output because every second address in the output is not the same of the RAM content. Satisfying correctness may seem easy at first glance. However, creating a correct copy of RAM memory becomes significantly more difficult in the light of a possibly subverted operating system. As reported in [31] is possible to develop a malicious software that is able to manipulate the buffer of the image that is ready to be written as the RAM output. In this way the malware causes the acquisition tool to write an incorrect image. Furthermore, there are at least two other factors that can make difficult the acquisition process:

- Errors in the acquisition software can result in memory regions being copied to wrong parts of the image. For example as reported in [31] a popular open source software had a bug that stored at the wrong offset the content of memory in the raw image. As a result, the address translation become impossible.
- A memory enumeration procedure that produce a wrong memory image. Some tools produce memory images with different pages number because they do not enumerate correctly the memory.

```
#Correct memory acquisition
for i = 1 to n {
       s[i] = m[i]
}


#Incorrect memory acquisition
for i = 1 to n {
       if i mod 2 == 0
              s[i] = m[i]
       else
              s[i] = 0
}
```

Figure 2.10. Example correctness.

Atomicity

Software-based memory acquisition tools do not copy memory in an atomic operation. Sometimes could happen that some memory regions are modified before being copied for the resulting image. The atomicity of a memory image is quantified by the amount of memory values that are changed by concurrent system activity during the time of the copying operation. In [33] is reported a formal definition based on the theory of concurrent systems. *A snapshot is atomic with respect to R if the corresponding cut is consistent. Likewise, a snapshot is atomic with respect to a subset of memory regions $r \in R$ if the corresponding cut through the partial space-time diagram is consistent.* In the first example in the listing 2.10 the algorithm does not take into account any precautions to work with concurrent system activity and in most of the cases produce non-atomic snapshots whether the environment is concurrent. As reported by Freiling et al. it is possible to modify the acquisition code reported in 2.10 in order to work properly in a distributed environment. As shown in 2.13 is possible to use Lock primitives in order to prevent a non atomic snapshot. If lock is active, the atomicity of the respective operations can be ensured. Note that the second algorithm, although is atomic, is still incorrect. As reported by Vömel and Freiling this shows that correctness and atomicity are two independent properties of snapshots. Vömel and Freiling formalised the definition of atomicity by reverting to the theory of distributed systems where concurrent activities are depicted using space-time diagrams. As reported in [31] an atomicity violation happens when the contents of a memory region are modified by concurrent system activity before it can be acquired, but then cause of the modification is not present in the image. An atomic snapshot should not show any signs of concurrent system activity.

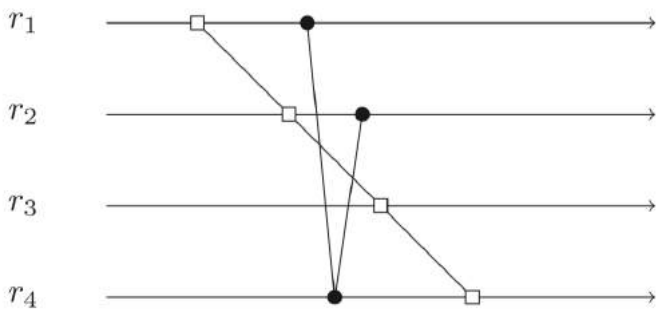

Figure 2.11. Atomicity violation (source: [34]).

As reported in [34], a general example is given in order to show an atomicity violation. In the Figure 2.11 it possible to see an acquisition procedure that runs in parallel to another activity

on a machine using four memory regions $R = \{r1, r2, r3, r4\}$. Each horizontal line represents the evolution of each memory region over time. The dots on the line are state changes. The imaging procedure is shown as four events, marked as squares, that read out each memory region sequentially. Concurrently there is a secondary process that modifies sequentially the regions $r1$ then $r4$ and finally $r2$. The cut distinguishes a "past", before the snapshot, from a "future", after the snapshot. Intuitively, as reported in [34] a cut is consistent if there are no activities from the future that influence the past. Since the cut is never vertical because the software acquisition tools spend time in order to access every memory region, the consistency is given by the fact that there are no activities from the future that influence the past. Given this intuition, it is clear that the snapshot created in Figure 2.11 is not atomic. A more detailed example is reported in [31] based of Figure 2.12. In this example the resulting image is correct but the process analysis will result incorrect because ofa lack of atomicities. The figure shows a set of processes $p \in P$ operating on memory regions $r \in R$ at times $t \in T$. The process $p2$, the memory acquisition software, starts to copy the memory content at time $t0$. $p2$ dumps the page table of the process $p1$ stored in $r1$. Simultaneously, the process $p1$ writes some data in $r3$ and releases the memory region. After that the process $p3$ allocates the same memory region and stores some data in $r3$ at time $t4$. The acquisition tools copies the data in $r3$ at time $t5$. The analysis is also performed correctly, but the software will interpret the region $r3$ as a region of $p1$ because in the dump there is the page table of $p1$. The result of the analysis is wrong, because the image is not atomic.

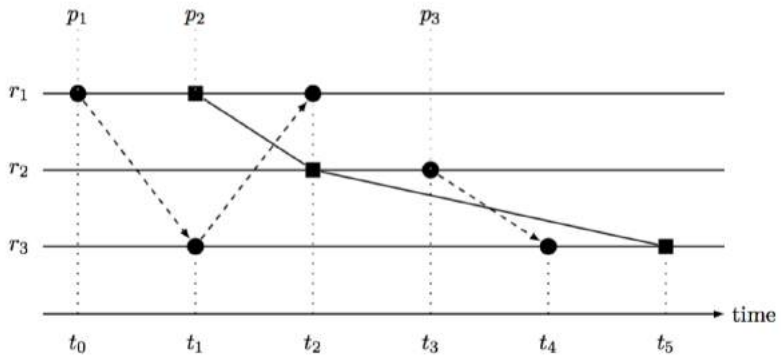

Figure 2.12. Atomicity violation (source: [31]).

As reported in [31], state-of-the-art memory analysis tools are unable to identify atomicity violations in a memory image. They assume to receive an atomic image as input image. However, if they do not receive an image with this property they could not work properly.

Integrity

It is possible to define the level of integrity of an image as the amount of memory that changes during the acquisition process. This property is measured relatively to a point in time. As reported by Freiling et al. in [33], the third property of memory snapshots ties a snapshot to a specific point of time chosen by the investigator. Intuitively, this is the time where the acquisition started, but the definition allows for any arbitrary point in time. A more formal definition is reported in [33] and is reported below for completeness.

Let $r \subseteq R$ be a set of memory regions and $\tau \in T$ be a point in time. A snapshot s satisfies integrity with respect to R and τ if the values of the respective memory regions that are retrieved and written out by an acquisition algorithm have not been modified after τ.

$$\forall r \in R : \ \tau \leq s(r).t \rightarrow \exists t \in T : \ t \leq \tau \wedge \forall t' \in T : t \leq t' \leq s(r).t : \ s(r).v = m(r, t')$$

In a certain sense, as reported in [34] the integrity refers to the "stability" of a memory region's value over a certain time period. An intuitive example is reported by Grhun et al. based on Figure

```
#Atomic and correct memory acquisition
Lock
for i = 1 to n {
       s[i] = m[i]
}
Unlock


#Atomic but incorrect memory acquisition
Lock
for i = 1 to n {
       if i mod 2 == 0
              s[i] = m[i]
       else
              s[i] = 0
}
Unlock
```

Figure 2.13. Example of an atomic acquisition algorithm. (source: [33])

2.14 where the memory is composed by four memory regions. $R = \{r1, r2, r3, r4\}$. At time t, the imaging operation by the acquisition tool starts and leads to a change in the memory regions $r3$ and $r4$, as indicated by the black dots because the tool needs to be in memory and the operating system allocates region for it. Moreover, libraries have to be loaded. All these operations cause memory to be overwritten after the imaging process has started. The snapshot events are represented by black squares. Referring to t, the snapshot satisfies integrity for memory regions $r1$ and $r2$ but not for $r3$ and $r4$. At time t the investigator decides to take an image of a computer's memory. As suggested by Freiling in [34], the time t should mark a time very early in the investigation process. Vömel and Freiling show that under certain assumptions the integrity of a snapshot implies its correctness and its atomicity.

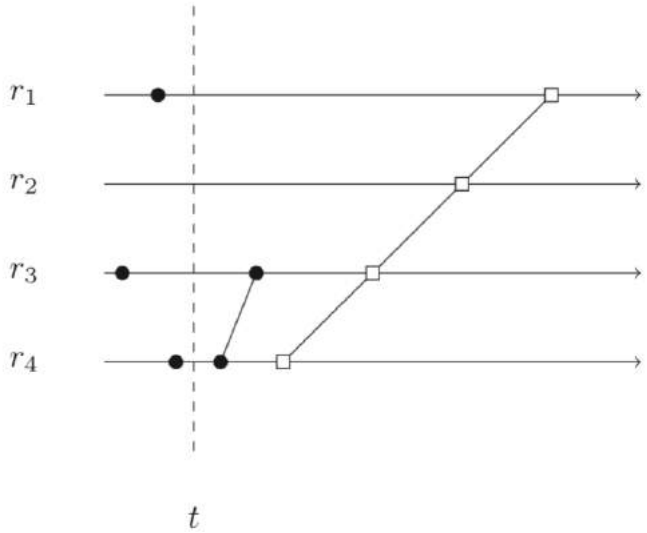

Figure 2.14. Integrity violation (source: [34]).

Forensic soundness

In the previous paragraphs are reported the three criteria defined by Vömel and Freiling identified as correctness, atomicity, and integrity. It is important to note that all three criteria are independent and cannot be reduced to each other. On this basis all three criteria have to be satisfied for creating a forensic copy of physical memory. As reported in [33] this leads to the following rule: *The quality of a forensic snapshot is determined by its degree of correctness, atomicity, and integrity.* Several debates within the forensic community over the last years have been made upon the meaning of "forensic soundness". As a conclusion Vömel and Freiling considered these debates in their definition of criteria. A memory image may be regarded as "forensically-sound" although parts of the image do not. According to their definitions, they neither specify how big the memory (or small) R should be, nor what areas of memory R should comprise. These aspects are intentionally left to be defined by the investigator in order to determine the degree of forensic soundness. A good acquisition method will seek to satisfy the different evaluation factors with respect to a set of memory regions R that maximizes both information quantity and quality. In this case it is intended quality a set of regions R that contains information that are "essential" to the investigation.

2.2.5 Memory acquisition methods

The purpose of this section is to categorise different acquisition methods in order to help the analyst to choose the best technique for the specific case. The categorisation is based on type of acquisition. Different approaches are possible in order to gather the content of the volatile-memory. The differences are mostly based on the degree of atomicity, reliability and availability (i.e., solutions must be device-agnostic). The main acquisition tools can be divided in three categories:

- Software-based tools: this kind of tools are based on loading a kernel module that maps the physical address space into the virtual address space of a particular process.
- Hardware-based tools: this kind of tools are based on the use of the Direct Memory Access (DMA) through the PCI bus.
- SMM-based tools: this kind of techniques exploits the System Management Mode of the x86 Intel CPU.

Software-based acquisition

Software-based approaches vary greatly both in complexity and reliability. All software-based acquisition tools follow a similar protocol to acquire memory. Because current operating systems operate in protected mode for security and safety reasons, acquisition of the entire physical address space can only be achieved in system mode. This can be done by loading a kernel module that maps the desired physical addresses into the virtual address space of a task running on the system. At this point, it is possible to access the whole data set from the virtual address space and then copy it on a non-volatile storage. As reported in [35] since the Linux kernel checks modules for having the correct version and checksums before loading, the kernel will typically refuse to load a kernel module pre-compiled on a different kernel version or configuration to the one being acquired. This check is necessary since the struct layout of internal kernel data structures varies between versions and configurations, and loading an incompatible kernel version will result in kernel instability and a potential crash. Acquisition software tools can follow two different paradigms in order to map physical address space into the virtual address space:

- Using an operating system API to create a page table entry. The typical functions used are: `ZwMapViewOfSection()`, `MmMapIoSpace()`, `MmMapLockedPagesSpecifyCache()` and `MmMapMemoryDumpMdl(`
- Allocating an empty page table entry and manually mapping the desired physical page into the page table entry.

As reported in [29], the acquisition tool has two problems to solve. The first is the enumeration of address space layout: the tool has to determine which parts of the physical address space are backed by physical memory as opposed to peripheral device DMA buffers. DMA regions must be avoided to prevent system crashes. The second problem is the physical memory mapping: Since the tool must access physical memory via the virtual address space, the tool must create a page table mapping between a region in the physical address space and the virtual address space.

Another significant difference is that not every tools identify the physical address space to include in the dump in the same way. Some tools start to dump from the physical address `0x000000` until they reach the end of the physical memory. This fact is crucial for the device memory regions. Almost all tools tend to skip those regions for the causes explained in the section above. As reported in [27] this precaution makes the process more stable, but sometimes could be possible to miss important evidence of sophisticated rootkits. On the other hand, some applications provide options for acquiring all physical addresses from page 0 to the perceived end of the physical address space. This is more risky but has the potential to produce a more complete representation of the memory. Ideally, the best tool should know how to acquire relevant data from device-memory regions avoiding to crash the system. For example, if the acquisition tool can identify the device that occupies the particular physical memory region, the tool can use a method appropriate for that particular device in order to obtain data in those regions. Hence, as reported in [29] reading some memory regions such as DMA mapped regions from the physical address space does not result in a read operation from system memory, rather it activates various hardware devices connected to the system. These read operations may trigger interrupts, system crashes, or even data corruption if not performed according to the device's specific protocols. For the memory acquisition tool, it is extremely important to ensure that DMA regions are avoided during acquisition in order to minimize the chance of unrecoverable system crashes.

The simplest way to dump the memory via software is to rely on special virtual devices, if present, like /dev/mem on Linux or\Device\PhysicalMemory on different Windows systems. Such devices, in fact, allow user space programs to read the whole physical memory of the running system. An alternate method is to use the `NTSystem-DebugControl` API call but newer versions of Windows prevent user mode access to this device, necessitating accessing physical memory from a custom device driver. This possibility leads to use common command line utilities, for example on Unix systems is possible to use dd or netcat command. First and foremost, as this approach is run as user level code on the target machine, other processes and the operating system continue to run while the acquisition process takes place. Memory will be continually changed by the continued running of other software, leading to the image not being of a particular point in time, rather, the image is a time lapse shot. As reported by Schatz in [30] this kind of technique is not robust, with a number of observed problems related to page caching. The main drawback of all these solutions is that they need to be loaded into memory in order to be executed, modifying the state of the target system. This causes reducing the level of integrity criteria presented by Vömel and Freiling. Another problem reported by Schatz is that all these techniques are subject to kernel rootkit subversion because they depend on OS target machine.

Schatz et al. in [30] proposes a new method called BodySnatcher in order to go beyond common problems presented above. B. Schatz proposes a method for acquiring the contents of RAM memory from arbitrary operating systems in a manner that provides atomic snapshot of the host OS volatile memory. As reported by B. Schatz, the method is more resistant to subversion due to its reduced attack surface. The method is based on the injection of an independent acquisition specific OS into the potentially subverted host OS kernel, snatching full control of the host hardware. In particular, the BodySnatcher method insert a minimalist OS into the kernel space of the potentially infected kernel, precisely saves the state of the running OS, then boots the acquisition OS in a restricted subset of the machine memory. The acquisition OS then employs a small and common subset of the host's hardware as an output channel for dumping an image of the physical memory of the host. The method has been demonstrated on Windows 2000 hosts using a custom version of the Linux kernel as the acquisition OS. The method holds the potential of offering the same availability as existing software-based approaches, while increasing the fidelity of the produced dump and reliability of the acquisition process. Main steps performed by the acquisition OS are the following:

- The host OS is halted, so that it is in a deterministic state.
- The host OS memory pages are quarantined in order to preserve the memory pages of the host OS.
- An output device (from a minimal set) is identified and initialized.
- A copy of the physical memory is written to the output device.

As reported by Schatz et al. this method assures that an image is atomic with respect to the host OS, by nature of the pausing of the host OS in a deterministic state. By utilizing only the memory handed it by the host OS, the acquisition OS ensures host OS memory integrity but has problem of impact on the integrity of the unallocated memory of the system. The approach has the potential to be more reliable by nature of having a smaller attack surface than other host OS dependent software based approaches. This approach has many limitations that are strictly linked to the low level nature of the approach. Despite of it requires an entire new OS in order to operate the host hardware, the approach still remains vulnerable to subversion via a number of vectors. Firstly, hardware virtualisation features of the newer generation of CPUs might be employed by a rootkit to prevent the acquisition OS from obtain the full control of the host OS hardware. Secondly, the necessity of relying on the host OS for loading the acquisition OS means that at this point in the life cycle of the acquisition method, the process is vulnerable to attack.

As said at the beginning of this section, all software-based methods leverage on a Loadable Kernel Module. Since Linux kernel checks the correctness of the module it is not possible to have a generic module in order to acquire memory. M. Cohen et al. in 2014, as reported in [35], proposes a method in order to go beyond this limit. In case of incident response this kind of problem makes memory acquisition problematic, since often the analyst does not know in advance which kernel version they will need to acquire. It is not always possible to compile the kernel module on the acquired system, which may not even have compilers or kernel headers installed. M. Cohen in [35] proposes an innovative technique to safely load a pre-compiled kernel module for acquisition on a wide range of Linux kernel versions and configuration. In particular, this technique in based on the injection of a minimal acquisition module called Parasite into another valid kernel module (host) already found on the target system. The resulting module is then re linked in such a way as to grant code execution. On the other hand, the host module remains dormant during runtime. Most modern kernels have a large number of legitimate kernel modules, compiled specifically for the running kernel, already present on the system. The approach locates a suitable existing kernel module, injects a new kernel module into it and loads the resulting module. The combined modified kernel module is fully compatible with the running kernel. Cohen et al. developed a physical memory acquisition kernel module and an LKM infection engine that, once built in the environment, can be loaded on any Linux kernel between 2.6.38 and 3.10, as reported in [35]. However, this technique is not perfect and has a particular problem. For sophisticated analysis of the acquired memory dump it is needed to gather information on data structures. Some well known analysis tools refer to this kind of information as a profile. His profile is built by compiling a module with DWARF information for the exact kernel version on the target, which is then parsed to extract struct layout and symbol information by the analysis tool. When the kernel version and configuration is not known this kind of approach is not possible.

As reported by Schatz in [30] the final current option for capturing memory contents by software is to use the hibernation functionality present in most modern operating systems. Windows for example hibernates by saving a subset of state of the system, including the contents of RAM to a hibernation file on the primary OS filesystem. Such facilities depend on hardware and BIOS support, and are typically found most widely deployed on laptops, server hardware, however, does not typically support this facility.

Other software memory solutions can be used when dealing with virtual machine. The execution of the virtualized system can be completely stopped by the hypervisor offering a point in time atomic snapshot of the target machine. In this research, in order to rapidly prototype the reported techniques, several methods have been used for acquiring the RAM content. In particular, mainly for convenience, it has been used software-based acquisition tools. In this part of the paragraph are reported two example for obtaining the content of the volatile memory using a software-based

technique. The first example uses utilities offered by the hypervisor VirtualBox. The software-based acquisition is possible following these three steps:

- Open the command line and type `VirtualBox --dbg --startvm [virtual_machine_name]`.
- In the VirtualBox menu click on `Debug` and then `Command Line`.
- In the VirtualBox Command Line type `.pgmphystofile [filename]` to save the physical memory to the given file.

When the command is launched the hypervisor freeze the virtual machine in order to maintain the machine in a deterministic state, offering the atomicity criteria. As shown in the Figure 2.15 if the operation is successful ended, a message is shown.

```
#VIRTUALBOX CONSOLE -- VBOXDbg Console
$ Welcome to the VirtualBox Debugger
$ Current VM is 0b9988000, CPU#0
VBoxDbg> .ppgmphystofile ramdump
$ Successfully saved physical memory to 'ramdump'
VBoxDbg>
```

Figure 2.15. VirtualBox output.

Another example uses the tool `LiME` (Linux Memory Extractor)[36], a Loadable Kernel Module (LKM) which allows acquiring the content of the volatile memory from Linux and Linux-based devices. As reported on [36] it minimizes its interaction between user and kernel space processes during acquisition, which allows it to produce memory captures that are more forensically sound than those of other tools designed for Linux memory acquisition. After that `LiME` has been compiled[36], in order to gather the content of the RAM it is needed the type the command `insmod lime.ko "path=[path_dump] format=lime"`.

Hardware-based acquisition

It is difficult to perform a reliable acquisition when the target system is compromised. The analyst should take in consideration the fact that the system could be under attack. The acquisition procedure should not rely on untrusted code, such as the operating system or every applications that are running on it. As reported in [12], Rootkits and Trojan horse attacks against applications and operating system kernels and they can cause the system to produce unreliable data, so it is desirable not to rely on the resources that the attacker could have modified or substituted. Existing methods for acquiring volatile memory involve untrusted software and are invasive because they typically write back to memory hiding important evidences. A different approach for acquiring the content of the volatile memory is the hardware-based technique. This kind of method is based on memory accessing through the Direct Memory Access (DMA) using a technology such as Firewire, Thunderbolt, ExpressCard or PCI. In this case, the analyst does not need credentials of the target system in order to access the memory.

One of the main aspects of these solutions is that they do not cause any change in the state of the running OS. Another important characteristic of this solution is that is not affected by most of attacks-hiding techniques, as reported in [27]. In [12] Grand et al. present a procedure for acquiring volatile memory using a hardware expansion card that can copy memory to an external storage device. The card is installed into a PCI bus slot before an incident occurs and remain disabled until a physical switch on the back of the system is pressed. The card cannot easily be detected by an attacker and the acquisition procedure does not rely on untrusted resources.

The solution reported by Grand et al. involves the installation of a PCI expansion card into a computer before an intrusion occurs. The PCI card is used to dump the exact contents of volatile

physical memory to an external, non-volatile storage. As reported in [12] the PCI controller on the card is disabled until the card is activated by the incident response team and therefore the card will not respond to bus queries from the host system. The card will remain hidden to the PCI bus until the device is disabled. Because of the design of PCI, which is software independent, the attacker may see the device but may not be able to tamper with its actions.

When the machine powers up, the acquisition card configures itself, then disables its PCI controller so that it does not respond to bus queries. Only when the device is actually enabled by the incident response team it becomes visible to the PCI bus. The back of the PCI card has a physical switch and an interface to an external storage. When the switch is turned on, the PCI controller on the card is activated and takes control of the bus. The card first suspends the CPU and then uses DMA to copy the contents of physical memory to an external non-volatile storage device. After that, the CPU is resumed and the operating system continues to run again.

A short overview of the architecture needed for this kind of technique is reported below for completeness. Grand et al. in [12] reported that the primary components of the acquisition card are a microprocessor and a PCI controller with bus master capability and support for DMA. The microprocessor is connected to ROM, which contain the operating firmware for the acquisition. Then it is needed a SDRAM, an external momentary switch, a LED, and an external storage interface, as shown in Figure 2.16.

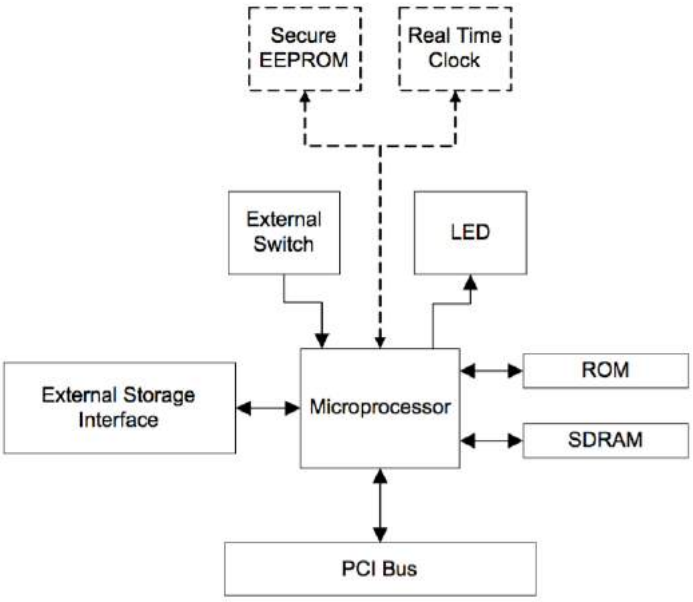

Figure 2.16. PCI-based technique block diagram. (source: [12]).

In the most of the modern architecture the North Bridge connects the host processor bus, where the CPU and physical memory are attached, to the PCI bus. A more general architecture is reported in Figure 2.17. DMA is used to provide block transfers of data between the PCI bus and memory without requiring resources from the target processor itself. Using DMA allows a peripheral device, such as the acquisition card, to transfer data directly from memory without interrupting the execution of the processor. During the acquisition process, the card take control of the bus and request a DMA transfer of system memory by specifying the desired base address and size of the block. The combination of PCI bus capabilities and DMA transfers is crucial to the effectiveness and reliability of this technique.

As reported in [27] the downside to this method is that unless the target machine is already equipped with a device, or if it supports hot swapping devices, it is needed to turn off the device in order to install the required hardware adaptors. As reported in [37], this fact greatly reduces their usability. The problem is that turning off the target system, it will destroy the volatile memory content losing important evidences. Other disadvantages include the fact that Firewire only permits acquisition of the first 4GB of RAM, which can severely limit evidences in such a case

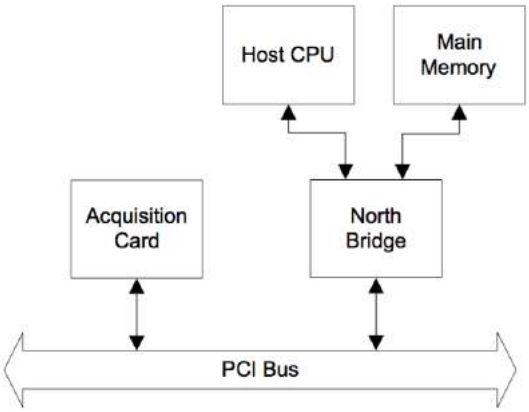

Figure 2.17. Layout of the PCI bus. (source: [12]).

the memory is larger than 4GB. Additionally, PCI devices for memory acquisition are extremely rare and also quite expensive. As reported by Grand et al. in [12], the thought of installing a PCI card into multiple systems before an incident occurs may sound expensive and impractical. However, the intent of the card is not to be installed into every computer system in a particular environment. This device is most useful when installed into at-risk, critical servers where an attack is likely and a high-stake intrusion investigation might occur. FireWire-based solutions solve this issue by allowing analysts to hot-plug them in the target system. These techniques suffer of three main problems. Firstly, they cannot access the processor state (i.e., registers). Secondly, a malware can perform a scan of the PCI bus and detect the presence of such ad-hoc devices and consequently decide to stop any malicious activity and wipe every fingerprinting left behind. Lastly, devices can be tricked to provide split views of memory contents, thus making the output of devices unreliable.

SMM-based acquisition

As reported in [28], software solutions for memory acquisition are not reliable because Rootkits can hide processes running on the target machine. Due to this limitation, researchers have turned their attention to hardware-based method using a customized PCI card that can be used to read the physical memory via Direct Memory Access (DMA), as reported in the previously section. Unfortunately, as said by Wang et al., using specialized PCI cards to access the volatile state has still two unresolved challenges. First, PCI cards can be blocked from access to all the physical RAM memory. Both Intel and AMD provide technologies to restrict the memory access for peripheral devices for security reasons. Second, all the methods presented in the previously section do not access CPU register values that are fundamental for a correct interpretation of the obtained dump. As reported in [28] by Wang et al. such values are, for instance, the interrupt descriptor table register (IDTR) that points to the current interrupt descriptor table (IDT). Without knowing the value of the IDTR register, an analyst has to apply heuristic methods to search for and identify the value of IDTR. But these heuristics are considered unreliable and cannot be used to do a robust forensic analysis. Another important value is the control register 3 (CR3), which points to the base address of the current page table. This kind of information is necessary for the analysis phase and without it the analysis will result impossible.

In the last years some researchers proposed solutions which exploit the characteristics of System Management Mode. Wang et al. in 2010 presented a mechanism for RAM acquisition that leverage the SMM. This approach uses a PCI dedicated network card that have to be installed on the target system. In [37] A. Reina et al. propose a different method, as based on SMM as Wang's method but it does not require a dedicated PCI network card. It leverage the one already installed on the machine and not a custom piece. Before explaining the methods invented by Wang et al. and A. Reina et al. it is needed to report what is the SMM and why it is important in those acquisition methods. The System Management Mode (SMM) is a special mode of operation of Intel CPUs that is different from the traditional protected and real-address mode. As reported in

[28] it provides a mechanism for implementing platform specific system-control functions, such as power management and system security functions. System Management Mode (SMM) was first introduced in the Intel386 SL and Intel486 SL processors, as reported in [28]. It became a standard IA-32 feature in the Pentium processor. The motherboard controller is programmed to handle many types of events and time-outs used to trigger SMM mode. As reported in [37] the processors enters SMM in response to a System Management Interrupt (SMI), which has a higher priority compared with other interrupts, and it is signalled through the SMI pin on the processor or through the APIC bus. When SMM is launched, the CPU saves the current state of the processor. After that, the address space changes to the System Management RAM (SMRAM) address space. After the microprocessor's state has been stored to memory, the special SMM handler begins to execute. SMM code is not intended for general purpose applications, but it is limited to system firmware only. All microprocessor context of the currently running code is saved in SMRAM memory. The SMRAM can be made inaccessible from other CPU operating modes. That is the reason why it can be considered as a trusted storage. During boot time, the firmware BIOS initialize the SMRAM, copy the SMI handlers to it, and, as a form of protection, lock SMRAM to disallow any further writing accesses to this area. In fact, SMM is the mode of operation with the greatest level of privilege, informally named ring-2. The only way to exit from the SMM operation mode is using the RSM instruction that is available only in the SMM. The RSM instruction restores the saved state of the processor, and returns the control to the interrupted program. After that, the previous program resumes its execution from where it was interrupted.

The method invented by Wang et al. in [28] is implemented in the trusted SMM code, called SMRAM. As said in the previous paragraph, this secure memory region remains locked from all the non-SMM CPU modes and thus, it can be safely used for examining the system without any risks of being modified by a malicious user or program. The Wang's method is a firmware assisted method that leverage commercial PCI network cards and the x86 implementation of the SMM to reliably replicate the physical memory and the CPU registers using the support of a hardware module. The Wang's technique permits both off-line and on-line investigation. The overall architecture of the system reported in [28] is depicted in Figure 2.18.

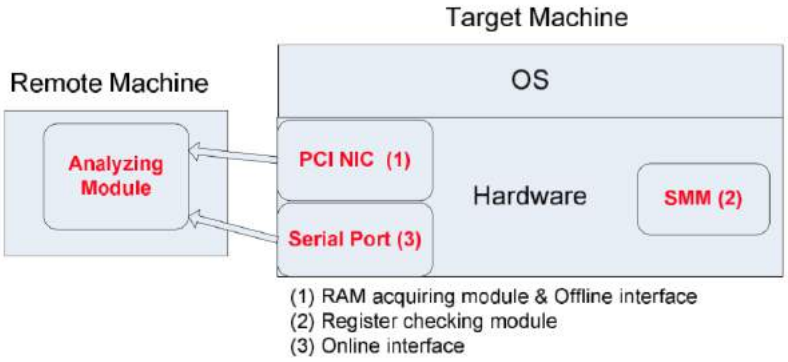

Figure 2.18. Architecture proposed by Wang et al. (source: [28]).

The architecture is composed by a computer that is used for the analysis and another machine that is the target system to be inspected. Target machine includes a dedicated network card and a serial port. These two machines are connected by a network cable and a serial cable. The network cable is used for off-line investigation, the serial connection for on-line investigation.

Another SMM-based method is presented by A. Reina et al. in [37]. This method is called SMMDumper, a technique to perform atomic acquisitions of volatile memory of running systems. SMMDumper is implemented as a x86 firmware and, as reported in [37], is resilient to malware attacks. SMMDumper can acquire more than the SMM-based methods where the barrier was 4GB. This point is crucial for the analyst because nowadays modern computer have usually more than 4GB of RAM memory and losing data could be fatal for the analysis process performed by the incident team. As said in [37] experimental results show that the time SMMDumper requires acquiring and transfer 6GB of physical memory of a running system is reasonable for

forensic analyses and incident responses. The A. Reina et al.'s solution is based on a *collector* and a *triggering* modules. The *collector* is resident in SMRAM and is responsible for transmitting the CPU state and the volatile memory content. On the other hand, the triggering module is responsible for activating the collector via SMIs. A bulletproof solution for the triggering module should rely on hardware-based action mechanisms, preventing software-based attacks. As reported in [37] a common scenario uses a specific keystroke directly connected to the SMI CPU pin. This methodology would isolate the software component, making it inaccessible from user- or kernel-space. However, the method presented in [37] uses a software-only implementation of the triggering module, for simplicity. A software-based solution could be realized modifying the local vector table (LVT) of the APIC controller to trigger an SMI upon the pressure of an appropriate keystroke combination [37]. Once the SMI is triggered, the CPU switches to SMM, the current system state is saved and the SMI handler that is the *collector* module is executed. In order to attack the software-based SMI trigger the malware could tamper the SMI triggering solution and subsequently prevent the launch of the dump of the system memory. The malware modifies I/O APIC, disabling the ability of a user to trigger an SMI by pressing a specific keystroke that launches the memory dump.

A general architecture of SMMDumper is shown in Figure 2.19.

1. The analyst invokes SMMDumper using a special keystroke sequence. The triggering module intercepts the sequence and switches the system CPU to SMM.
2. The collector module starts to dump the target volatile memory.
3. The collector module transmits the memory dump to a trusted host over the network.
4. The cryptographic module provide strong integrity of the transmitted data.

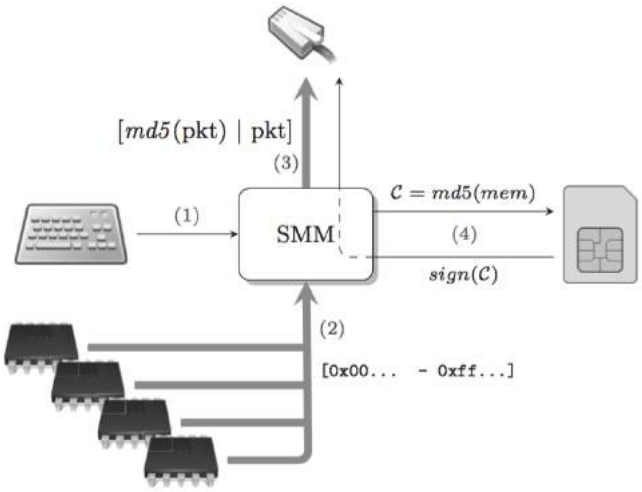

Figure 2.19. Architecture proposed by A. Reina et al. (source: [37]).

Remote acquisition

An important factor that the analyst have to evaluate is the physical access to the target system. As reported in [27] a case can quickly become complicated if the computer is located in another state or country. Moreover, the system target could be a server with no keyboard or monitor attached. In these situations, remote acquisition over the network might be the only available option during the acquisition phase. The point is that not all acquisition methods support the data transferring over the network. SMM-based techniques do not have all functions offered by the operation system, the same for hardware-based techniques. Some software-based technique send the executable file to the target system scheduling a task or install a service on the target system that runs the tool and sends the contents of physical memory back to the investigator via

a netcat listener or other protocol. The main problem of such technique is the exposure of the RAM contents over the network because the content is sent as plain text. Main computer memory contains a lot of sensitive information that might be disclosed when acquired in plain text over an open network. Another problem is the exposure of the administrator credential over the network. Ligh et al. propose a solution in which they suggest creating a temporary admin account that only permits access to the target system before the acquisition. After that, the account will be deleted. This prevents any malware or attackers from using the stolen credentials accessing the system subsequently. The market offers some tools that natively support end-to-end encryption using SSL/TLS for the transmission of the volatile content through the network. Well-known tools are CryptCat, KnTDD or F-Response Enterprise. Other factors that the investigator should take in consideration are the compression factor of the dump that have to be sent over the network and the integrity of the dump. A solution to data loss and integrity violation for SMM-based acquisition mechanism has been proposed by A. Reina et al. in the SMMDumper tool [37]. SMMDumper divides the physical memory into chunks of fixed size of 1 KB and each chunk is then embedded in a packet structured. The base address of the memory chunk contained in the packet is meta-data and represents a unique label that is used by the receiver to correctly handle out-of-order or missing chunks. Checksum over the whole packet payload it is used to offer the opportunity to detect integrity violations.

The checksum-based solution presented by Reina et al. protects only against network transmission errors, similar to what a TCP segment checksum does, but it fails against an attacker that modifies data and recomputes the checksum to reflect such changes. To address this threat, SMMDumper computes an incremental checksum C of the whole physical memory. Subsequently, when the transfer of the whole physical memory is completed, SMMDumper signs C and sends the resulting ciphertext blob to the receiver, which verifies the signature and compares C against a freshly computed checksum over all the received packets. A valid signature guarantees the integrity of the received signed checksum and matching checksums guarantee the integrity of all the received packets. The private key is stored into an external pluggable smart card device D. When SMMDumper is started it waits for D to be plugged into the system. Not only this procedure guarantees the integrity of the collected memory, but it also provides validity as the signing key can only be accessed by whoever has access to D.

SMMDumper implements a basic network driver that is able to communicate through I/O operations with the Network Interface Card of the target machine. As noted elsewhere, operating in SMM does not allow relying on any operating system-provided service, such as networking. Therefore, SMMDumper is also equipped with the code responsible to forge UDP packets.

Cold boot attack

In the previous sections several methods are described for acquiring the content of the volatile memory. It has been categorized those methods depending on what the method is based. Some methods leverage on software mechanisms, other methods are based on the dialogue with the DMA and, in the last section, acquisition methods that exploit the System Management Mode are presented. In this paragraph is presented a well-known attack called Cold Boot Attack that can be exploited in order to get the content of the RAM. If the target system owner is a criminal, he could turn off the machine in such cases police officers are not so fast to gather evidences. In those particular cases this method is the unique method that can provide a chance for obtaining useful evidences. The Cold Boot Attack exploits the remanence effect of modern RAM technology. As reported in [38] modern RAM technology is commonly based on dynamic random access memory (DRAM), a type of RAM in which data is stored in chip made up of an array of capacitors. Each capacitor is charged or discharged, depending on whether the bit is set to 1 or to 0. As reported in [39] each capacitor voltage is compared to the load of a reference cell which stores a voltage half way between fully charged and fully empty. If the voltage of a cell is higher than the reference voltage the cell stores a one-bit, otherwise it stores a zero-bit.

As reported by Bauer in [38] since capacitors have a leakage current, their data content slowly dissipates over time and each cell has to be periodically refreshed in order to avoid losing the data stored in. This is achieved by reading the contents and writing it back to the RAM chip. The

time that DRAM will keep its contents without leakage affecting the content is called retention time. The fact that the retention time is non-zero and that memory will keep its contents for a while even when it is not actively refreshed, is often referred to as the remanence effect. Cold boot attacks exploit the remanence effect and can be executed in different ways. In [40] Halderman et al. presented a method for executing a Cold Boot Attack resetting the target system using the reset button. After boot the system using a special imaging USB stick that contains a minimal operating system used for acquiring the content of memory. Ideally, the original contents of RAM are maintained and can be recovered, apart from those parts that have been overwritten by the acquisition software, as reported by Bauer in [38]. Unfortunately, is possible to easily work around by using protection mechanisms like BIOS passwords.

Another way to execute the Cold Boot Attack is to physically remove the RAM module at runtime from the device under investigation and putting it into an acquisition system which performs a memory image extraction. As reported in [40], extracting residual memory contents requires no special equipment. When the system is powered on, the memory controller immediately starts refreshing the DRAM, reading and rewriting each bit value. At this point, the values are fixed, decay halts, and programs running on the system can read any residual data using normal memory-access instructions. One challenge is that booting the system will necessarily overwrite some portions of memory. In tests reported in [40] it is noted that the BIOS typically overwrote only a small fraction of memory instead of loading an entire operating system. That's the reason why in the solution proposed by Haldermann et al. it has been used a tiny special-purpose programs that, when booted from either a warm or cold reset state, copy the memory contents to some external storage. Several researches prove that if the semiconductors are properly cooled before transplantation, they will retain most of their content for a long period of time. As reported by Bauer in [38], it is beneficial to freeze the DRAM modules using cooling spray in order to increase the remanence effect. This kind of method offer higher level of availability, integrity and atomicity compared to the first approach.

Note that both approaches have their pros and cons. The first method is easier, as there is no need to open a running machine, to remove a DRAM module and to insert it quickly into a second machine. However, this approach bears a significant risk: the digital forensic investigator does not know anything about possible boot protection mechanism. In those case where a protection mechanism is enabled, the machine will not boot after the cold boot attempt. Moving the DRAM to a second machine, on the other hand, could be very hard. However, the digital forensic investigator can perform the cold boot on a machine that is completely under control.

Gruhn and Muüller in [39] reported that the method proposed by Halderman in [40] cannot be repeated using modern DDR3 RAM technology because the image obtained from cold booting device is random for reasons linked to that kind of technology. The empirical measurements performed by Gruhn et al. on DDR1 and DDR2 showed the correlation between temperatures and RAM remanence, demonstrating that even minor cooling of the surface temperature of a RAM module by just 10°C prolongs the remanence effect notably. But the important contribution is the discovery that the elevation of the attack to currently used DDR3 RAM however fails, and the analyst in not able to accomplish any attack more advanced than the basic warm reset attack on DDR3 RAM (which can be prevented with a simple BIOS boot lock). As a new contribution in 2016 Bauer et al. [38] present an in-depth analysis of DDR3 scrambling showing how to use this knowledge to develop a practical method of descrambling DDR3 memory in real-world scenarios in order to exploits cold boot attacks in modern technologies. Lindenlauf et al. in [41] demonstrate that cold boot attacks on DDR1, DDR2 and DDR3 RAM using a particular hardware setup. Lindenlauf demonstrates that cold boot attacks on DDR3 SDRAM are possible using the right mainboard for the attack. The Lindenlauf's cold boot attack procedure consists of five main steps in order to demonstrate the feasibility of his method. In his experiment the GA-G41M-Combo board worked for both SDRAM types DDR2 and DDR3, the ASUS P53E main board only for the DDR3 modules.

1. Preparation of test data and cold boot USB stick.
2. Cooling of the SDRAM and measuring its temperature.
3. Removing power and starting the stop watch.

4. Moving the SDRAM module to the cold boot machine (optional).

5. Reconnecting power, stopping the stop watch and cold booting the machine from the prepared USB stick.

In the in Figure 2.20 is represented the offline analysis performed by Lindenlauf in order to show different error rate depending the temperature of the RAM memory before the acquisition. The analysis shows that the test image revels that some bit errors resulted in more significant errors in the visual representation of the Lena picture.

Figure 2.20. Visual Reprentation of Error Rate. (source: [41]).

Results of the visual representation using a test image are reported in the table below.

Figure	*Temperature* [C]	*Error Rate* [%]
a	-35	0.041
b	-5	0.273
c	15	1.756
d	30	34.284

2.2.6 Mobile phone acquisition

In this section are analised different methods for acquiring the content of the RAM memory of mobile phone, describing when and whether it is possible to dump the memory of mobile devices. Despite that the research presented in this dissertation is not strictly connected to mobile application resurrection, it is important to note that the principle could be the same and a porting of the project could be possible in the future. That is the reason why it is presented modern techniques used for dumping RAM memory of mobile phones. Due to the capabilities of modern mobile devices, mobile phone are used broadly in criminal activities especially in cybercrime. The volatile data stored in mobile phones usually contain important evidences regarding the crime. However, collecting these volatile data in a forensically sound manner would not be easy. As reported in [42] mobile phones play important role in digital crimes that makes forensics analysts to create series of investigation procedures for forensically sound investigation of these devices. However, mobile phones are produced in different models, with different architectures and this fact creates lots of difficulties in developing a global procedure for acquisition and investigation of all kinds of smart phones. As reported in previous sections, the most challenging task in the forensic investigation is the acquisition of volatile data. This fact persists also in case of a mobile phone.

One of the major differences between mobile devices and computers is that they have limited resources and the data in the volatile memory is not preserved. When the operating system of a mobile phone needs to free resources, it might remove or close some of the running processes that have lower priority without the user permission. Therefore, acquiring the volatile memory of the mobile phone is the most important step in mobile phone investigation procedure. As reported by Dezfouli et al. up to now, there has not been a forensic investigation procedure that covers all kind of mobile phones and a lot of researches mostly are based on specific types of mobile devices.

This indicates the need for a new forensic investigation procedure that can support all kinds of mobile phones in order to preserve these valuable data. In 2005, as reported by Dezfouli et al., Willassen came up with some approaches for retrieving the data stored in a volatile memory such as RAM. One of the models suggested by Willassen for acquiring the data in the volatile memory is de soldering the mobile phone, as reported in [43]. In this method, the chip is removed from the Printed Circuit Board (PCB) by heating the PCB to certain temperature and then specific reader defined by the mobile manufacturer can be used to read the chip. Another technique reported by Willassen is to use JTAG in-system programming in order to tap to the PCB's bus and then tap it to some other devices for extracting the data in the volatile memory. Another different approach proposed in 2008 by Distefano in [44] consists of a tool that is installed inside the media card and is supported by the phone itself performing the data acquisition of the volatile memory. The tool presented by Distefano et al. is called MIAT (Mobile Internal Acquisition Tool) and is the main tool used which has the ability to perform acquisition from several mobile devices simultaneously within a period of 10 to 15 minutes, as reported in [42].

The main problem of this kind of technique is that before the acquisition the mobile phone is rebooted in the recovery mode. After that the MIAT tool starts but some data and files might be modified or deleted. As reported by Dezfouli et al. MIAT makes fewer changes to the data in compare to other tools such as Paraben and XRY Forensic software that are as used as MIAT. Another interesting technique has been proposed in 2008 by Irwin and Hunt and presented in [45]. Such a technique leverages the mobile network, where the phone is connected to, for performing a mobile memory acquisition. This technique can be implemented over Wi-Fi or GSM networks. Their research mainly concentrates on process of data acquisition of windows mobile phones over the mobile network. As reported in [42] they developed three different tools for their research. The first tool is called DataGrabber that is installed on a workstation and is connected to the mobile network that mobile phones are connected to. When the mobile phone is connected to the workstation, the DataGrabber uses another tool called ActiveSync which synchronizes the information inside the mobile phones into the workstation to acquire the data from the mobile phone. Irwin and Hunt created another tool called CTASms (Contact, Task, Appointment and SMS) that copies the data extracted from PIM (Personal Information Manager) and copies them to the same folder inside the workstation where the DataGrabber exists. The last tool developed by Irwin and Hunt is the SDCap (Storage Device Capture) which uses either TCP/IP or UDP/IP protocols to allow investigators to remotely transfer the data associated with the mobile phone over the mobile network from the media card, as reported in [45]. The problem of these tools is that they still make some changes to the data and make some contamination on the volatile content of the mobile phones. Another disadvantage of these techniques is that they are not architecture- and platform-independent. Dezfouli et al. in [42] propose a different approach for acquiring the volatile data inside a mobile phone in a forensically sound manner that minimizes the chance of evidence modification. Despite that each mobile phone has its own technology and architectural design, there is a basic structure shared in all of them including having a temporary memory which caches every instruction that the user makes before processing it or terminating the old processes when a new process is instructed by the user.

To overcome all the problems mentioned before in mobile forensics investigation and to go beyond architecture-based methods, Dezfouli et al. propose a solution applicable to any types of mobile phones such as windows mobile phones, android mobile phones and iPhones. In order to preserve the volatile data in the memory of mobile phones, the Dezfouli's solution reserves part of the mobile phone's internal memory as backup. All the volatile data inside the memory of the device will be stored inside this space that is reserved as backup. This backup is updated in a certain period of time with the volatile data in order to store all the processes that are running on the mobile phone. Even if a process is already terminated due to limited resources of the mobile phones or it already served its purpose and was not required any-more, it still can be found inside the backup folder. Using this approach allows the device to manage the backup space and at the same time collect the appropriate volatile data. This solution makes the volatile data acquisition process much easier and faster for the investigators and also preserves any evidence contamination or modification to the data during this process. As reported in [42] the method does not use any external tools for extracting the volatile data from the memory and does not create any extra files on the mobile phone which results in modification of the volatile data. In this approach,

the investigators only need to access the backup folder in order to acquire the volatile data of the mobile phone. The only thing required for this approach to be implementable is that all the mobile phone manufacturers must include this backup space in the internal memory of their products.

Another important contribution to the research on mobile phone acquisition and analysis is given by J. Sylve at al. in 2012. The research, as reported in [46], discusses some of the challenges in performing Android memory acquisition and presents a new kernel module for dumping memory called DMD. The DMD acquisition tool supports dumping memory to either the SD card external storage on the phone or via the network. Since Android does not support a memory device that exposes physical memory, i.e. in Linux there exists /dev/kmem, and furthermore does not provide APIs to support userland memory acquisition applications, several steps are required in order to prepare the smart-phone. Memory acquisition of physical memory requires gaining root privileges on the phone so that code can be loaded into the OS kernel to read and export a copy of physical memory. The first step in the preparation process is gaining root privileges on an Android phone. This step is commonly referred to as "rooting" and there exists a lot of methods that allow the elevation of normal user process to root access where the user id is 0. As reported by J. Sylve there are great concerns about using privilege escalation exploits to obtain root privileges, and an investigator should only use rooting techniques that have been verified to work reliably on a particular phone and for not obtaining malicious code. A rooting tool-kit with verified functionality is therefore a useful component of a live forensic investigator's tool set, along with proper acquisition tools. Once exploited, an Android process continues to execute as root until closed, which provides a vector for loading code into the kernel. The binary containing the exploit can be transferred to the target phone in a number of ways, but the most portable method to transfer files to and from the phone is through the adb application that is distributed with the Android SDK. One of the major issue that affects all kernel-based techniques is the portability across variety of phone models. It is a very difficult task try to perform a kernel module loading that is mobile-phone independent. Moreover, if module verification is enabled, the kernel performs a number of sanity checks to ensure that the module was compiled for the specific version of the running kernel. If any of these checks fail, then the kernel refuses to load the module.

In order to support acquisition of kernel memory across all Android devices, we have developed a kernel module that acquires a copy of system RAM with minimal interaction from the investigator. The DMD tool developed by Sylve et al. performs several steps reported below.

- Parsing the kernel *struct iomem_resource* to learn the physical memory address ranges of system RAM.
- Performing physical to virtual address translation for each page of memory.
- Reading all pages in each range and writing them to either a file (typically on the device's SD card) or a TCP socket.

As reported in [46] the memory dump is written directly from the kernel to limit the amount of interaction with userspace saving a substantial number of system calls and other kernel activity that is necessary when using userland tools such as dd and cat. Adherence to the community forensic soundness guidelines determines if evidence will be admissible in court and usable in other legal settings. Sylve et al. believe DMD technique meets basic forensic soundness standards. The method attempt to minimize the impact on the target device when transferring data to and from it. Once connected the phone through the USB, only a single binary (the kernel module) needs to be transferred and executed to perform the acquisition. The kernel module requires just 70 KB and uses very few kernel functions to acquire memory. As explained previously, minimal interaction with userland is needed beyond loading the module, since all reading and writing of data to files or via the network is handled within the kernel. This saves hundreds of system calls and other function invocations that would otherwise need to be performed. As an example Sylve et al. produce a 99% accuracy on a system made up 512 MB of RAM memory using DMD.

2.2.7 Evaluation platforms

In the section 2.2.4 has been reported different evaluation criteria of memory dump in order to understand whether a dump can be evaluated as an exact copy of the original source or not. In the section 2.2.5 has been reported different approach for acquiring the content of volatile memory focusing on different between them, analysing pros and cons of techniques. The purpose of this section is to present different methodologies and platforms that are able to evaluate a memory dump basing on the criteria reported in the section 2.2.4.

As reported by [27] one of the major challenges of evaluating memory acquisition tools is that they might perform differently depending on the version of the OS, the configuration of the operating system, and the hardware that is installed. It is also important to emphasize that virtualisation platforms are often far more predictable and homogeneous than real hardware, which means they might not provide a good indication of how a tool will perform in the heterogeneous world of real hardware.

From an operational perspective a trusted forensic acquisition tool must acquire evidence in a manner that is accurate, complete, documented, and with robust error logging. One of the major problem is that current acquisition tools when they fail they often silently fail. This problem prevents investigators from realizing that a problem even exists until they reach the analysis phase, at which point it is frequently too late to go back and acquire another memory image from the target system. No evidence gathering technique is free from error as reported in section 2.2.3. However, if an acquisition tool reliably logs errors, then the analyst can decide how to deal with the error. As reported in [27], just because a tool did not crash or freeze the target machine, that does not mean it produced an accurate and complete memory "image". Some researches have been done during the last years about how is possible to evaluate the accuracy and the completeness of an acquisition method. In 2013 Vömel and Stüggen published research about an evaluation platform. They present in [47] the platform for forensic memory acquisition software. This platform is capable of measuring distinct factors that determine the quality of a generated memory image, specifically its correctness, atomicity, and integrity in an in-depth and repeatable manner. The reason why this tool could be very helpful is that by measuring instead of estimating factors for sound memory imaging, investigators have a starting point for comparing available products more reasonably and better appraise their individual strengths and weaknesses. As reported in [47] using the tool presed by Vömel et al. the analyst will be able to whether an acquisition utility produce a snapshot that equals the size of the physical address space. Moreover, whether the created snapshot contain the data that was stored in a memory page at the time said page was imaged. Subsequently, is possible to understand how the acquisition utility cope with errors and areas of memory that cannot be accessed and how much the concurrent activity interfere with the imaging process. Other questions could be what is the impact of an acquisition utility, and how much memory is changed when the application is loaded into RAM or what is the amount of memory that is changed in the course of the imaging period. The method uses a white-box testing methodology, inserting a number of hypercalls that inform the platform of important system events and operations such as the point of time when the acquisition process is started or ended or when a page of RAM is duplicated. The platform intercepts the hypercalls, creates a protocol of the different activities, and generates its own view of the system state. This view is matched with the produced memory snapshot in a later analysis phase to derive the value of the described metrics. In order to understand where the hypercalls are inserted it is important to understand what is the general algorithm for an acquisition tool in order to acquire the memory. As represented in Figure 2.21 the design of an acquisition program can be quite simple in practice. As said by Vömel et al. errors may particularly be introduced if the size of the physical address space is not correctly calculated, or the return values of called functions are not rigorously tested.

An important point to note is that the Vömel and Stüggen's method is applicable to all memory acquisition applications for which access to source code is given. As reported by Vömel et al. the dependency of the time that is required for the imaging process is strictly linked to the degree of correctness, integrity and atomicity. The more the time is used, the more is difficult to perform a good acquisition. Using their platform, the degree of such consistency violations can be estimated.

In order to determine the correctness of an acquisition solution, a view of the guest's physical address space is created in parallel to the imaging process. This external view is then matched

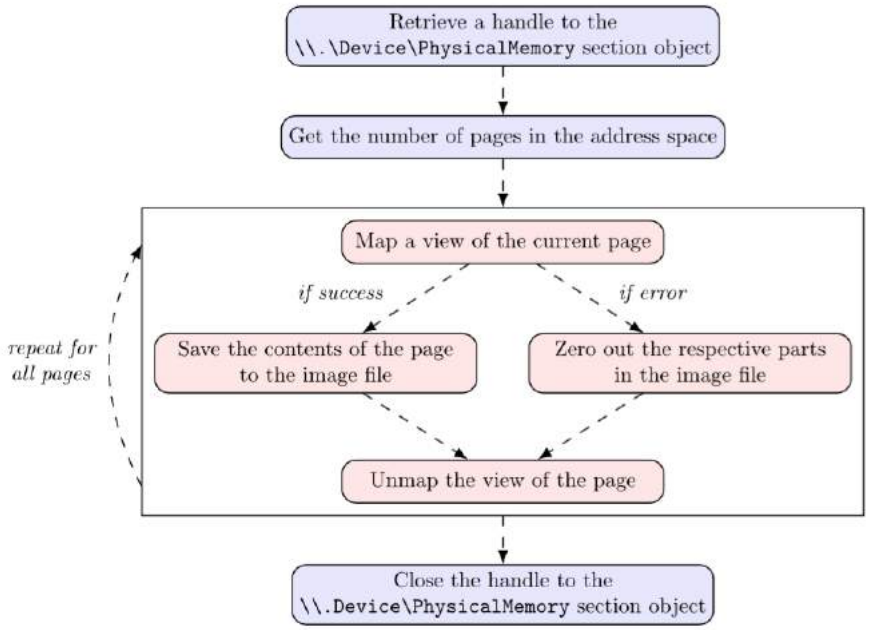

Figure 2.21. General aquisition algorithm on Windows. (source: [47]).

with the produced snapshot in a later analysis phase to identify possible deviations. The tool is able to verify whether the size of a forensic memory snapshot equals the size of the physical address space as well as whether the data stored in the image file corresponds to the contents of memory at the time the snapshot was taken.

About atomicity the tool does not measure the pure atomicity criteria but a slightly weaker metric, namely the degree of potential atomicity violation. To quantify the amount of potential atomicity violations it is used a callback function to observe all memory operations once the imaging procedure has started and keep track of the pages that have already been acquired. If a thread accesses a page after it has been imaged, it is inserted into a self-balancing binary search tree that serves as a watchlist for potential atomicity violators. When a thread performs a write operation on a memory region that has not been imaged yet, it is checked whether the corresponding thread identification number is included in the watchlist of potentially atomicity-violating candidates. The level of integrity is evaluated by creating copies of the physical address space at several points in time that are later matched in an analysis phase. The state of memory is duplicated shortly before the acquisition program is loaded into RAM as well as after the imaging process has finished. When the hypercalls are received, the platform temporarily suspends the execution of the guest system and creates a raw memory snapshot. Weakness of this evaluation platform are firstly it can evaluate just only 32-bit applications, secondly it can evaluate just 2 GB of ram image.

Results of integrity and atomicity testing three well-known tools such as Win32dd, Mdd and WinPMEM are depicted in Figure 2.23 and 2.22.

In 2015 Gruhn introduced a gray-box methodology with which memory address translation could be inferred. Gruhn notes the methodology can also be used to evaluate the memory snapshot correctness criteria.

A more recent research has been done by M. Gruhn and Felix C. Freiling in 2016 about how is possible to evaluate criteria presented in section 2.2.4 on a memory image. The method presented by Gruhn et al. is completely different from the previously white-box method. The Grhun et al.'s method is based on a black-box analysis technique in which memory contents are constantly changed via a payload application with a traceable access pattern. This way, given the correctness of a memory acquisition procedure, the technique is able to evaluate the atomicity and integrity as defined by Vömel and Freiling in [33].

The main weakness of all the white-box techniques is that they are only possible for tools that

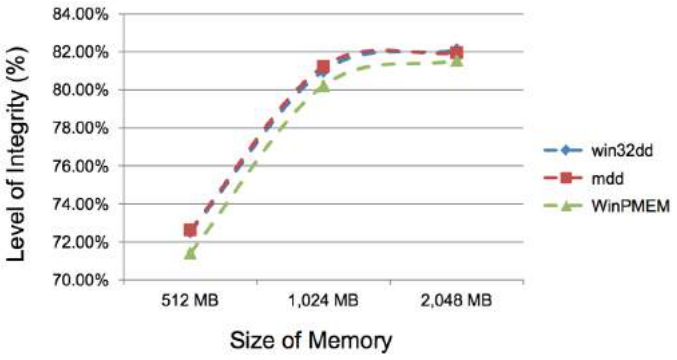

Figure 2.22. Evaluation atomicity using Vömel and Stüggen's platform. (source: [47]).

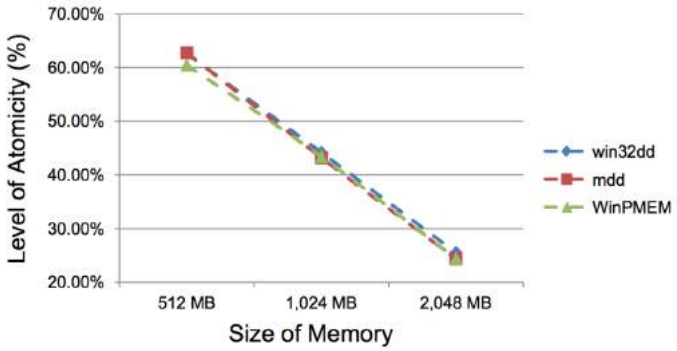

Figure 2.23. Evaluation integrity using Vömel and Stüggen's platform. (source: [47]).

is available the source code, thus severely restricting the scope of the measurement. It is clear that all kind of approaches presented in 2.2.5 such as DMA, cold boot attacks or SMM-based techniques can only be measured using a black-box approach.

In [34] Gruhn and Freiling present the first black-box methodology for measuring the quality of memory acquisition techniques. Their approach allows comparing not only different software utilities with each other but also to compare them with totally different approaches like DMA, cold-boot attacks and virtualization-based techniques.

The main idea of their approach is to apply the memory acquisition method to memory content that changes in a predictable way. The method uses a program that writes logical timestamps into memory in such a way that investigating the memory snapshot yields the precise time when a certain memory region was imaged. This allow inferring an upper bound in integrity and atomicity meaning that these criteria will be at most as bad for the respective procedures. Grhun et al. provide a framework to evaluate memory forensic tools using a black-box approach. The technical implementation of the black-box approach is based as follows. First, they have implemented a payload application called RAMMANGL.EXE. This payload application allocates memory regions. Each memory region is marked with a counter which is constantly increased by the payload application like a timer. Secondly they implemented an analysis framework that reads the counter value back from each region and runs statistics on them according to the estimation explained in the next section. The analysis framework is based on two measures called the atomicity delta and integrity delta. As reported in [34] *the atomicity delta is the time span between the acquisition of the first memory region and the last memory region.*

Then, the integrity delta *is the average time over all regions between starting the acquisition and acquiring that memory region.* Both measures are illustrated in Figure 2.24.

As reported by Grhun et al. the lower these values the better the atomicity and/or integrity respectively. As an intuitive example is reported in [34] where the memory of a system consists of four memory regions, each memory region containing one counter. Initially the counters are all set to 0. Once the memory acquisition starts, the counters are atomically increased every timer tick.

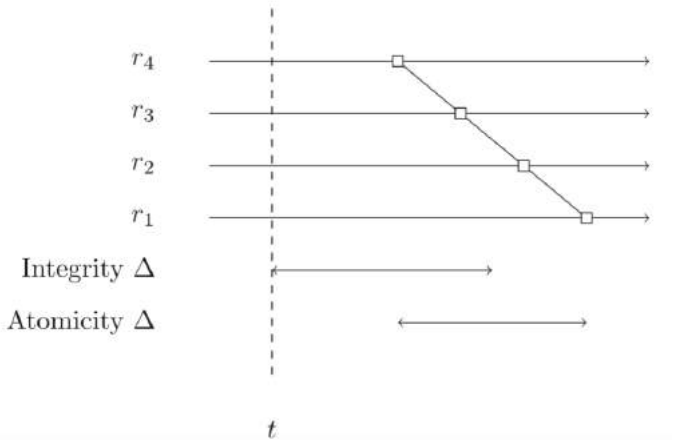

Figure 2.24. Atomicity delta and Integrity delta. (source: [34]).

So the ideal memory acquisition process should provide a memory image of $C = \{0,0,0,0\}$ that is all counters of all four memory regions still being in the exact state the acquisition was started. An example for an acquisition with high integrity but low atomicity would be the counter values $C = \{0,0,40,0\}$. This is indicated by a high atomicity delta of 40 and an integrity delta of 10. An example for an acquisition with high atomicity but low integrity on the other hand would have counter values $C = \{42,42,42,42\}$. In fact an atomicity delta of 0 indicates perfect atomicity.

Interesting results have been discovered for the first time using this approach. In summary it can be argued that there is quite a difference regarding atomicity and integrity considering different acquisition methods. However, as can clearly be seen from the acquisition density plot in Figure 2.25 that the different methods seem to cluster. For example, the kernel-level acquisition methods are all pretty identical with regard to atomicity and integrity, as well as acquisition speeds. On other group would be DMA acquisition, here represented by the inception toolkit. Then user-mode dumping, which should be split into methods suspending process execution and methods that do not. Last but not least are the ultra high atomicity methods, which can not even be distinguished any more in Figure 2.25 because the all cluster along the y-axis. These are virtualisation and emulation methods and the physical cold-boot RAM attacks.

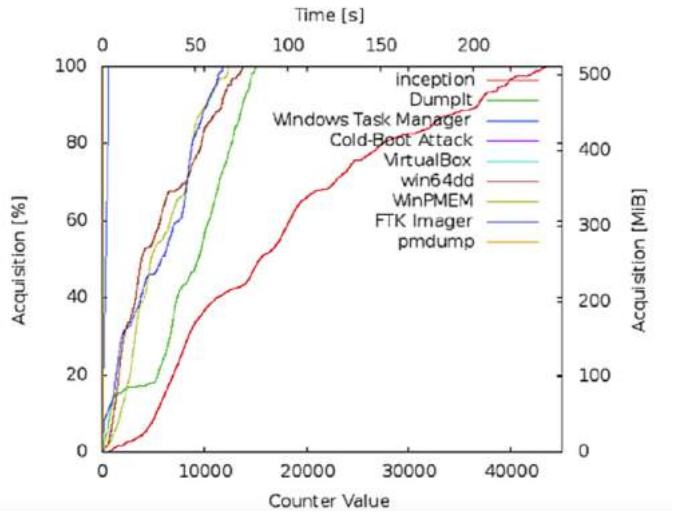

Figure 2.25. Results of black-box approach. (source: [34]).

Another useful representation as a result of Gruhn et al.'s research is depicted in Figure 2.26 that represents a matrix composed by atomicity/integrity delta.

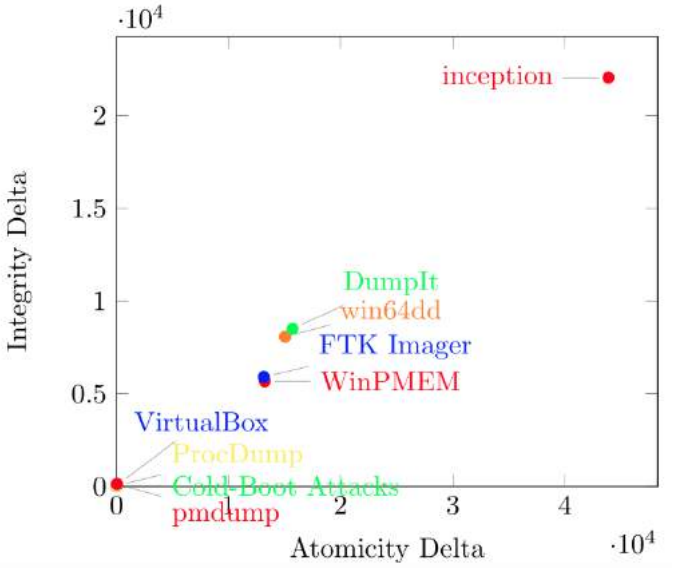

Figure 2.26. Atomicity/Integrity delta matrix. (source: [34]).

2.2.8 Memory dump formats

Usually, in the gathering evidences phase, the person who is responsible for acquiring the memory is not the same person that is in charge to analyse the memory dump. Thus, the analyst cannot get the opportunity to share best practices with the person will acquire the memory, i.e. recommend some tools or request the evidence in a particular format. Moreover, the analyst have to deal with what someone else has acquired previously. Luckily, almost all the analysis tool, such as Volatility, uses address space to automatically identify the memory dump format. Tools cycle through all supported formats until they can identify the appropriate address space for the target.

Nevertheless, it is important to understand how many formats exists and what are the differences between them. Memory images can be stored in a variety of formats. The simplest format for a memory image is a raw image. It is a binary file that represents an exact copy of the physical address space of the target system. As reported in [31], more sophisticated formats adopt a sparse approach. By providing metadata on the physical address of memory regions and their respective file o sets in a header of the file, they don't have to carry padding for inaccessible memory regions. Because of this sparseness, they are much smaller than a raw image and roughly amount to the size of physical memory.

ELF and Mach-O core files are binary file formats to store dumps of virtual memory and debugging information, which makes them ideally suited to also store physical memory images. However, at the time of writing there is no general method of storing metadata in a non-raw memory dump. For example, WinPMEM (Cohen, 2012) is able to store additional metadata like the page file or the address of the kernels page tables in a YAML Ain't Markup Language (YAML) footer at the end of any container file.

In the successive sections several memory dump formats are presented focusing on differences between them. The main memory dump formats, as reported in [27], are Raw Memory Dump, Crash Dump, Hibernation File and Machine Memory Dump.

Raw memory dump

As reported in [27] the raw memory dump is the most widely supported format among analysis tools. It does not contain any headers, metadata, or magic values for the file type identification. It is a binary file that represents an exact copy of the physical address space of the target system. Inaccessible regions, such as device memory regions, are zero padded and the resulting file has the same size as the physical address space. This is done to preserve the physical address of data in

the image. Each memory region is located at the same offset in the image file as it was stored in the physical address space. Physical memory references can thus be resolved by interpreting them as a file offset. Note that because of the padded MMIO regions this file can be much larger than the amount of memory installed in the system. Because of its simplicity, the format is supported by most major memory analysis frameworks.

Crash dump

Crash Dump file, especially Windows Crash Memory Dump was designed for debugging purposes. Crash dumps begin with a `_DMP_HEADER` or `_DMP_HEADER64` structure. The Signature member contains `PAGEDUMP` or `PAGEDU64`, respectively. The header identifies the major and minor OS version, the kernel DTB (DirectoryTableBase), the addresses of the active process and loaded kernel module list heads, and information on the physical memory runs. It also shows the bug check codes, which a debugger uses to determine the cause of the crash. The following list describes how is possible to lead the target system to generate a crash dump file.

- Blue Screens: It is possible to configure a system to create a crash dump when a Blue Screen of Death (BSoD) occurs. Common program for creating such a situation is to intentionally run programs that contain bug driver that will cause the Blue Screen of Death. This method powers down the system hard, which might result in loss of useful evidences.
- `CrashOnScrollControl`: Some PS/2 and USB keyboards have special key sequences that produce a crash dump. However, these methods typically require pre-configuration of the registry and BIOS.
- Debuggers: Sometimes debuggers have the possibility to create crash dump files. If the analyst is attached to a target system using a remote kernel debugger such as WinDBG, he can use the .crash or .dump commands.

All the methods described above for creating a crash dump suffer from a few similar weaknesses, as reported in [27]. They typically do not include device memory regions or the first physical page, which might contain a copy of the Master Boot Record (MBR). Moreover, it is possible to subvert these techniques just disabling the kernel debugger.

Hibernation file

A hibernation file `hiberfil.sys` is a compressed copy of volatile memory created by the system during the hibernation process. Hibernation files consist of a standard header (`PO_MEMORY_IMAGE`), a set of kernel contexts and registers such as CR3, and a list of data blocks that are compressed. About Windows Hibernation File the compression format used is basic Xpress or Huffman and LZ encoding. It is important to remember that during the analysis of a certain segment, it is needed to decompress the section. A good practice is to convert the hibernation file dump in a raw memory dump and analyse the raw memory dump in order to save time. As reported in [48], to create a Windows Hibernation File, it is needed to activate hibernation in the kernel setting and then turning off the machine. The resulting files are under `C:\hiberfil.sys`. A copy of that path is necessary for the analysis.

Virtual machine memory dump

Depending on the Hypervisor and the version, several files are created dumping a Host OS. For example in some case using VMWare Hypervisor the VM's memory is entirely contained in a single `.vmem` file. In other cases, instead of a `.vmem`, it is possible to have a list of `.vmsn` or `.vmss` (saved state), which are proprietary file formats containing memory and metadata. At offset zero of the metadata files, there is the `_VMWARE_HEADER` that contains a 8 Byte Magic number that must be `0xbed2bed0`, `0xbad1bad1`, `0xbed2bed2`, or `0xbed3bed3` as reported in the listing 2.27.

```
#_VMWARE_HEADER
0x0:Magic
0x8: GroupCount
0xc:Groups


#_VMWARE_GROUP
0x0:Name
0x40:TagsOffset


#_VMWARE_TAG
0x0:Flags
0x1:NameLength
0x2:Name


#DBGFCOREDESCRIPTOR
0x0:u32Magic
0x4:u32FmtVersion
0x8:cbSelf
0xc:u32VboxVersion
0x10:u32VBoxRevision
0x14 :cCpus
```

Figure 2.27. VMWare and VirtualBox structures.

The Groups member is a list of `_VMWARE_GROUP`. In the `_VMWARE_GROUP` struct the field `TagsOffset` specifies the address of the list `_VMARE_TAG`. Using information in the TAG is possible to find the physical memory data in the metadata files. On the other hand, using a different Hypervisor such as VirtualBox is it possible to create different dump in various dump formats. Using the method described in the section 2.2.5 the resulting dump is in raw format. Another method exploits the `vboxmanage debugvm` commands. This method creates an ELF64 core dump binary file with custom sections that represent the guest physical memory. The structure of a ELF64 core dump is reported in the listing 2.27. The ELF64 core dump file have several custom program headers. The header `PT_NOTE` has the value `VBCORE`. This segment contains the `DBGFCOREDESCRIPTOR` structure. This structure contains the VirtualBox magic signature `0xc01ac0de`, the version and number of CPUs for the target system. Each segment `p_paddr` member in the headers is a starting physical memory address. The `p_offset` member is the address of physical memory chunk. The `p_memsz` represents the size of the chunk in bytes.

Converting dumps

Sometimes the analyst uses a tool that does not support all existing formats. In such case a conversion could be the solution to the problem. Several tools exist on the market for performing such a conversion. As reported in [27] the raw format is the most widely supported, so that often becomes the destination format during the conversion.

A useful tool is `MoonSols Windows Memory Toolkit` (MWMT) that is a toolkit provides utilities to convert hibernation files and crash dumps into raw format. Another well-known tool is `VMware vmss2core` that can convert VMware saved state `.vmsn` or snapshot `.vmsn` files into crash dumps compatible with the Microsoft WinDBG or `gdb`.

Another way to convert memory dump files is to use specific `Volatility` plugin such as `imagecopy` and `raw2dmp`. The first plugin can copy out a raw memory dump from crash dump,

```
#Example using imagecopy
$ python vol.py -f win7x64.dmp --profile=Win7SP0x64 imagecopy -O copy.raw


#Example using raw2dmp
$ python vol.py -f memory.raw --profile=Win8SP0x64 raw2dmp -O win8.dmp
```

Figure 2.28. Examples using imagecopy and raw2dmp Volatility plugin.

hibernation file, VMware, VirtualBox, QEMU, Firewire, Mach-o, LiME, and EWF file formats. The `raw2dmp` plugin can convert a raw memory dump into a Windows crash dump for analysis with Microsoft's WinDBG debugger. Some examples are reported in the listing 2.28. In the first example the conversion is from a crash dump to raw format. The second example performs a conversion from a raw format into a crash dump.

2.2.9 Anti-memory forensics techniques

Anti-Forensics has been defined as *any attempt to compromise the availability or usefulness of evidence to the forensic process* by R. Harris in 2006 in [49]. As reported in [29] by Stüggen and Cohen anti-forensic attacks fall into two large categories. Those techniques which prevent evidence from being acquired, and those techniques which delete data from the collected evidence such that the collected evidence can not be analysed.

As reported in [31] the first kind of techniques intercept system call APIs (hooking). This rootkits can filter the view of the system memory. The second kind of techniques are based on the Direct Kernel Object Manipulation (DKOM). DKOM attacks directly manipulate kernel data structures to hide processes, threads, network connections and other malware traces. An example of the first category anti-forensics techniques has been presented by Haruyama and Suzuky in 2012 called One-Byte Abort Factor Attack. They presented a method to prevent analysis by Volatility Framework just overwriting the `KdDebuggerDataBlock.OwnerTag` string. As reported in [50], this example shows that important signatures are easily over-written by malware in such a way that memory analysis can fail to find it. Because this data structure is only used for kernel debugging, destroying it does not impact regular system operation. The problem such technique is that can be overcome with sufficient effort. The analyst could improve his methods to deal with these techniques and he can analyse again previously acquired memory images and find evidence they missed in past analysis.

The second approach attacks the acquisition phase in a permanent way, because of the volatile nature of RAM. If malware succeeds in hiding its traces from a memory image, it is very impossible for the investigator to acquire another image in the same condition.

A number of effective anti-forensic techniques against memory acquisition have been proposed in the literature. One technique is called substitution attacks, in which data fabricated by the attacker is substituted in place of valid data during the acquisition process. In such situation for the analyst is not possible to gather all evidences because true data are substituted. Another technique involves a root-kit that disrupt the acquisition process altogether, e.g. hanging the hardware, when detecting the presence of a forensic process. This approach is really effective against memory acquisition techniques, since the volatility of the evidence does not permit the investigator to reacquire the memory under the same conditions.

As reported by Stüggen et al. due to lack of research and understanding of anti-forensic techniques in memory acquisition, current commercial or free memory acquisition tools do not appear to implement mechanisms to protect their operations against anti-forensic attacks. Bypassing software-based memory acquisition tools that depend on the OS was demonstrated by the DDFY tool in 2006 by Bilby. As reported in [29] this tool hooks the physical memory device and filters

certain pages from being read through this interface, providing instead a cached copy. In principle, any OS facility can be hooked in a similar manner in order to subvert the acquisition tool.

Hardware-based solutions were proposed as being resilient to rootkit manipulation but even hardware-based acquisition can be subverted using very low level manipulation of the memory controller hardware registers as demonstrated by Rutkowska in [51]. As reported by Cohen et al. the current generation of forensic memory acquisition tools produce correct and forensically sound images under idealized lab conditions but they completely fail when simple anti-forensic attacks are present. Cohen et al. have made tests in order to understand what are methods more resilient than other. They discovered that all tools tested can be subverted by simple anti-forensic attacks.

As reported by Stüettgen, all software memory acquisition tools must solve two primary challenges: the memory mapping and the memory enumeration.

- Attacks on Memory Mapping: memory acquisition software must map all physical memory into its virtual address space to be able to access it. By intercepting the memory mapping APIs in the OS, rootkits can change memory content that contain traces of their existence with a benign copy of this region.

- Attacks on Memory Enumeration: to prevent access to MMIO regions which can destabilise the system, software must enumerate the physical address space and identify all physical memory regions. This information is passed on to the OS by the firmware, which is generally not available to drivers at runtime. Software can query the OS for the memory map, which provides an overview on the physical address space. Rootkits can intercept these APIs to hide specific regions from memory acquisition software. While it is not possible to hide all system modifications this way, rootkits can still hide their code and data from a memory image. Depending on the implementation of the OS it is also possible to perform a DKOM attack on the memory map directly.

As reported in [31] memory acquisition drivers need to enumerate the physical address space before the acquisition. On Windows operation system, the function that is able to return this information is the undocumented symbol `MmGetPhysicalMemoryRanges()`. By patching this function to always return NULL, which is the failure default value for this function, we prevent drivers from learning about the physical address space layout. As reading from device memory can crash the kernel, this effectively prevents memory acquisition. An actual rootkit could simply return a modified version of the memory map, which excludes ranges it is trying to hide. Acquisition would then appear successful, while being incomplete.

In [29] is reported an example of Memory Mapping Attack. To actually access physical memory, acquisition drivers need to map it into the kernels virtual address space using `ZwMapViewOfSection()`, `MmMapIOSpace()` and `MmMapMemoryDumpMdl()`. Returning `NULL` in these function the acquisition tool will be subverted. It is important to note that `MmMapMemoryDumpMdl()` is undocumented and a modification does not alter the system stability. In [29] Stüggen proposed a simple Python kernel patcher that demonstrates how easily could be to subvert an acquisition tool using techniques described above that is reported for completeness in the listing 2.29.

Stüttgen et al. recently suggested that the memory acquisition process can be trusted with two conditions.

- The acquisition module has not been tampered with.

- All the operations are performed without relying on the operating system or any other untrusted software.

However, a more recent research presented by Zhang et al. in [52] shows that this assumption is not true. Zhang et al. presented Hidden in I/O Space (HIveS), an operating system (OS) agnostic anti-forensic mechanism, that is capable of evading the most updated software based memory forensics tools.

```
from volatility import session
from volatility.plugins.overlays import windows


def KernelApiPatch(session, symbol, patch):
        kernel_image = windows.pe_vtypes.PE(
                session.kernel_address_space,
                session.kdbg.KernBase)
        offset = kernel_image.GetProcAddress(symbol)
        session.kernel_address_space.write(offset,patch)


if __name__ == "__main__":
        s = session.Session(filename = r"\\.\pmem")
        return_null_shellcode = "\x48\x31\xc0\xc3"
        KernelApiPatch(s,"MmGetPhysicalMemoryRanges", return_null_shellcode)
        KernelApiPatch(s,"MmMapMemoryDumpMdl", return_null_shellcode)
        s.kdbg.Header.OwnerTag = "MOOF"
```

Figure 2.29. Python kernel patcher based on Volatility Framework. source([29])

Physical address space on x86 platform is shared between physical memory and I/O devices. The access to the physical layer is redirected to the memory controller or the I/O bus, depending on the address space. The physical address layout is used by forensic tools to understand where the physical memory regions are located. A memory acquisition tool understand this information interacting with the OS or BIOS assuming this layout is always updated and correct. The research presented by Zhang et al. shows that this condition can be easily violated using an anti-forensic mechanism called HIveS. HIveS alters the machine physical address layout while the system is in operational state by modifying registers in the processor. The basic idea is to map (or lock) the HIveS memory into the I/O space, so that any operation on the physical memory address will be redirected to the I/O bus instead of the memory controller. When the HIveS memory is locked, its memory contents cannot be accessed by any processor. When the attacker wants to access the HIveS memory, he would first unlock the memory region by mapping it back into the memory address space. HIveS requires kernel privilege to manipulate the system registers, and thus it is not available to user space malwares. HiveS redirect memory accesses at a hardware level. It results that all software based accesses to the memory are subverted.

2.3 Tools

The development of this thesis required two different kinds of tools: a memory forensic tool and a checkpointing tool. In this section of the Background the tools will be described and their use will be explained. Firstly the Volatility Framework will be explained, it is one of the most commonly used forensic framework. It is different from specific tools, which are able to perform a particular and well defined forensic analysis, it is, instead, a generic tool that offers to analyst a shell like interface to a physical memory dump, in order to permit them to perform something similar to the live interrogation but in a repeatable manner. In a second moment, the checkpointing tools will be explained and in particular the CRIU framework. Checkpointing tools permit saving a snapshot of an application's state, so that it can restart from saved state in case of failure. CRIU is one of the most famous tool in this area, it is a software for Linux operating system. Using it, it is possible to freeze a running application in a collection of files on disk. Then using those files is possible to restore the application and run it exactly as it was during the time of checkpoint.

2.3.1 The Volatility framework

The Volatility Framework is an open collection of forensic tools, developed in Python. Volatility is used by analysts for the extraction of digital artifacts from a physical memory dump. This section explains the benefits of using Volatility and describes some of the internal components that make the tool a true framework. Firstly, it is important to understand why Volatility was chosen for this thesis. It is not the only open source memory forensic tool, but it has some unique features which make it different [27]. It is a single framework able to analyse memory dumps from various kinds of system, it has a modular implementation which make it easy to be extended for new operating systems or for the extraction of new digital artifacts. Volatility is entirely written in Python so many libraries can be used and integrated by developers, furthermore it can run in all operating systems. It is very extensible by developing of plugins, the framework offers a lot of API which can be used by developers for innovate generating new functionalities. Volatility uses fast and efficient algorithms, it is maintained by a big community composed by commercial companies, law enforcement, academic researchers and security companies. Volatility performs operation completely independently from the Operating System. It is focused on malware detection and incident response so some functionality which could be required by a different kind of analysis could not be implemented. The Volatility Framework consists of several components which work together to provide various features. There are some key components that will be explained in this section.

One of the most important concept which have to be explained is the VType structure, it is Volatility's structure definition, operating systems and applications are often written in C, with a large use of data structures to organize data. There is the need of a way to represent C data structures in Python source files. VTypes enable you to do exactly that, for example, as described in [7] a C structure is defined as:

```
struct process {
    int pid;
    int parent_pid;
    char name[10];
    char * command_line;
    void * ptv;
};
```

The structure has five members: two integers, a character array, a pointer to a string, and a void pointer. The equivalent in the VType language is as follows:

```
'process' : [ 26, {
    'pid' : [ 0, ['int']],
    'parent_pid' : [ 4, ['int']],
    'name' : [ 8, ['array', 10, ['char']]],
    'command_line' : [ 18, ['pointer', ['char']]],
    'ptv' : [ 22, ['pointer', ['void']]],
}]
```

It is a series of Python dictionaries and lists, the first key is the name of the structure, then there is the total size, subsequently for each field there is the name as a key and an array containing the offset and the type as value. VTypes have to be generated before using Volatility, it is impossible to create them by hand, they change with each new version of an operating system, indeed there are various tools for automatically generating VTypes from the Operating System debugging symbols. Another important concept is the Volatility object, it is an instance of a structure which exists at a specific address in a defined address space. It is possible to consider the address space as an interface to the memory image file, it handles virtual-to-physical-address translation, it must have the knowledge of memory layouts for reconstructing the right view of the memory. In order to create an object, the analyst has to call the `obj.Object()` API defining the structure type, the starting address and the address space, then it is possible to access to all fields of the structure for

using values. Classes enables developers to extend the functionalities of objects, for example it is possible to attach new methods or properties which can be used by many plugins. For example a simplified structure representing a process in Linux can be defined as follow:

```
import volatility.obj as obj

class task_struct(obj.CType):
    """An object class for Task_Struct"""
    def getcwd(self):
        ...
    def get_elf(self, elf_addr):
        ...
    @property
    def uid(self):
        ...
    @property
    def gid(self):
        ...
    @property
    def euid(self):
        ...
    def bash_environment(self):
        ...
    def lsof(self):
        ...
    def netstat(self):
        ...
    def get_process_address_space(self):
        ...
    def threads(self):
        ...
    def get_proc_maps(self):
        ...
```

Address spaces

An address space is an interface able to provide a flexible and consistent access to data in RAM. As explained in previous sections address space can be identified as virtual or physical, a forensic tool has to work with both point of view. The virtual or paged address space are used for reconstructing the virtual memory. Volatility uses many of the same algorithms which processors use for address translation, so it is possible to gather data independently from the target operating system. The virtual address space contains the portion of memory that programs on a system can access at the time of the acquisition. The virtual address space can be broken down into kernel and process space. The kernel space provides a view of the memory accessible to modules running in kernel mode. A process space provides a view of user memory from the perspective of a specific process. The physical address space is related to memory dump file format, Volatility is able to manage many file formats, it contains memory ranges marked by operating system as free or de-allocated allowing to find residual of terminated processes. In order to use or develop new plugins it is important to understand how manage address spaces and how to read from it. Volatility offers a base class for all address spaces objects, `BaseAddressSpace`, which maintains all standard functions available to an address space. Every implementation of a new address space has to overload or inherit these methods and properties. In order to access to an address space, a related Python object has to be created, the `BaseAddressSpace` class offers a constructor which generate the object. Analysts can read from the instantiated space using two different functions, the first is the `read( addr, length )` method, it takes an address and a length as parameters and it reads from `addr` to `addr+length`. The second offered method is `zread( addr, length )`, it is similar to the previous one but it returns 0's in any location which can not be read, it can be useful when a user

wants a stream with the size equals to the specified length. Moreover, the base class offers other methods for manage an address space, for example the `get_available_pages`, which returns the list of available memory pages in the selected address space.

Volatility profiles

A collection of VTypes is called profile, it contains also informations about the operating system, the system calls, the native types definition and the system map. Volatility includes by default a many profiles for windows systems, instead, for Linux systems there are not profiles, because Linux has a large number of different kernel versions and customized kernels. In order to analyse a Linux memory dump, it is fundamental to build a suitable profile. Building profile is not a complex operation, it consists on generating VTypes and the System.map file for the target kernel version. In order to generate VTypes the `module.c` file, distributed with Volatility, has to be compiled. It is a kernel module which declares all data structures needed by the framework. Sometimes modification to `module.c` file can be useful if some missing data structures are needed by the analysis. Module compilation requires `dwarfdump` command, which is normally used for compiling kernel modules.

```
> dwarfdump -di module.ko > module.dwarf
```

After VTypes generation, symbols are needed. They can be found in a `.map` file stored in `/boot` directory of the system. The `System.map` file contains the addresses of all symbols used by kernel, later used by Volatility for locating data structures in the dump file. Symbols change at each kernel update, of recompilation so it is important to check to use the right `.map` file for the analysis. When symbols and VTypes are collected, profile can be generated putting them in a zip archive. As mentioned above, it is possible to modify the `module.c` file in order to generate needed VTypes. Sometimes analysts need some unconsidered data structures in order to get a better view of the target memory dump, they have to modify the module file including those structures. The inclusion operation requires libraries to be included and then desired structures have to be used in `module.c` file instantiating them in some way, for example declaring a variable. After module recompilation selected structures will be available as VTypes in the target profile, and it will be possible to use them in all plugins.

Plugins development

The core Volatility Framework includes over 200 analysis tools but as mentioned before, it is a very extensible framework which can be expanded through plugins development. They are written to find and analyse specific components of the operating system and applications. Regardless of plugin type there are APIs and templates provided by the framework to design and integrate new ideas. To develop a new plugin, a Python class that inherits from commands `.Command` have to be implemented and some base methods must be overridden. In particular, the main functions which have to be implemented are `calculate` and `render_text`, the first is the method which performs the work, the second is where output has to be formatted. In short, the results of the calculation are passed to `render_text` for be rendered on the terminal. An example plugin can be written as follow:

```
import volatility.commands as commands

class ExamplePlugin(commands.Command):
    """Description of an example plugin"""

    def calculate(self):
        """Performs the analysis"""
        for proc in tasks.pslist(addr_space):
            yield proc
```

```
    def render_text(self, outfd, data):
        """Formats output"""
        for proc in data:
            outfd.write("Process: {0}\n".format(proc.ImageFileName))
```

Plugins can also use inheritance, so it is possible to extend an existing one for developing something new, for example a very common extended plugin is `pslist`, which lists all processes found in memory. When inheritance is used, only one function has to be overridden: the render text, as calculate function the `pslist` one will be used.

```
import volatility.plugins.linux.pslist as linux_pslist

class extending_plugin(linux_pslist.linux_pslist):
    """Plugin which extend pslist"""

    def render_text(self, outfd, data):
        """Data contains the list of task listed by pslist plugin"""
        for task in data:
            ...
```

Volatility frameworks permits plugins to accept arguments which can be used by users to select options different from default. For example, a plugin, which requires to save something on a local file, could accept a parameter for specify the target directory where file have to be saved. It can be done in the `__init__()` method of the plugin.

```
def __init__(self, config, *args, **kwargs):
        self._config.add_option('DUMP-DIR', short_option = 'D', default =
            "./", help = 'Output directory', action = 'store', type = 'str')
```

Arguments support inheritance, so if a parent plugin had declared an argument, then it can be used in the new plugin.

Using Volatility

Volatility is very easy to use, after the installation and setup of profiles, it can be used for the analysis of memory images. The most basic command is the follow:

```
> python vol.py -f <DUMP_FILE_PATH> --profile=<PROFILE_NAME> <PLUGIN>
    [OPTIONS]
```

The main Python script is executed and the path to memory dump file, the name of the profile and plugin to execute are passed to it. As mentioned before, the identification of the right profile is very important in order to perform the analysis, it can be manually selected with `--profile` option, which tells what type of system the memory dump come from, specifying which data structures, algorithms, and symbols have to be used. If the analyst does not know what profile have to be selected, he can run the `imageinfo` plugin to understand which profile is related to the memory dump. This plugin uses `KDBG` scanning and it is available only on Windows systems.

```
> python vol.py -f dump_file.raw imageinfo
Volatility Foundation Volatility Framework 2.5
Determining profile based on KDBG search...
         Suggested Profile(s) : Win10x64, Win8x64
                    AS Layer1 : AMD64PagedMemory (Kernel AS)
                    AS Layer2 : FileAddressSpace
    (/Users/Analyst/Desktop/dump_file.raw)
```

```
                    PAE type : PAE
                         DTB : 0x187000L
                        KDBG : 0xf80002803070
          Number of Processors : 1
     Image Type (Service Pack) : 0
                KPCR for CPU 0 : 0xfffff80002804d00L
             KUSER_SHARED_DATA : 0xffffff78000000000L
           Image date and time : 2017-01-03 15:08:00 UTC+0000
```

When the right profile is selected it is possible to use all plugins available for the target operating system. There are many plugins for all main OS: Windows, Linux and Mac OS. Through them Volatility offers to analysts a shell-like interface to the memory dump, they can use commands as in a live interrogation in order to gather data. The main plugins used for the implementation of the thesis will be now explained, and the documentation of all available command is available on [7].

Plugin: The Volshell

One of the most useful plugin for developers and analysts is the `volshell`, it is a Python shell with the volatility environment which gives access to the memory image file. The `volshell` offers various commands for searching information in the memory dump. The main command of the `volshell` is `dt()`, it shows how a structure is composed, it works with volatility objects but also with names of VTypes, when a name is passed as parameters `dt()` shows how the VType is composed.

```
>>> dt("task_struct")
    'task_struct' (2376 bytes)
    0x0 : state ['long']
    0x8 : stack ['pointer', ['void']]
    ...
    0x280 : tasks ['list_head']
    0x290 : pushable_tasks ['plist_node']
    0x2d0 : mm ['pointer', ['mm_struct']]
    0x2d8 : active_mm ['pointer', ['mm_struct']]
    ...
    0x1e4 : pid ['int']
    ...

>>> p = proc()
>>> dt(p)
    [task_struct task_struct] @ 0xFFFF880036FD13B0
    0x0 : state             1
    0x8 : stack             0xFFFF8800797EC000
    ...
    0x280 : tasks           0xFFFF880036FD1630
    0x290 : pushable_tasks  0xFFFF880036FD1640
    0x2d0 : mm              0xFFFF88007964D400
    0x2d8 : active_mm       0xFFFF88007964D400
    ...
    0x370 : pid             3600
    ...
```

As previously mentioned, the `volshell` is an interface for analysts to the memory dump, through it data structures can be read and easily collected. It is a very powerful plugin, it works on a context which is the entire memory image file, but it is able also to work on specific process's context. With `ps()` command all processes in memory are listed and using the `cc( pid )` function

the process with desired `pid` is set as working context. It is possible to specify the context using the `pid` of a target process when launching the `vol.py` script with the argument `-p`.

```
>>> ps()
    Name            PID     Offset
    init            1       0xffff32007b988000
    kthreadd        2       0xffff32007b9896a0
    ksoftirqd/0     3       0xffff32007b98adb0
    kworker/0:0     4       0xffff32007b98c4f0
    kworker/u:0     5       0xffff32007b98dbc0
    [snip]

>>> cc(pid = 1)
    Current context: process init, pid=1 DTB=0x32007b
```

When the desired context is specified, analysts can get the `task_struct` structure using the `proc()` for reading properties and collect values, furthermore they can also read data directly from the selected address space in the hexadecimal format with command `db( address, length, space )`.

```
>>> p = proc()
>>> dt(p)
    [task_struct task_struct] @ 0xFFFF880036FD13B0
    0x0 : state             1
    0x8 : stack             0xFFFF8800797EC000
    ...
    0x280 : tasks           0xFFFF880036FD1630
    ...
    0x370 : pid             3600
    ...

>>> db(0xFFFF880036FD1630, 64)
    0xffff880036fd1630  90 a4 af 79 00 88 ff ff d0 03 91 7c 00 88 ff ff
    0xffff880036fd1640  8c 00 00 00 00 00 00 00 48 16 fd 36 00 88 ff ff
    0xffff880036fd1650  48 16 fd 36 00 88 ff ff 58 16 fd 36 00 88 ff ff
    0xffff880036fd1660  58 16 fd 36 00 88 ff ff 68 16 fd 36 00 88 ff ff
```

Using the Volshell helps developers to build new plugins, and to understand how it is possible to find or use some data.

Plugin: Pslist

Another important plugin is `pslist`, it enumerates processes by walking the active process list, the kernel uses this list to store active processes, it is stored in kernel-space and never exported to user-space. The `pslist` plugin is often extended in new plugins which have to work on a single process. The output is rich of information belonging to each process:

```
> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_pslist
Volatility Foundation Volatility Framework 2.5
Offset              Name            Pid Uid Gid DTB    Start Time
------------------  ------------    --- --- --- -----  ----------
0xffff88003e253510  init            1   0   0   0x370  15:16:22
0xffff88003e252e20  kthreadd        2   0   0   -----  15:16:22
0xffff88003e252730  ksoftirqd/0     3   0   0   -----  15:16:22
0xffff88003e283550  kworker/u:0     5   0   0   -----  15:16:22
...
```

Plugin: Process map

In order to understand how a process uses its address space, an often used plugin is the `proc_maps` it walks the `mm→mmap` struct of each process reporting regions of the address space. The output contains start and end addresses for each region, page permissions, page offset and inode number. The `proc_maps` can be used combined with the `dump_map`, which can dump on a file the selected region reading from the memory dump.

```
> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_proc_maps -p 1
Volatility Foundation Volatility Framework 2.5
Pid Start               End                 Flags   Pgoff   Inode   Path
--  ------------------  ------------------  ------  -----   -----   ----
1   0x0000000000400000  0x0000000000409000  r-x     0x0     1044487 /sbin/init
1   0x0000000000608000  0x0000000000609000  rw-     0x8000  1044487 /sbin/init
1   0x0000000001dc1000  0x0000000001de2000  rw-     0x0 0           [heap]
...
1   0x00007fff23e5f000  0x00007fff23e81000  rw-     0x0 0           [stack]
...

> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_dump_map -p 1 -s
    0x400000 -D dump
```

Plugin: Find file

Another commonly used plugin is `find_file`, it is able to list all files found in the file system. It returns for each file the name and the address of related `inode`. This plugin can be used mainly in two different ways. The first one consists on calling it with the `--listfile` argument, the output will be a list of files with related informations. Otherwise, if it is called with `--inode` option, followed by an `inode` address, it will try to extract the selected file from the memory dump. If the analyst knows exactly the name of desired file, it is possible to skip the `--listfile` option substituting it with `-F` option followed by the file name, it will return the `inode` address of specified file.

```
> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_find_file --listfile
Volatility Foundation Volatility Framework 2.5

Inode Number                  Inode     File Name
---------------- ------------------     ---------------
        ...
        130564     0x88007a85acc0       /tmp/file1.txt
        ...
        130590     0x88007a85abf0       /tmp/file2.pdf

> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_find_file -i
    0x88007a85acc0 -O file1_txt.dump
```

Plugin: Lsof

As mentioned before, Volatility offers to analysts a shell-like interface to the memory dump. There are many plugins which are very similar to shell command on a real system. An example is the `lsof` command, it prints the list of open file descriptors for each process, it is possible to select a specific process with the `-p` option. The `lsof` command is very useful for understanding if a process uses some files, sockets or more generically file descriptors.

```
> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_lsof -p 3578
```

```
Volatility Foundation Volatility Framework 2.5

Pid     FD      Path
-----   ----    -------------
3578    0       /dev/pts/0
3578    1       /dev/pts/0
3578    2       /dev/pts/0
3578    3       socket:[6745]
3578    4       pipe:[7345]
```

Plugin: Netstat

Regarding network connection, volatility offers many plugins which are very similar to shell commands, the most used for the thesis implementation is the `netstat` command. It helps analysts to understand how opened sockets were used in the target system. This plugin iterate over all open file descriptor searching for sockets, then it converts the selected structure to `inet_sock` which is a VType related to the network socket structure of the kernel. The output of the plugin is very similar to the output of shell command, it lists all open sockets distinguishing between protocols and specifying network addresses.

```
> python vol.py --profile=KaliLinuxx64 -f kali.lime linux_netstat
Proto   Source IP:Port  Destination IP:Port State   Process
-----   --------------- ------------------- ------  -------
TCP     192.168.1.80:22 192.168.1.4:52314   EST     sshd/2952
TCP     0.0.0.0:22      0.0.0.0:0           LISTEN  sshd/1548
UDP     0.0.0.0:215     0.0.0.0:0                   nmbd/2121
...
```

Community

One of the main strength point of the Volatility framework is the fact that it is an open source tool, it is maintained by a big community which fixes bug continuously and updates the framework permitting to use it with the modern operating systems. The community develops many plugins which are not included in the framework but analysts could easily use them in their analysis. The Volatility Foundation publishes the best community plugins in an official repository [53].

2.3.2 Check-pointing tools

Check-pointing is the technique of storing the volatile process state on a non volatile storage in such a way that process can be restarted from the point at which the checkpoint was created. The checkpoint is created by stopping the execution of the process, saving the whole process address space and kernel state to a file, and then resuming the execution of that process. No additional system resources are required, just the storage necessary for containing the checkpoint file. Well-known benefits, as reported in [54], of such technique include process migration, fault tolerance, and roll-back recovery. Often, checkpoints are made at regular time intervals during the execution of a process in order to have backup process states. With check-pointing feature, an application will be able to restart from the last check-pointed state instead of starting from the beginning which would have been computationally expensive in particular environments.

Use of storage space can be optimized when creating checkpoints at regular time intervals by saving only the difference between the current checkpoint and the most recent checkpoint. This is usually referred to as incremental check-pointing. A particular process state is that one before the termination of the process and this kind of checkpoint is called terminal check-point. Check-pointing technology creates a snapshot of a process address space that can be used to restart the given process, thus it must contain all details about the process. Checkpoints can be created very

quickly, without modifying the process being check-pointed. Check-pointing technologies are used also in malware analysis because check-pointing malicious processes does not affect them and does not notify any potential attacker who is controlling such processes. Since check-pointing creates non-volatile data from processes, including check-pointing technology in investigation process creates a new source of non-volatile evidences that can be used in a court. Increasing the amount of non-volatile data increases the amount of forensic evidence available to the digital forensic investigator.

In the case of a checkpoint, the checkpoint resides in a file and can therefore be digitally fingerprinted immediately following its creation. In a courtroom, this digital fingerprint can be verified to show that the checkpoint file remains unaltered. A time-stamp can also be included and verified with a digitally fingerprinted checkpoint file. As reported in [55], check-pointing tools can be classified into 2 different classes:

- Kernel-level tools: such tools are built into the kernel of the operating system. During a checkpoint, the entire process space is dumped to physical storage. The user does not need to recompile or re-link the application. Check-pointing and restarting of application is usually done through OS commands.
- User-level tools: these tools are built into the application which will periodically write their status information into physical storage. The dump of the process is usually done by sending a signal managed by the application that will dump itself. The restore is usually done by calling the application with additional parameters pointing to the location of restart files.

Another distinction found in the literature is the distinction between kernel-space and user-space check-pointing tools:

- Kernel-space tools: these tools are built into the kernel and work in the kernel address space.
- User-space tools: these tools are user-space applications able to dump and restore other processes in userland interacting with the operating system through several systemcalls.

As reported in [56] there has been relatively little news from the checkpoint/restart community in recent years. As reported in [57] several efforts has been done about process checkpointing, including BLCR, `DMTCP`, ZAP and `CRIU`. In order to move out the majority of the work out of the kernel, new tools has been developed such as `CRIU`, a checkpointing tool that works in user space, minimizing the amount of the in-kernel changes needed. The next section describes how to use `CRIU` and how `CRIU` works.

CRIU project

As reported in [58] `CRIU` (Checkpoint/Restore In Userspace) is a software tool for Linux operating system. Using this tool, it is possible to freeze a running application - or part of it - and checkpoint it as a collection of files on disk. Subsequently it is possible to use the files to restore the application and run it exactly as it was during the time of freeze. With this feature is possible to do application live migration, snapshots and remote debugging.

The checkpoint procedure relies heavily on `/proc/` file system. The `/proc/` file system contains a hierarchy of special files which represent the current state of the kernel, allowing applications and users to peer into the kernel's view of the system. Within the `/proc/` directory, it is possible to find information detailing the system hardware and any processes currently running. In addition, some of the files within the `/proc/` directory tree can be manipulated by users and applications to communicate configuration changes to the kernel.

The `/proc/` directory contains a file called virtual file. It is for this reason that `/proc/` is often referred to as a virtual file system. Most of the time and date settings on virtual files reflect the current time and date, indicative of the fact they are constantly updated. Virtual files such as `/proc/interrupts`, `/proc/meminfo`, `/proc/mounts` and `/proc/partitions` provide an

real-time view of the system's hardware. Others, like the `/proc/filesystems` file and the `/proc/sys/` directory provide system configuration information and interfaces. A process reading the `/proc/pid/dump` file will obtain the information about the contents of the CPU registers and its anonymous memory. The `/proc/pid/mfd/` directory containing information about files mapped into the process's address space. Each virtual memory area is represented by a symbolic link whose name is the area's starting virtual address and whose target is the mapped file. The bulk of this information already exists in `/proc/pid/maps`, but the mfd directory collects it in a useful format and makes it possible for a checkpoint program to be sure it can open the exact same file that the process has mapped. The information gathered from `CRIU` under the `/proc/` virtual file system includes files descriptors information via `/proc/pid/fd` and `/proc/pid/fdinfo`, memory maps via `/proc/pid/maps` and `/proc/pid/map_files/`.

The process dumper does the following steps during the checkpoint stage. The `pid` of a process group leader is obtained from the command line using the `--tree` option. By using this `pid` the dumper walks though `/proc/pid/task/` directory collecting threads and through the `/proc/pid/task/$tid/children` to gathers children recursively. Walking tasks are stopped using the ptrace's `PTRACE_SEIZE` command. At this step `CRIU` reads all the information about collected tasks and writes them to dump files. VMAs areas are parsed from `/proc/pid/smaps` and mapped files are read from `/proc/pid/map_files` links, while file descriptor numbers are read via `/proc/pid/fd`. Core parameters of a task such as registers are being dumped via ptrace interface and parsing `/proc/pid/stat` entry. Then `CRIU` injects a parasite code into a task through the ptrace interface. Using the parasite code `CRIU` is able to gather more information such as credentials and contents of memory. After everything has been dumped `CRIU` uses again ptrace interface dropping out the parasite code and restoring the original code. The Figure 2.30 illustrates the schema described about `CRIU` dump and restore session.

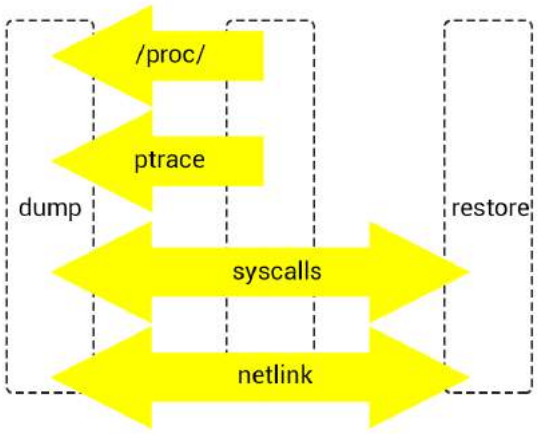

Figure 2.30. Dump and Restore schema in `CRIU`.

The restore procedure is done by `CRIU` transforming itself into the tasks to be restored. `CRIU` needs to resolve shared resources reading the image files and finds out which processes share which resources. Subsequently shared resources are restored by some one process and all the others either inherit one on the 2nd stage like session. After that, `CRIU` calls `fork()` many times to re-created the processes tree needed to be restored. Threads are not restored in this step but later. Later `CRIU` opens files, prepares namespaces, maps (and fills with data) private memory areas, creates sockets, calls `chdir()` and `chroot()`. Timers, credentials, threads and memory mapping location are restored at the end of the procedure. The restore command has several options on how to restore the process tree. Depending on the options passed to `CRIU` the process tree could result different. The default is that `CRIU` just forks the root task and all other tasks, restores the tasks, then waits for it to exit and after it exits itself. The process tree right after restore is shown in the Figure 2.31.

An option –restore-detached makes `CRIU` exit right after restoring the tree thus causing it to get reparented to the init process as depicted in Figure 2.32.

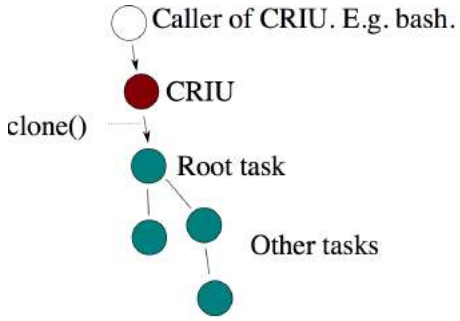

Figure 2.31. Default process tree after restore. (source: [58]).

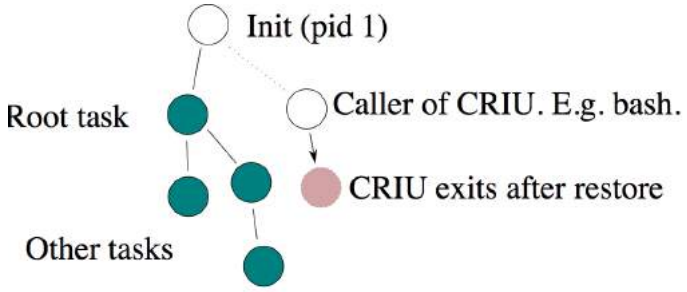

Figure 2.32. Default process tree after restore. (source: [58]).

An option `--restore-sibling` makes `CRIU` create root task as its own sibling and thus as caller's child. After restore is completed, `CRIU` also exits thus leaving the restored tree being a pure child of the caller. It is even possible to pass a custom binary that will be the parent of the root task using the –exec-cmd option.

Examples of dumping and restoring processes

In the previous section has been explained how `CRIU` works internally. In this section it is described how `CRIU` can be used from the command line in order to dump and restore processes. It is important to know that `CRIU` offers several usage scenarios, not only from the CLI but also with C API or RPC. This section covers examples just using the CLI. Let's create a simple bash script that perform a loop such as:

```
#!/bin/sh
while :; do
    sleep 1
    date
done
```

Now run the script. `CRIU` does not support the dump of stopped task, it is needed to open another terminal, get the `pid` of the process and dump it.

```
> criu dump -t $(pidof simple_loop.sh) --shell-job
```

`CRIU` created a list of dumped files that are used for the restore. These files are image files and can be analysed using the `CRIT` tool, as reported in Figure 4.7.

The command for restore the process is:

```
> criu restore -d ./ --shell-job
```

The `pid` of the process will be the same as before of the dump.

```
root@machine:~/simple_loop_folder# ls -l
total 308
  4 -rw-r--r-- 1 root root    965 Jan  7 05:14 core-1866.img
  4 -rw-r--r-- 1 root root    958 Jan  7 05:14 core-1948.img
  4 -rw-r--r-- 1 root root     56 Jan  7 05:14 fdinfo-2.img
  4 -rw-r--r-- 1 root root     44 Jan  7 05:14 fdinfo-3.img
  4 -rw-r--r-- 1 root root     18 Jan  7 05:14 fs-1866.img
  4 -rw-r--r-- 1 root root     18 Jan  7 05:14 fs-1948.img
  4 -rw-r--r-- 1 root root     32 Jan  7 05:14 ids-1866.img
  4 -rw-r--r-- 1 root root     32 Jan  7 05:14 ids-1948.img
  4 -rw-r--r-- 1 root root     38 Jan  7 05:14 inventory.img
  4 -rw-r--r-- 1 root root    901 Jan  7 05:14 mm-1866.img
  4 -rw-r--r-- 1 root root   1237 Jan  7 05:14 mm-1948.img
  4 -rw-r--r-- 1 root root    168 Jan  7 05:14 pagemap-1866.img
  4 -rw-r--r-- 1 root root    131 Jan  7 05:14 pagemap-1948.img
112 -rw-r--r-- 1 root root 114688 Jan  7 05:14 pages-1.img
116 -rw-r--r-- 1 root root 118784 Jan  7 05:14 pages-2.img
  4 -rw-r--r-- 1 root root     45 Jan  7 05:14 pstree.img
  4 -rw-r--r-- 1 root root   1202 Jan  7 05:14 reg-files.img
  4 -rw-r--r-- 1 root root    836 Jan  7 05:14 sigacts-1866.img
  4 -rw-r--r-- 1 root root    752 Jan  7 05:14 sigacts-1948.img
  4 -rw-r--r-- 1 root root     35 Jan  7 05:14 stats-dump
  4 -rw-r--r-- 1 root root     32 Jan  7 05:14 tty.img
  4 -rw-r--r-- 1 root root    178 Jan  7 05:14 tty-info.img
```

Figure 2.33. `CRIU` output files.

CRIT tool

One of the most important tool used in this research is the tool called `CRIT`. `CRIT` is a command line tool used for converting core image file to JSON format and vice-versa. This tool substitutes the old `CRIU show` command.

For example, referring to the list generated in the previous example, the `inventory.img` file contains information reported in the listing 2.34.

For encoding text file in JSON format it is needed to use this command:

```
> crit encode -i inventory.json -o inventory.img
```

More complex examples

When dumping a process sub-tree, `CRIU` checks that the resulting image is consistent and self-contained, meaning if an object A references another object B, and A goes into dump, then B should be dumped as well. For example, if there is a pipe between *process_E* and *process_G* `CRIU` needs to dump the entire tree starting from *process_D* because of the object between processes, as represented in Figure 4.14. For avoiding error it is important to dump the entire tree starting from a node in order to have all objects self-contained.

The same procedure is required for unix sockets. If there is a socket (A) that is connected to another socket (B), and criu dumps socket A, because it is opened by a process it dumps, it must also dump socket B and the task who owns it. Possibly socket B is dumped some time later in the dumping process, but it must be dumped. As reported in [58] this restriction is in place to make sure that upon restore applications will continue to work instead of receiving `SIGPIPE` signals due

```
> crit decode -i inventory.img --pretty
{
    "magic": "INVENTORY",
    "entries": [
        {
            "img_version": 2,
            "fdinfo_per_id": true,
            "root_ids": {
                "vm_id": 1,
                "files_id": 1,
                "fs_id": 1,
                "sighand_id": 1,
                "pid_ns_id": 1,
                "net_ns_id": 2,
                "ipc_ns_id": 3,
                "uts_ns_id": 4,
                "mnt_ns_id": 5,
                "user_ns_id": 6
            },
            "ns_per_id": true,
            "root_cg_set": 1
        }
    ]
}
```

Figure 2.34. `CRIT` usage.

```
init ---process_A---process_B
        |
        |---process_C---process_D---process_E
        |                          |
        |---process_F              |---process_G
        |
        |---process_H
```

Figure 2.35. Process tree self contained.

to half-closed pipes or sockets. If during the dump `CRIU` finds external sockets the user will receive an "ERROR: external socket is used" meaning `CRIU` detected a unix socket connected to another socket which is not being dumped.

However, sometimes it is possible to dump and successfully restore only one end of a unix socket pair. One particular example is the datagram sockets with on-way connection from the client to the server. These connections are thus uni-directional. In this case it is possible to dump a program with the client-side socket and on restore the socket needs to be reconnected back to the original server.

This is when `--external unix[:inode]` option should be used during the dump and the restore procedure. For `CRIU` dump, this option enables dumping datagram unix sockets with additional information about that other ("external") socket it is connected to. For `CRIU` restore, this option asks `CRIU` to re-connect such sockets back.

Named unix sockets with stream/seqpacket options cannot be dumped and restored, as once one end is dumped, the other end will see `EOF` on the socket closing it. Unnamed unix sockets created with `socketpair()` system call can be dumped and restored. On restore, the server should create a new pair and call `CRIU` restore asking it to inherit one.

```
> criu dump -D images -o dump.log -v4 --external unix[11890815] -t 16528
> criu restore -d -D images -o restore.log --pidfile restore.pid -v4 -x
    --inherit-fd fd[3]:socket:[11890815]
```

CRIU alternatives

As reported in the previous section, `CRIU` is not the unique choice as Checkpoint/Restore tool. Several alternatives are possible among which `BLCR`, `DMTCP`, `OpenVZ`, `PinLIT` and others less known.

Berkeley Lab Checkpoint/Restart (BLCR) is a kernel module that allows you to save a process to a file and restore the process from the file. This file is called a context file. A context file is similar to a core file, but a context file holds enough information to continue running the process. A context file can be created at any point in a process's execution. The process may be resumed from that point at a later time, or even on a different workstation. BLCR does not support checkpointing of a process group yet. To restart from a context file, the PID of the original process must NOT be in use.

To restart from a context file, the original executables and shared libraries used must exists and contents remain the same. Berkeley Lab Checkpoint/Restart (BLCR) is a part of the Scalable Systems Software Suite, developed by the Future Technologies Group at Lawrence Berkeley National Lab under SciDAC funding from the United States Department of Energy. BLCR is implemented as a GPL-licensed loadable kernel module for Linux 2.4.x and 2.6.x kernels on the x86, x86_64, PPC/PPC64, ARM architectures, and a small LGPL-licensed library.

Another available tool is called `DMTCP` (Distributed MultiThreaded CheckPointing) project has been busy since about 2.6.9. It is able to checkpoint groups of processes connected by sockets - even across different machines - and it requires no changes to the kernel at all. `DMTCP` implements Checkpoint/Restore of a process on a library level. This means, that if you want to C/R some application you should launch one with `DMTCP` library dynamically linked from the beginning. When launched like this, the `DMTCP` library intercepts a certain amount of library calls from the application, builds a shadow data-base of information about process internals and then forwards the request down to glibc/kernel. The information gathered is to be used to create an image of the application. With this approach, one can only dump applications known to run successfully with the `DMTCP` libraries. Another implication of this approach is potential performance issues that arise due to proxying of requests. Restoration of process set is also tricky, as it frequently requires restoring an object with the predefined ID and kernel is known to provide no APIs for several of them. For example, kernel cannot fork a process with the desired PID. To address that, `DMTCP` fools a process by intercepting the getpid() library call and providing fake PID value to the application. Such behaviour is very dangerous, as application might see wrong files in the /proc filesystem if it will try to access one via its PID. `CRIU`, on the other hand, doesn't require any libraries to be pre-loaded. It will checkpoint and restore any arbitrary application, as long as kernel provides all needed facilities. Kernel support for some of `CRIU` features were added recently, essentially meaning that a recent kernel version might be required. `DMTCP` is available with the openSUSE, Debian Testing, and Ubuntu distributions. A detailed table with all differences between tools is reported in [59].

Chapter 3

State of the art

3.1 Memory forensics in investigation field

The advancement in the use of internet and technologies leads to an increase of cyber attacks. Digital forensic is adopted for acquiring malicious evidences found in system which can be presented to the court. Number of crimes committed against an internet and malware attacks over the digital devices have increased. The recovered information can be used to prosecute a criminal. The goal of digital forensics is to execute an accurate and depth investigation while keeping a chain of evidence to find out exactly what can be found on a computing device. As mentioned in [60] there is a model typically used by forensics analyst, it is composed by a general set of procedures: devices have to be isolated, in order to make sure they cannot be infected, investigators make a digital copy of the device's storage media and takes also the dump of physical memory. A variety of scientifically derived and proven methods and techniques are used by analysts in order to obtain the collection, validation, identification, interpretation and presentation of digital evidence derived from digital sources. Therefore, it is possible to distinguish four general steps: acquiring, authenticating, analysing and reporting. The acquiring phase consists in creating the bit-by-bit copy of the physical memory and storage devices. Authenticate ensures that the data used for analysis is exactly equals to the one taken in acquiring phase. It is possible to authenticate data using checksums. Analysis can be performed using different tools, it is possible to combine various forensics techniques including static analysis, live interrogation and memory image analysis. The fourth phase is called reporting, when the actual scenario is reconstructed.

3.1.1 Digital evidences

Carrier and Spafford in [61] defined what is a digital evidence. They stated that a forensics evidence is data which supports or refutes a hypothesis about digital events or the state of digital data. This definition includes evidences which can not be presented into a court of law, but may have investigative value. This definition is in agreement to Nikkel definition in [62]. In [5] it is explained which are the characteristics of a digital evidence, it is fragile, therefore it can be easily altered, damaged or destroyed by improper examination. They are data of investigative value stored or transmitted by device. In its natural state, they can not be known by the content of devices, evidences are found through a complex examination process, and it is very hard to keep an evidence in its original state. Precautions have to be taken to manage and preserve evidences, reports are also required to explain investigation process and its limitations. Digital evidence is stored in data storage device, and its content can be understood through printing, playing, or execution. In addition, the digital evidence has the following features: easy to modify and copy, hard to understand the content directly without the conversion process and not easy to retains the original state [63]. To conclude, digital objects are collection of data, them state changes when they are involved in a digital event, if an event violates some policy or rule, then it becomes a digital incident, a digital evidence is a digital object which contains reliable informations useful to supports or refutes a theory about an incident.

3.1.2 Forensics investigation process

In the first DFRWS, 2001, a model of forensic process was created, it was composed by some agreed steps, and it was discussed by a civilian, military and digital forensics science audience. In the years, many process model were proposed by researchers and in 2006, the National Institute of Standards and Technology (NIST) defined the basic forensic process with following phases:

- Collection: data are identified, recorded and acquired.
- Examination: data are forensically processed, evidences become visible, interesting data are extracted.
- Analysis: results of examination are analysed with a prepared strategy, which includes many tools, methods and techniques.
- Reporting: it is the most important phase in order to guarantee found evidences could be used. In this phase results have to be reported, performed actions have to be described and explained.

As highlighted by [5], since the first DFRWS in 2001, the need for a standard model has been understood and a lot of different models were presented in the years, in 2002 Reith et al. proposed an enhancement of DFRWS model, they was inspired by traditional forensic practised by law enforcement. This model is composed by nine phases, in particular some preparation phases precede the four ones explained before. In these new phases, the idea is to prepare a strategy, in order to perform better the following steps, in particular tools, methodologies and procedures are defined. Carrier [64], in 2003, presented his model where tools abstraction layers have been introduced. An year later he introduced, with Spafford [6], an event based investigation framework where the abstraction layer concept is extended to other sections of a digital investigation. In 2005 Ruibin, Yun and Gaertner identified the importance of computer intelligence technology to the computer forensics. They explained that computer intelligence should offer more assistance in the investigation process. A particular concept was introduced by the authors, the notion of Case-Relevance in order to describe the distinctions between computer security and forensics, also defining degrees of relevance. Beebe and Clarke [65] state the need for a new framework for digital forensic process and divide it into six phases and propose a two tier hierarchical objectives framework. Perumal proposed, in 2009, a new model that clearly defines that the investigation process will lead into a better prosecution, focusing on fragile evidence. In order to do that it is important to include in the model also live interrogation and static data acquisition. Wang et al [66] presented in 2009, a model of computer live forensics based on physical memory analysis, it is composed by four stages, and it focuses on give credibility to the live forensics minimizing the impact on the system and enabling validation methods for the evidences. In 2011, Agawal et al. proposed a model for helping digital forensic experts for setting up appropriate policies and procedures. The model focuses on investigation cases of computer frauds and cyber-crimes. In the same year, Ademu et al. developed a new model for the digital forensic investigation process. It was described with a four tier iterative approach. For each tier, different rules was defined, in order to guarantee a systematic approach to digital investigation process.

Issues of memory forensics

The main goal to be reached through a well defined investigation model is to solve many credibility issues of live forensics. Issues can be identified as: authenticity, consistency, repeatability and fault-tolerance. Authenticity, it is important to verify if physical memory region acquired by tools is identical with real physical data. For example, data can be altered by trojan or rootkits. Integrity check if the raw physical memory is completely imaged. Consistency ensures the possibility to use results of investigation in a court of law, in a model it is important to define steps for granting consistency. Repeatability allows to investigators to repeat the analysis several times, the model should separate the phases of the forensic process and make clear the elements in each phase. Live forensics methods should allow fault-tolerant, because it should not be interrupted by missing of some data, or if part of the data was tampered.

A model for the investigation process

In [6] it is presented an example of a model for investigation process, it is explained in details, and it is a generic structure of a typical model. Carrier model is composed by three different macro phases, each of them is related to a different moment of the investigation.

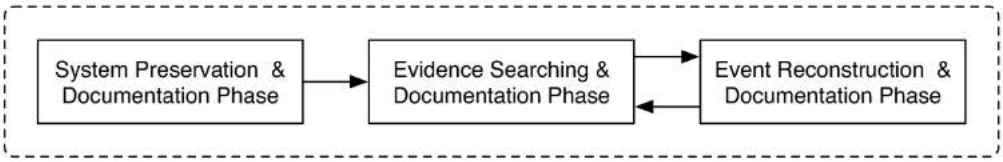

Figure 3.1. Investigation process phases (source: [6]).

The first phase has to preserve the crime scene. It is important to reduce the number of events that may damage or change the state of digital objects, in order to preserve the state of the scene. Different kinds of events could modify an evidence or destroy it. As outlined in the background chapter, in memory image analysis techniques, the choice of an acquisition method is very important in order to guarantee to do not modify the memory state. Every step in the first phase has the goal to preserve the crime scene state, but it is also important to document the state of it. The documentation gives the possibility to later refer to the original state of the scene, permitting verifying if anything is modified or to proving that a certain evidence existed in the crime scene. At the beginning, if it is possible, the physical memory has to be dumped with the chosen acquisition method, later the system is powered off and mirror copies of all the storage disks are made. These steps allow creating an exact copy of the system in a lab. Hashes of raw images are calculated in order to check later if changes occur, therefore if images are kept safe, and them integrity is granted, no other steps are required. Another approach makes use of live response techniques, for example if the victim system is a critical server which cannot be turned. As mentioned before, with live techniques, all useful data are collected when system is running and many preservation steps cannot be taken. Unfortunately, not having a complete image of the system may allow false conclusions to be made because the contradicting evidence does not exist.

When steps for granting the preservation are taken, investigators can move to the searching phase, searching for evidences. The goal of this phase is to recognize the digital objects which may contain information about the incident. As highlighted by [6], researching process is an iterative procedure, and it is possible to identify four different steps:

- Define a target.
- Extract data using selected tools.
- Compare extracted data with defined target.
- Update knowledge and definition of new targets.

Target definition is a very challenge process. It is different for each investigation, investigators are helped by their experience, they have to determine what kind of evidence is expected to be found. The searching phase is iterative and targets are also defined using evidences already found. Data extraction is the step where forensic tools are used, it is conduct using interpretation or abstraction layers. A digital evidence is often very difficult to be found in raw data, therefore they are translated through one or more layers of abstraction using forensic tools until they can be understood, each level provides a different amount of data. Layers used in tools could introduce errors: tool implementation error introduced by tool development and abstraction error which is introduced by the simplifications used to generate the tool. Investigators have than to understand if gathered data are evidences or not, comparing them with determined target. When evidences are collected, analysts have to document and preserve them in order to satisfy legal requirements for the court.

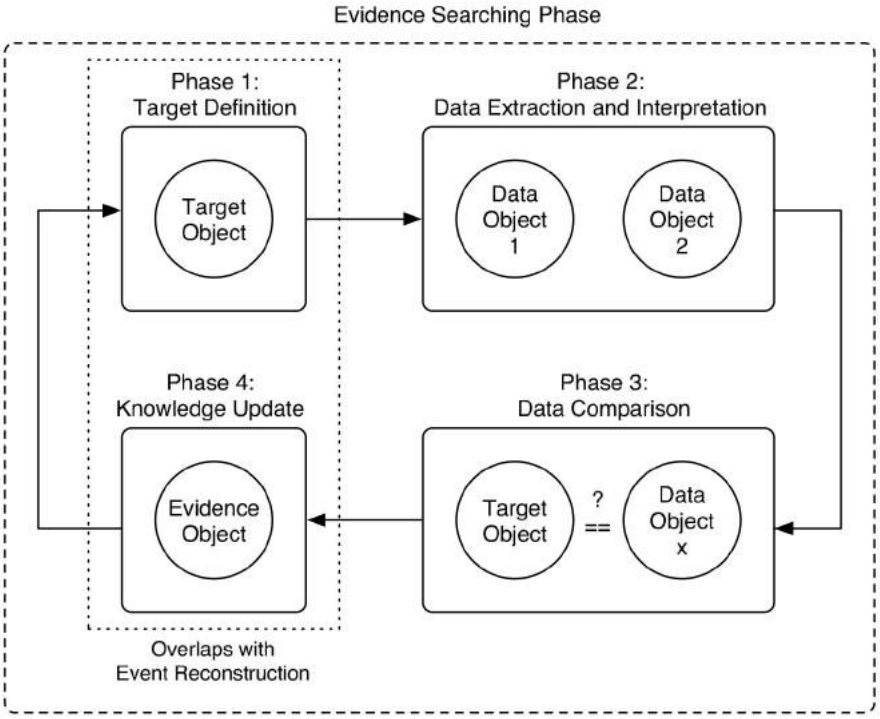

Figure 3.2. The searching phase (source: [6]).

The third stage is the digital event reconstruction and documentation, in this phase gathered evidences have to be used for understanding which events caused the incident. Firstly, each of the digital evidence objects have to be examined and identified, sometimes it happens in the search process, when an evidence is found, but with this phase additional analysis are performed.

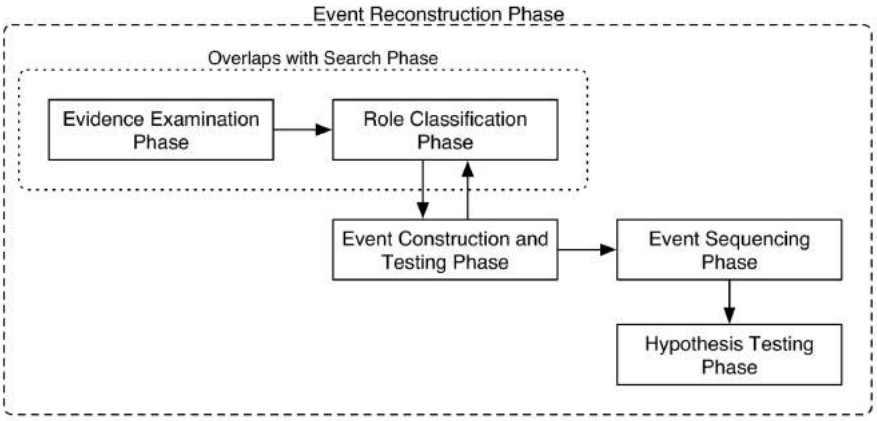

Figure 3.3. The documentation and reconstruction phase (source: [6]).

Then digital objects are classified basing on what roles the object could have played in the incident. When objects are examined and classified, it is important to reconstruct the events, if any object is missing, it must be searched again with the second phase. In event reconstruction, the contents of digital evidences are analysed in order to set the time windows within certain activities might have occurred. Reconstructed events have to be tested, in order to guarantee to have all evidences needed to prove that each event occurred. Event chain is then reconstructed in order to understand the incident and to make hypotheses. Event chain has to be documented by describing causes and effects. An explanation of what is missing is also required. Results can be used by investigators to come to a final conclusion.

3.2 Analysis techniques

In previous section, the investigation process was explained, in this section it is possible to focus to different analysis strategies. In particular, techniques that will be described, have objectives similar to ours, because some of them have as objective the restoration of previous states, the others belong to the investigative forensic area. In literature, there are not techniques directly comparable with this work, because none of them is able to restore a process in a previous state and make it interactive for the analyst.

3.2.1 Replicating processes state using volatile memory forensics

Traditional digital forensics can be divided into two main branches: static analysis of data preserved on permanent storage media and live analysis which uses running system to obtain volatile data for understanding what is going on. As mentioned in background chapter, sampling running system could irreversibly change its state making collected evidence invalid. In paper [3], it is explained how is possible to combine static and live analysis techniques in order to perform a good forensic investigation. Virtualization is used to bring static data to life. A physical memory dump is used with the virtual machine created to provide better picture of the system bringing it close to the state it was in when it was seized. Investigator can have an interactive session with virtual machine which can be analysed several times, without violating evidence integrity. This idea is something similar to live response but in repeatable manner. Methods presented in the paper are proof of concept and are not sufficiently robust to be considered ready for use is forensic analysis. The idea of the authors is to provide a technique, able to resolve limitations of standard analysis strategies. Static analysis, indeed, has many limitations, and in particular it can not provide a complete picture of events because the system has to be shutdown. On the other side, live analysis is not repeatable, and it can possibly damage evidences. The two used strategies, combined in this project are static analysis with virtualization and volatile memory analysis. Usage of virtual machines was proposed for the analysis phase of a digital forensic investigation, the idea is to create a virtual machine from an image to be investigated. The problem is that the virtual machine simulates only some basic hardware components and the image cannot be immediately booted in the virtual machine, therefore, many changes to the original environment are required to enable the image to boot in the VM environment. One solution was proposed with the Live View tool, which is able to create a VMware machine starting from an image of a physical disk. This allow analyst to boot up the system and gain a user-level perspective of the environment, moreover the system is interactive, and all changes do not modify the underlying image. Volatile memory analysis is a well-known technique, the strength-points of this technology are the repeatability of all the operations and the possibility to analyse also hidden or terminated processes.

The proposed model by authors is a combination of static and live analysis, in order to provide a more insight into current state of the system, which can be better examined, understanding the events that led to an incident. Using Live View, an image of a physical disk is booted in VMware, the memory dump acquired before system shutdown is used to restore the booted system to the state it was when dump was collected. The described process is not entirely automatic, a manual start of all programs is required. It is possible because the memory dump contains data on the processes that were running on the system and it contains also times when the processes were started. Hidden programs like rootkits that would not be visible in live analysis are included. The same operation have to be performed for open files and network ports, which have to be open following informations gathered with the analysis of the memory dump. The described procedure permits performing an analysis with the goal to understand all possible scenarios, it is also repeatable and can be presented in a court of law. This approach is very similar to live analysis, indeed, analyst can put the system in hibernation, take an image of the hard-disk and then reboot the system. His state will be almost the same with no need of additional tools.

Authors though also to test cases in order to have a proof of concept. System to be investigated was Windows XP. The hard disk partition and RAM are small compared to the standard of those years. They were set small to enable a faster manipulation but big enough to show the procedure. Some sample programs were stared on the system, a file was opened for editing in Notepad and

Netcat tool was started from a command prompt. TrueCrypt was used to create a new encrypted volume, one file on it was opened for editing in Notepad. The system was then hibernated and an image of the disk was acquired and stored with his MD5 checksum in order to later check his integrity. The second phase of the test consisted in creating the virtual machine. Live View was started giving to it some parameters related to original system environment, for example the amount of ram or the installed operating system. In the second stage hibernation files were also extracted in order to analyse them using Volatility framework. In the third phase, VMware was started, it realized that the system was hibernated and tried to wake it up. The restoration process failed as expected due to hardware difference between original system and virtual machine, therefore the operating system deleted hibernation files and started a new Windows session. All previously started programs were not running and all files were closed. Using Volatility it was possible to extract various data from hibernation files, for example the list of running processes. The list was then compared with the list of running processes in virtual machine. The difference consisted in the processes which were manually started on the investigated system. All needed information was retrieved through volatility framework except for open connections, it has been expected because for the hibernation process all connections have to be closed. Processes found in the volatile memory image, were then started in the virtual machine, opened files found in memory were open with the same applications in the virtual machine. After that procedure the virtual machine was brought close to the state the original system was in before hibernation. At the end of the test, virtual machine was stopped, and original files integrity was checked, all files were not modified meaning that evidences were not altered. This concept is very important in order to guarantee the possibility to repeat the analysis and verify it.

In conclusion, presented work has some strength points, it is able to provide something which can be used from investigators for interacting with a system very similar to the investigated one. Static analysis can still be performed, and data available during a live analysis can be simply extracted. The two major weakness are, the risk to change original system state with the hibernation procedure and the leak of the possibility to interact with interesting processes in their original state.

3.2.2 Rendering data extracted from a memory dump

Various techniques propose to analyse a memory dump, in order to understand what a user was doing on the system at a certain time. In particular there are some strategies which propose to render in different ways data structures in the memory dump.

Extraction of screen content from a windows memory dump

In 2009 Klitz et al [8] proposed a technique for acquiring and rendering Windows graphic content from a physical memory dump. They simply explained the strategy without very practical informations. The chosen example was the gathering of content displayed in the Internet Explorer web browser.

The first described phase is the data gathering, a physical memory dump is collected with the chosen technique. It is a very important stage, gathered data could be inconsistent because content change during the acquisition process. After data gathering the second stage is the dumping of process address space. In this phase a forensic tool have to be used in order to collect needed informations. For example in the paper the Volatility Framework with LinuxDumpProcMap plugin are mentioned. Using Volatility it is possible to extract the address space of a single process in order to analyse it. The last phase of the procedure is the rendering of data. For this step the Irfan View tool is used, the tool takes a raw file in input and, with appropriate parameters it is able to render it as a picture. The described strategy is a very simple technique, and it can not be effectively used in a real investigative scenario, but it is an example of how is possible to use data in memory dump for generating graphical outputs.

Interpreting and rendering data structures in memory

In 2014, Brendan Saltaformaggio et al [9] presented a new tool able to render forensic data using application logic reuse. Many techniques use signature-based scanning of memory images in order to gather interesting data structure for investigators. Often, researches are not able to interpret collected data, even with a good understanding of data structure semantics. Sometimes investigators know exactly what a group of bytes is, but they are not able to render the object. In these papers it is presented a technique based on data structure content reverse engineering, it is called DSCRETE, a system able to automatically interpret and render extracted data, in order to generate a human-understandable output. It does not use structures signature scanning, it is a scanner and renderer tools which uses application logic. The presented tool is able to recover a variety of application data with a very high accuracy, it could be very useful for investigators which otherwise can not really understand raw data. For the tool development, authors considered some memory image analysis guidelines, for example they tried to build a system as minimally invasive as possible, moreover they wanted to develop a tool able to recreate the system previously observable state starting from a memory dump. DSCRETE is based on the following statement: the application in which a data structure is defined, usually contains interpretation and rendering logic to generate human-understandable output for that data. The key idea of authors was to reuse existing data structure interpretation and rendering logic in the original application binary. At the high-level a generic application processes into its internal logic data structures, when an output have to be produced, the application is able to reconstruct a well formatted output from internal structures. If an investigator found in a memory dump, data structures of the selected application, DSCRETE would be used to identify and automatically reuse the program render logic. The rendering component is a function which takes in input a data structure and produce a human readable output. The presented tool, firstly identify this component, and then reuse it to create a scanner tool to identify all instances of subject data in memory.

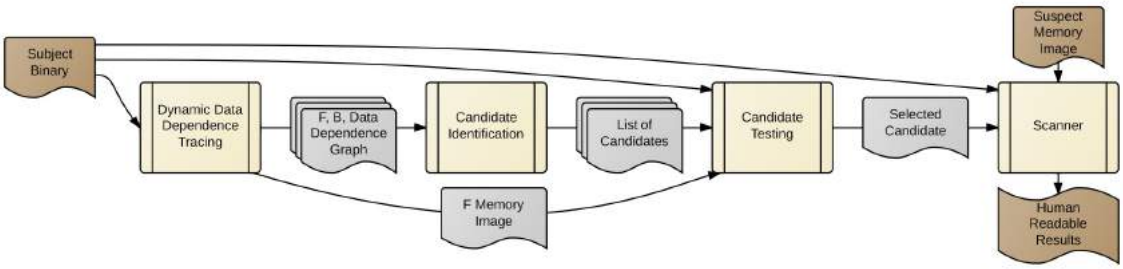

Figure 3.4. Investigation process phases (source: [9]).

In figure 3.4 it is described the DSCRETE workflow. The first input is a binary application for which an investigator wants to recover application data from a memory image. To avoid compatibility issues, this binary should be the same as the one used in the victim machine. In the second phase selected binary is executed in order to identify the part of code responsible to for generating output starting from internal data, it is called the tracing phase. The next stage is the identification of the right rendering component, a graph is used to formulate a list of possible candidates, then each of them is tested with a tester component which is able to verify if it could be a good memory scanner. Then DSCRETE packages identified logic in a context-free memory scanner which could be used by investigators in future analysis involving the same application. All previous phases are a one-time procedure which is internal to the system and have not to be performed directly by analysts. Authors made various assumptions in their work, firstly they assumed that the subject binary can be executed before the analysis, in order to recreate the execution environment. Moreover, the address space layout randomization have to be turned off when producing DSCRETE memory scanners, this assumption is required for granting that multiple execution of the same binary, always produce the same address space layout. All kernel data structures have to be intact, in order to be able to isolate only pages related to the subject application.

The proposed workflow is composed by four different stages. The first is the dynamic data

dependence tracing, in this phase the tool collects all dynamic dependence from the subject application binary, all library functions and system calls invoked are recorded, as also their input parameters. Structures are created in order to contain collected data, then analysts have to express their interest marking desired structures. In order to perform described operations the binary have to be executed and the tool have to interact with it. A dependence graph is built using traces gathered in this phase. In the first stage also heap and stack of the application are collected at the invocation of each external library function or system call. When tracing phase is completed, functional closure have to be identified, closure points are instructions in the dependence graph which directly handles a pointer to forensically interesting data and any instruction which reads a field of data structure is dependent on the closure point. These points are interesting because changing the pointer to the data structure is possible to change the output. The identification of closure points, is a trivial operation, without the source code it is not possible to know the exact closure point, there may be many candidates in the same program. In the third stage, the scanner entry point have to be found, each closure point candidate have to be tested using a particular component of described tool. It takes as inputs the known end point of the scanner, the memory image taken when rendering function was executed, the list of candidates and the subject binary. When the closure point is identified, it is possible to move to the fourth phase, the memory image scanning. The tool operates in this stage as a scanning and rendering tools, it is quite similar to the one used for testing candidates, and it takes in input the chosen entry point and exit point of the printing function, the subject binary, and the memory image which have to be analysed. The binary is executed and when the chosen entry point is reached, the scanner pauses the application. For each address in the memory image, the scanner will fork an identical child and assign as data pointer the next address in the memory image. In essence, the scanner is executing the rendering function with a pointer to each byte of the suspect memory image. At the end of the execution, outputs of each rendering function can be inspected by investigators. Various experiments conducted by authors shown that DSCRETE is able to successfully identify rendering functions in many real-world applications. The scanner+renderer tools can identify and render various types of data structure contents from memory images with high accuracy.

Screen content rendering with mobile memory forensics

Another paper which presents a strategy able to render graphical content starting from a memory dump, was presented in 2016 by Saltaformaggio et al [11]. Nowadays smartphone are often involved in cyber investigations, in them is possible to find many types of evidence, an app's prior screen displays may be the most intuitive for an investigator, revealing the intent, targets, actions, and other contextual evidence of a crime. The presented tool was called RetroScope, it is able to recover multiple previous screen of an Android app in the same order they were displayed. It is different from others techniques because does not require app details likes internal structures definition or rendering logic, moreover with this strategy enables spatial-temporal forensics by revealing what the app displayed in a certain time interval and not in a single time instance. A similar technique was presented by the same team, it was called GUITAR [10] and permitted to recover only the last displayed screen, it is very less useful because for example if a suspect has logged out from an application before the memory dump acquisition, the last displayed screen is the logout page, which is not very informative. RetroScope achieves near perfect accuracy in terms of reconstructed screen display and their temporal order.

GUI data structures created for previous screens get almost overwritten when a new screen is rendered. This is exactly why GUITAR is unable to reconstruct previous screens, and this is why authors try to use a completely different approach for designing RetroScope. Although the GUI data structures dissolve quickly, the actual internal data displayed on those screens remain in memory for more time. Following the traditional methodologies of searching and rendering instances of those app data, rendering logic and structures definitions would be required breaking the app-agnostic property. Therefore, authors proposed to redirect Android draw mechanism to those old app data, this would cause the previous screens to be rebuilt and rendered. In Android, when an app is brought to foreground, its entire screen must be redrawn from scratch according to layout. RetroScope was designed to trigger the execution of an app screen-drawing code from a memory image.

The presented tool is fully automated, it requires in input the memory image from the android device and it will create as many previous screens as possible. Authors took advantages of drawing mechanism of Android framework. The main activity of the presented tool is the selective reanimation, the target application runtime environment is setted up. A new process, the symbiont app, is started in android emulator and RetroScope recreates the original address space from memory dump. In particular, the tool load data and code segment at their original addresses allowing pointers to these segments to be valid. Reloading segments is not enough for the reanimation, the tool searches for global java runtime objects in memory image. Gathered data are then copied in the symbiont app address space which contains at this time two full application. Then the tool has to mark the draw functions in order to receive the `redraw` command from the operating system, RetroScope traverses the target app classes searching for top level views which inherit from Android View Class and are not drawn inside any other views. Each view class have to implement a draw method, when the tool encounter one of them mark it as a entry point for the reanimation. In order to receive the `redraw` command, the symbiont app current view is invalidated by the tool and when the `redraw` command is received, RetroScope replace the view with another one from the target app. These last steps can be repeated in order to regenerate all possible screens. Finally, redrawn screens can be reordered to match the temporal order in which they were displayed, as showed in 3.5. This can be done comparing View ID fields in the target app views recovered from the memory image.

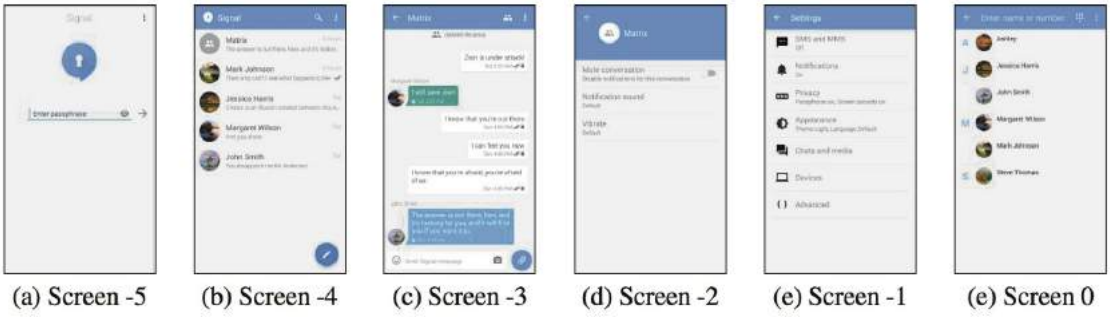

Figure 3.5. Investigation process phases (source: [11]).

RetroScope is a very powerful technique, it is able to recover screen contents with a very high temporal depth. It permits to investigators to really understand what a user was doing with the system, and if it is involved in an incident under investigation. The described strategy is difficult applicable to an environment different from Android which have two fundamental features: view class with draw function for each app which have to draw content on screen, the drawing function is isolated and can not perform blocking operations.

3.2.3 Extracting user input

During a forensic investigation, it is very important to understand what a user was doing on the analysed system. Many techniques tried to render screen content, or other human-understandable output, in [67] authors proposed to use user input informations for investigative purposes.

The paper investigates the amount of user input that can be recovered from the volatile memory of a Windows system. Authors proposed also a temporal and functional analysis end event reconstruction of user input activities such as what the user is typing on the application. It is forensically interesting to retrieve user input informations, firstly investigators can understand how and why a suspect was using an application, moreover reconstructing the event of user input can reveal a good amount of relevant information which can be documented and used as evidence in a court of law. Authors stated also that with gathered informations is possible to better know the user, who he is, how he interacted with the application and when his last access was on it.

The described procedure is very simple, firstly the physical memory is dumped. The image is then analysed and processed using searching techniques and the natural language processing.

A program analyses the dump and automatically extract strings for evidence searching. During the second step, a pattern evidence matching is performed using commonly used words. Idea of the authors was to determine what data in a memory dump could be relevant for forensic investigations. The user input is really important for analysts, it can be reconstructed and used during the investigation in order to answer to many forensic questions. Presented strategy is not a very practical work, but it is an example of what actually researches are doing in forensic field for better understand how a system or a user is involved in a digital investigation.

Chapter 4

Process resurrection

4.1 Objectives

Before explaining how the resurrection process works, it is important to understand which are the objectives of this work. It could belong to the memory forensic area, and in particular to the investigation field. The main and simpler objective of this thesis is to help analysts to understand what was performed on a system before the physical memory dump. As illustrated in chapter 3, there are many comparable techniques which try to make evidences easy to be collected and understood and others which try to give to investigators the possibility to perform a live analysis in a repeatable manner. The result of this thesis can be used in both scenarios increasing the reliability of the outputs. The main idea is to identify and select one target process from a Linux memory dump, then the process can be extracted and isolated from the memory image and at the end it can be restored in a new machine prepared for the analysis. In the first scenario improvements regard collection and identification of the evidences. Giving the opportunity to resuscitate a process permits to analysts to directly visualise the outputs and to interact with the program. It is really a big improvement for a forensic analysis, it usually requires many forensic experts able to understand data extracted by low level forensic tools, using BackToLife investigators, ideally, have only to decide which is the target process, restore it and perform the analysis at an higher level. Many techniques presented in chapter 3 were able to render data in order to produce human readable output, it is very important for permitting to analysts which are not forensic experts to understand the collected digital evidences. However, none of them was developed for producing an interactive output. An interactive output can be really important in a forensic analysis because it gives to analysts the possibility to perform the investigation directly on a running process. It means that investigators can collect data and directly watch what the user was performing with the investigated process. In the second scenario the results of the thesis permits collecting more reliable data, giving the opportunity to perform a repeatable live analysis on a target process which is restored exactly how it was, before that the memory dump was taken. In chapter 3 a technique which uses static and memory image analysis was presented, it proposes to manually reconstruct the state of the system for collecting the evidences and perform the analysis. With BackToLife the target process will be automatically restored at the desired state and it is ready to be analysed. The presented project can be situated in the investigation process presented in chapter 3 subsection 3.1.2, in particular it can be included in the *Evidence Searching phase*, where target are defined and searched through analysis of digital objects. This can be noticed also by the similarity between the workflow of the second phase of the investigation process and the workflow of the presented model.

In Figure 4.1 an high level model for a forensic resurrection performed by an analyst is shown. After the memory acquisition, and the others operations of the first phases of the investigation model the dump can be analysed by investigators in an analysis machine with many forensic tools. The purpose of this thesis is to resurrect a process, the analyst using many forensic tools, like the Volatility Framework, has to obtain a list of all processes found in the memory image. Then a target process have to be selected, and `BackToLife` can be used to firstly extract the selected

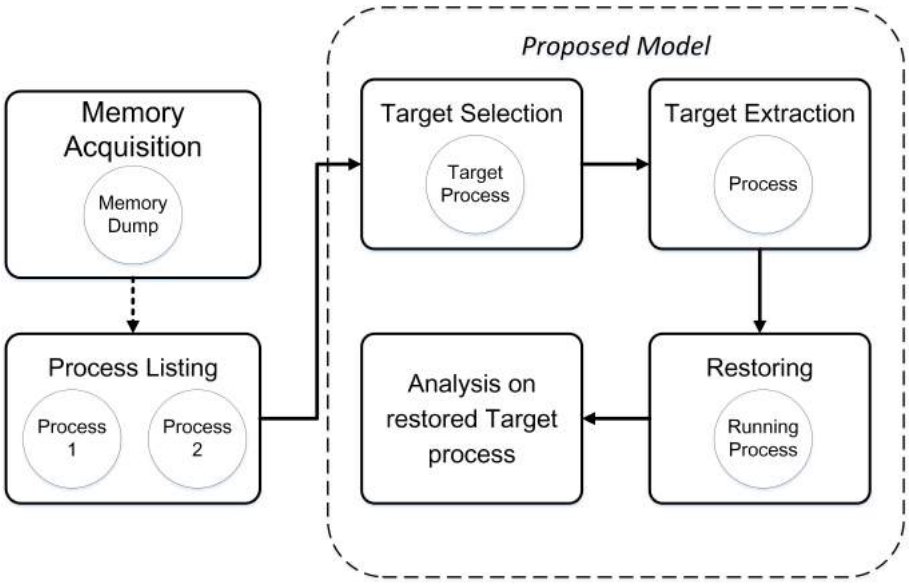

Figure 4.1. BackToLife Workflow for analysts.

process and secondly to restore it in a new machine. Therefore, the main phases of the presented model are:

- Memory acquisition: the memory image can be acquired with the chosen technique by the investigated system. After the acquisition phase, and before the effective analysis, many operations should be performed in order to guarantee integrity and reliability.
- Process listing: with a memory forensic tool the dump have to be analysed in order to obtain the list of available processes. It includes processes which were running, and also terminated processes not overridden in memory and in kernel structures.
- Target selection: the target process have to be identified by analysts. Investigators have to choose an interesting process in order to perform the examination.
- Target Extraction: this is the first phase performed automatically by the presented tool, the target process is identified in memory, as also his data structures in kernel or user space. Then useful data are extracted and moved to the next phase.
- Restoring: Starting from the extracted data the process is completely restored in the new machine, the program state is replicated, the tool try to automatically restore also network connection and all is needed to obtain an exact copy of what is running on target machine before the dump acquisition.
- Analysis on restored process: Investigators, in this phase, can perform any kind of examination of the running process, knowing that it has the same identical state it had before that the memory dump was taken. It can be simply graphical, but also something similar to live analysis can be performed.

This steps sequence can be compared with logical workflow of some technique presented in chapter 3, in particular the concept of target selection and data extraction phases can be found also in other strategies. In particular, in techniques which try to graphically render extracted data, the logical flow is very similar to the `BackToLife` workflow. The last four phases, enclosed in the dashed container in Figure 4.1, can be repeated by analysts many times changing for each cycle the target process.

4.2 Implementation

In this section it will be explained how this thesis proposes to implement the process resurrection. The though strategy can be represented with the schema in Figure 4.2, firstly, starting from the memory dump a forensic tool is used in order to collect needed informations of the target process. In a second moment, taking the advantages of a checkpointing tool the process can be restored at the previous state starting from the collected data. For the purpose of this thesis, two different tools are been chosen: Volatility Framework as memory analysis tool, and the `CRIU` project as checkpointing tool. The strength point of them were explained in section 2.3.2, and in this section, how them can cooperate will be explained.

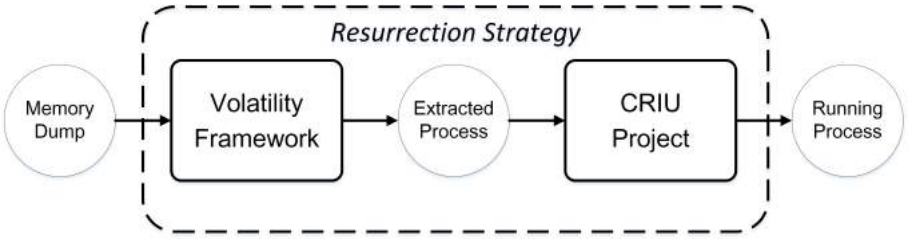

Figure 4.2. Proposed strategy for the resurrection of a process.

The `CRIU` tool is a checkpointing tool, it is able to dump a process with his state in a certain number of files on the file system. Those are binary files, which can be converted in a human readable format using a companion program: `CRIT`. It is able to convert files which contains `CRIU` data structures in `.json` data. `CRIU` can be also used to restore a program starting from the collection of binary files, it can be run by the user and it will automatically start the old process with his identical previous state. Volatility is a framework used in memory forensic fields by many experts, it was developed with the focus on malware analysis and identification on Windows operating systems, but thanks to his developer community it supports now many operating systems and many kinds of analysis. It offers an interface to the memory image helping analysts to collect data. It is implemented in *Python*, and it is a very modular framework, extensible by plugin development.

The main idea of the presented project is to use Volatility for collecting data and to structure them in `json` structures. Then through the `CRIT` tool, data can be formatted as `CRIU` expect, in order to generate a checkpoint of the extracted process. Using `CRIU` it is possible to resurrect the program starting from the generated checkpoint. Therefore, the main steps for process resurrection starting from a memory dump are:

- Volatility analysis: the memory dump can be analysed with volatility in order to collect target informations. All needed data have to be extracted from the memory dump and organized in well defined structures in particulars `json` files.
- File conversion: through the `CRIT` tool, generated files have to be converted in binary files readable by `CRIU`. In this phase a checkpoint is created starting from data extracted by memory image. It contains the target extracted process with all is needed for restoring it at his previous state.
- `CRIU` restoring: The generated checkpoint is used by `CRIU`, which run the process restoring his state in the new machine.

In order to perform the presented workflow there are some missing parts which have to be implemented, from an high-level point of view, what is needed is something able to use Volatility to automatically identify and extract useful data, organizing them in well defined `json` structures. In following sections the implementation phases will be explained from a lower point of view, focusing also on how researchers operated in order to develop the project.

4.3 Plugin design

4.3.1 The analysis model

In this section the analyst is able to understand one of the model used during the development of this research. The model is called DACA (Dump, Analysis, Composition, Automation) and it represents the steps needed for understanding which are missing parts in order to obtain the expected result. Following the model it has been possible to develop in a correct manner scripts and tools for the purpose of the research.

As depicted in the Figure 4.3 the model can be divided into several steps that are explained here below.

1. Starting processes: in this part the researcher chooses a particular process and start the execution of it. The researcher should know the behaviour of the process in order to have some known information making easier the analysis process.

2. Dump of RAM: in this step the dump of the RAM is made obtaining a file that contains all the volatile content of the RAM memory. One important phase before dumping the memory is to pause the target process for some reasons that will be explained in the next section.

3. Dump with `CRIU`: at this point the process should be started again because `CRIU` does not support the dump of a stopped process. Now the analyst as fast as possible should dump the target process generating the list of files which contain memory content of the process. The dump with `CRIU` should be made as near as possible to the dump of the RAM step. The purpose is to have `CRIU` generated values that are quite equal to the values in the dump of the RAM.

4. Analysis and extraction: in this part the researcher analyses the `CRIU` output files. A deep study of theory of that particular part of the operating system is done. Subsequently it is used the Volatility Framework in order to analyse the dump. In particular, the Volshell Plugin is used for moving deeply into the memory of the RAM and for extracting specific memory areas.

5. Composition of data: after that all data used by `CRIU` for the resurrection has been extracted it is needed to compose it in a manner that can be interpreted by the tool. In this part python scripts are made and CRIT tool is used for the generation of files.

6. Development of automation process: in the last part of the analysis model the development of an automation process in done. The automation process is used for automate the extraction of memory areas and the composition of data for restoring the process using `CRIU`.

As depicted in the Figure 4.4 the DACA model is applied cyclically in order to obtain a functional tool that is able to work properly with different kind of processes, starting from simple processes since to socket-based or multi-threading processes.

As a first step the researcher select a target process that would resurrect. The behaviour of the process is known in order to make the analysis phase easier. A good method could be to develop a program in order to know every single instruction of the program. Subsequently the DACA model is applied looking for memory areas and needed information useful for the resurrection. The data are composed and made readable for `CRIU`. Finally tools and plugin are refined and modified basing on results obtained by the application of the DACA model. The model is applied with different kind of processes, always more complex. Starting from a basic counter, to multi-threading processes, socket-based processes and so on. The development of tools has been possible because the tools have been developed and refined for every kind of process that has been analysed for the resurrection.

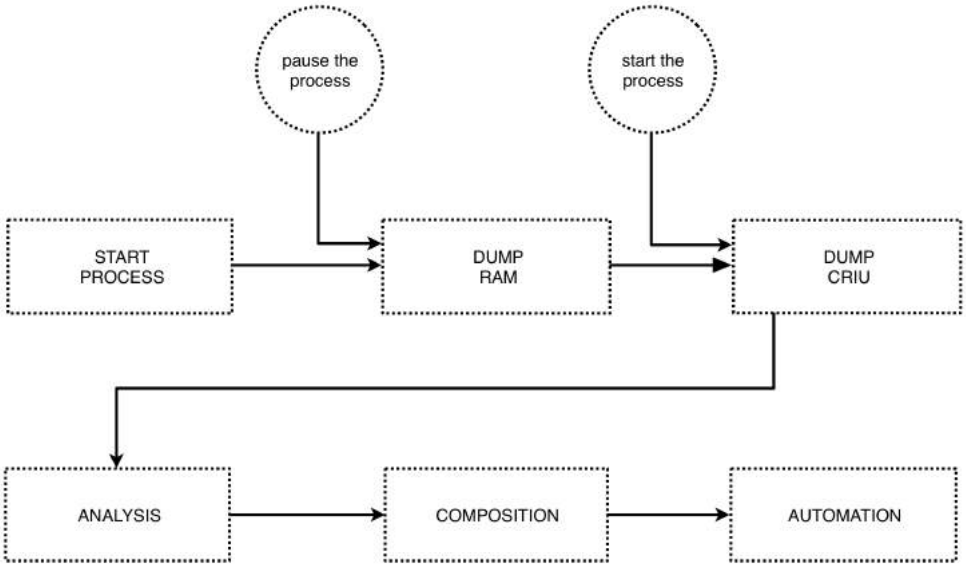

Figure 4.3. The DACA model.

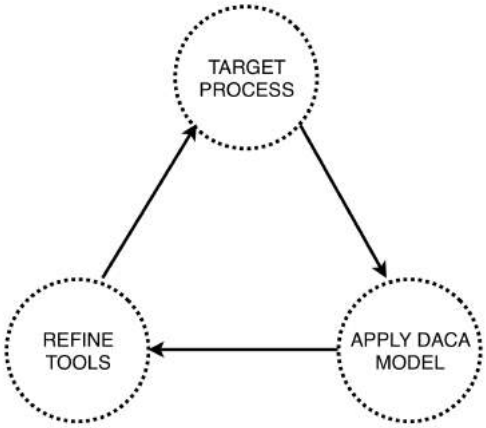

Figure 4.4. The cyclical procedure.

4.3.2 The analyst operations

In this section is explained the scripts and tools that have to be run by the analyst in order to correctly resurrect the target process. Before to analyse the procedure it is important to know what are the scripts and tools that have to be used. The development of tools produced by this research is mainly composed in two parts. The first contribution is a Volatility plugin called `linux_backtolife` that is able to extract necessary data for the resurrection. This data include all memory areas, CPU registers, socket information, files and all information that are linked to the process.

The second part of the development is about a script called `prepareMachine.py` that is needed for generating files and for preparing the environment for the resurrection of the process. The resurrection is not isolated and independent from the machine. The process expects to find some files or libraries under particular path that may be not present in the resurrection environment. The `prepareMachine.py` script is responsible to prepare the environment in order to have the resurrection machine as same as possible to the dumped machine. The Figure 4.5 reports the flow followed by the analyst for correctly have the resurrection of the process. Starting from a physical memory

dump that is made on the target machine, the analyst should launch the `linux_backtolife` plugin using Volatility specifying a particular process that is the target process that the analyst want to resurrect. The plugin generates a list of files as output. After that the `prepareMachine.py` script should be launched using Python in order to modify the files produced by `linux_backtolife` accordingly to the resurrection environment. The `prepareMachine.py` script prepares the environment for hosting the process. The detailed explanation will be discussed in the next sections. Finally the analyst have to launch `CRIU` using the files produced by the previously tools. The process will be resurrected from a physical memory dump on the analyst's machine on a new Terminal window.

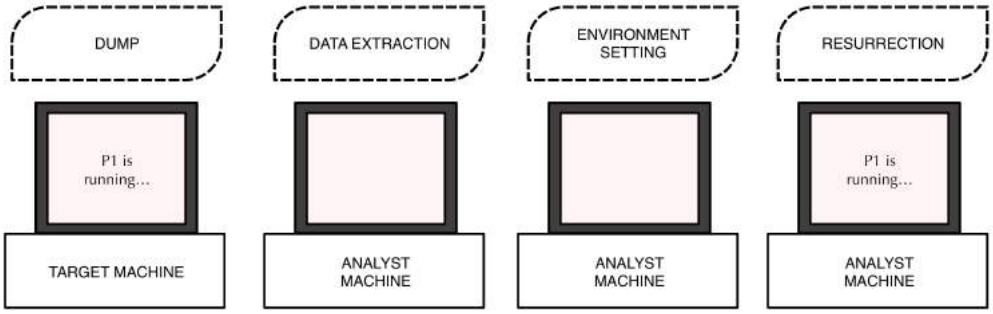

Figure 4.5. The analyst operations.

4.3.3 A basic process

Introduction

In this section it is described step by step how researcher have built tools for obtaining the expected results. In the first part is it described the target process and the application of the DACA model. Subsequently it is deeply analysed every single file produced by `CRIU` and the methodology applied by researchers in order to find and extract the needed informations. The objective is to create files as same as `CRIU` have generated them in order to correctly resurrect the process.

The DACA application

The first step of the DACA model is to select a target process. In this case the program is created by the researchers their-selves. The simplest program that can be developed could be a well-known `Hello_World.c` program. The problem with this kind of program is that it is really difficult to have a dump with this process in execution. The researcher decided for a program that can give an output that can be interpreted during the resurrection phase. Furthermore the program should run for a long time period. The listing 4.6 contains the code for a simple counter. If the tools worked, it would resurrect the process with the same value reached at the time of the dump.

After that the program has been created and compiled it can be launched typing the following commands:

```
> gcc -o counter counter.c
> ./counter
> Counter value:0
......
> Counter value:69985
> Counter value:69986
> ^Z
[1]+  Stopped                 ./counter
```

```
#include<stdio.h>
#include<stdlib.h>

/* A BASIC COUNTER */

int main(int argc, char* argv[])
{
        int count = 0;
        while(1)
        {
                printf("Counter value:%d\n",count);
                count++;
        }
        return 0;
}
```

Figure 4.6. Example of a counter C program.

The program have to be stopped before continuing the procedure for reasons that will be explained later in this section.

The second step of the DACA model is to dump the RAM. For this research it has been decided to use LiME as linux memory extractor. In order to start the dump the researcher launched the `insmod` command because lime is a kernel module.

```
> cd /lime_modules
> insmod lime-4.0.0-kali1-amd64.ko  "path=./counter_backtolife.dump
    format=lime"
```

The third step of the applied model is to dump the process using `CRIU`. Before to start the dump with `CRIU`, as described in the previous section, it is needed to start the process using the `fg` command because `CRIU` does not support the dump of stopped processes. After that the process is in execution the researcher opened a new Terminal window and dumped the process using `CRIU`.

```
>  ps -le | grep counter
0 T     0  1799  1711  0  80   0 -  1018 -      pts/0    00:00:00 counter

> criu dump -t 1799 -j

> Counter value:230672
> Counter value:230673 --> Killed
```

As reported in the listing 4.7 `CRIU` generated a list of binary files. Those files can be read using `CRIT` tool. The following sections will describe how researchers have found those information in the dump and how they built an automation process able to extract and compose those data as output interpretable by `CRIU`.

BackToLife plugin

In order to collect all data needed for the generation of the listed files, a Volatility plugin have to be implemented. The plugin developed for this thesis was called `linux_backtolife`, it uses other plugins and offers new features in order to generate `.json` files needed by `crit` for generating image files. The plugin has firstly to take care of the selection of target process, it can be done using a standard plugin, as `pslist` which automatically select the process with `PID` passed as parameter.

```
> ls -la
-rw-r--r--   1 root  staff   936B 15 Gen 16:14 core-1799.img
-rw-r--r--   1 root  staff    44B 15 Gen 16:14 fdinfo-2.img
-rw-r--r--   1 root  staff    18B 15 Gen 16:14 fs-1799.img
-rw-r--r--   1 root  staff    32B 15 Gen 16:14 ids-1799.img
-rw-r--r--   1 root  staff    38B 15 Gen 16:14 inventory.img
-rw-r--r--   1 root  staff   763B 15 Gen 16:14 mm-1799.img
-rw-r--r--   1 root  staff   120B 15 Gen 16:14 pagemap-1799.img
-rw-r--r--   1 root  staff    96K 15 Gen 16:14 pages-1.img
-rw-r--r--   1 root  staff    26B 15 Gen 16:14 pstree.img
-rw-r--r--   1 root  staff   328B 15 Gen 16:14 reg-files.img
-rw-r--r--   1 root  staff   752B 15 Gen 16:14 sigacts-1799.img
-rw-r--r--   1 root  staff    35B 15 Gen 16:14 stats-dump
-rw-r--r--   1 root  staff   178B 15 Gen 16:14 tty-info.img
-rw-r--r--   1 root  staff    32B 15 Gen 16:14 tty.img
```

Figure 4.7. `CRIU` output files.

The development choice was to extend the `proc_maps` plugin which extends `pslist` but returns also the memory area of the target process. This choice was taken because `linux_backtolife` have also to manage memory area. The basic part of the `linux_backtolife` plugin is shown in Figure 4.8. The `render_text` function overrides the corresponding method of the `proc_map` plugin, it receives the output of the `calculate` function of the extended plugin which is an array of couples of Volatility objects. These objects represent the task and the memory area. The task object is a VType which is relative to the `task_struct` of the Linux Kernel.

```
import volatility.obj as obj
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.proc_maps as linux_proc_maps


class linux_backtolife(linux_proc_maps.linux_proc_maps):
    """Generate images file for CRIU"""


    def __init__(self, config, *args, **kwargs):
        linux_proc_maps.linux_proc_maps.__init__(self, config, *args,
    **kwargs)

    def render_text(self, outfd, data):
        if not self._config.PID:
            debug.error("You have to specify a process to dump. Use the
    option -p.\n")

        for task, vma in data:
                ...
```

Figure 4.8. Basic structure of BackToLife plugin.

Every process in Linux is represented by a `task_struct` structure in the kernel which must have

a clear picture of what each process is doing. It must know, for example, the process's priority, if it is running on a CPU or in a waiting state, what is his address space, which files can be accessed by it, and so on. This is the role of the process descriptor (`task_struct`) which contains all the information related to a single process. In addition to a large number of fields containing process attributes, the process descriptor contains several pointers to other data structures. Through the Volshell it is possible to understand how a task struct is composed, and the main fields are represented in the Figure 4.9.

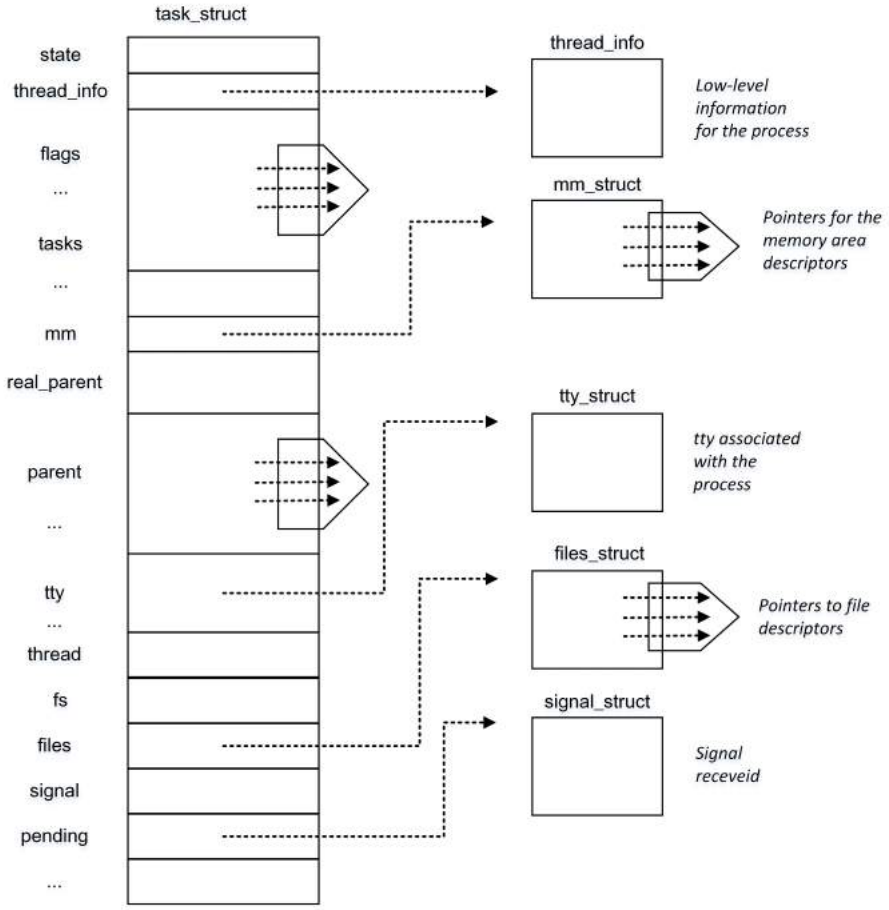

Figure 4.9. Representation of the `task_struct` structure.

Analysis of generated files

Before starting with the analysis of all `CRIU` files, an important tool which have been developed will be explained. Before developing code for automatically generate image files, values were often researched through the `volshell`. As explained in section 2.3.2, this particular shell permits to navigate inside the memory dump through many structures. It is often very difficult to find target values, this is why a particular tool was implemented. It is the `findAttributeVolatility.py` script, which have to be executed inside the `volshell`. The purpose of the tool is to search, starting from a root structure to all leaf children, all fields with a specified name, it defines a function `search()` which receive the root object, the name of the root and the name which have to be searched. An example is following:

```
>>> execfile("/BackToLifeTools/findAttributeVolatility.py")
>>> p = proc()
>>> search(p, "p", "signal")

p.exit_signal : 17
```

```
p.pdeath_signal : 0
p.signal : 18446612134328097600
p.signal.shared_pending.signal : 18446612134328097680
p.sighand.signalfd_wqh : 18446612134339004816
p.pending.signal : 18446612134363031528
```

Generation of file: `sigacts.img`

The `sigacts.img` image file contains the list of all *sigactions* registered by the target process. In order to understand what *sigactions* are, it is important to explain what are Linux signals. Signals are a fundamental method for interprocess communication, kernel uses them to notify a process if something happened, developers use them for synchronize processes. Signals are very important in Linux for handling exceptions and interrupts. A signal is generated when an event occurs, sometimes a process can send a signal to other processes and the kernel originates a signal in many situations, for example when a file size exceeds limits, when an I/O device is ready, when encountering an illegal instruction or when the user sends a terminal interrupt. In Linux sixty-four signals are defined, their name start with `SIG` and they can be identified with the corresponding code number. In order to send a signal to a target process it is possible to use the shell command `kill`, specifying the number or the name of the signal and the `PID` of the target process. When a process receives a signal it can decide how to handle it:

- The signal can be ignored: so nothing will be done when the signal occurs. Most of the signals can be ignored but not all of them, for example `SIGKILL` and `SIGSTOP` interrupts.
- The signal can be caught: a handler function is executed, the programmer can assign a handler to each signals.
- The default action is executed: if a programmer does not assign a special handler to the signal, the corresponding default action is executed. If the default action does not exist, the signal is ignored by default.

In order to specify how to handle a signal, developers can use the `signal(signal, handler)` system call, through it, it is possible to specify which action have to be performed when the target signal is received. Another way for specifying how a signal have to be handled is the `sigaction ( signum, act, oldact )` system call, it is more powerful because it permits to have a better control over a given signal. Through it developers can assign to the specified signal (`signum`) the `sigaction act`, storing the old action in the `oldact` structure. In the `sigacts.img` file CRIU store all the setted `sigaction` structures of the target process, avoiding signals which can not be overridden (`SIGKILL` and `SIGSTOP`). The collection of `sigaction` structures in Volatility is not performed by default, therefore the implementation of a new plugin was needed. The basic idea of the `linux_dump_signals` plugin is to search in memory the target structures starting from the `sighand` field of the target process descriptor, which contains the array of sixty-four actions.

```
>>> dt("task_struct")
 'task_struct' (2376 bytes)
...
0x628 : sighand                 ['pointer', ['sighand_struct']]
...
```

```
>>> dt("sighand_struct")
 'sighand_struct' (2088 bytes)
0x0   : count                   ['__unnamed_0x346']
0x8   : actions                 ['array', 64, ['k_sigaction']]
0x808 : siglock                 ['spinlock']
0x810 : signalfd_wqh            ['__wait_queue_head']
```

The extraction of `sigaction` structures using the integrated objects in Volatility is not correctly performed, in particular some values are not collected in the right way. This is the reason which brought developers to write a completely new strategy for reading values from the memory dump. For understanding the main part of the `linux_dump_signals` it is important to clarify how the target structure is composed.

```
>>> dt("sigaction")
 'sigaction' (32 bytes)
0x0   : sa_handler              ['pointer', ['void']]
0x8   : sa_flags                ['unsigned long']
0x10  : sa_restorer             ['pointer', ['void']]
0x18  : sa_mask                 ['__unnamed_0xa602']
```

The `sa_handler` field specifies which action have to be performed when the corresponding signal is received, the `sa_mask` is a mask that specifies which signals should be blocked, the `sa_flags` field store flags which modify the behaviour of the corresponding signal, each bit of this field represents a different flag, which can be 0 or 1. The last field, `sa_restorer` is not for application use, it is used when the signal is handled and the processs have to return to the previous context. From the previously presented `sighand_struct` structure, it is possible to understand that bytes within `0xF` and `0x808` compose the array of actions. The strategy implemented in the plugin is to parse these bytes for composing the `sigaction` structure. The main parts of the plugin are the presented in Figure 4.12. It extends the `pslist` plugin for selecting the target process specified with the `-p` parameter, as output it prints a table with a row for each signal. In each row extracted data are listed. In the `read_sigactions()` method the parsing is performed reading the right number of bytes directly from the memory dump.

Generation of file: fdinfo.img

The `fdinfo.img` image file contains the list of open file descriptors at the checkpointing time. In a simple process like the counter there are normally only file descriptors relative to the used `tty` pseudo terminals. In more complex programs it should contain also elements relative to opened files or opened sockets. In Linux and others Unix systems each resource can be seen as a file, a file descriptor is an integer which represent an open file (or resource) for a process. It can be also considered as an index in a table which contains file references for each process and that is called `file_table`. Each entry of this table points to an other element included in a general kernel table which contains informations, as for example the access mode or permissions, for each open file. Each element of this last table points to elements of another kernel table containing all `inode` structures for all open files and resources. Each Unix process has its own file table, which contains at least three file descriptors relative to the three standard streams: `stdin`, `stdout` e `stderr` which are the three standard pseudo terminal.

Using Volatility it is possible to find the `file_table` relative of a selected process, through the volshell it is possible to access to the `task_struct` which contains the pointer to the desired table: `files` field.

```
>>> dt(proc())
[task_struct task_struct] @ 0xFFFF88007A1E6190
0x0   : state 132
...
0x370 : pid 1799
...
0x608 : fs 18446612134340226176
0x610 : files 18446612134329092160
...
```

```
class linux_dump_signals(linux_pslist.linux_pslist):
    """Dumps sigactions of a process"""


    #Method for generating sigactions
    def read_sigactions(self, task):

        sigacts = []
        handler = task.sighand
        action_vector = handler+8

        #Signal SIGKILL(9) and SIGSTOP(19) are not considered
        for i in range(1, 65):
            if i == 9 or i == 19:
                continue

            action = self.read_field(task, action_vector, 8)
            action_vector += 8
            ...
            action_element = {"sigaction":action, ...}
            sigacts.append(action_element)

            action_vector += 8

        return sigacts
```

Figure 4.10. Parsing of sigaction structures in the `linux_dump_signals` plugin.

Following the pointer the table is accessed, each entry corresponds to a file descriptor and it contains the pointer to the relative `file_struct`. From each of these structures, the `inode` can be accessed.

In order to find the list of used file descriptor, in the `linux_backtolife` plugin, the `task` object is used, through its method `lsof()` it is possible to get access to the `file_table` for reading informations about all open files. The `fdinfo` file contains the list of all used file descriptor, for each of them it indicates also the kind of resource: TTY, regular file or socket. For collecting this information the Volatility plugin get access to the `inode` structure corresponding to the file descriptor. When needed informations are collected the plugin write all informations with the expected format in a `.json` file.

Generation of file: pstree.img

The `pstree.img` is an image file which contains the process hierarchy. Specifically, for each process of the tree, it has an entry, and for each entry some data are stored. In particular for each process CRIU stores: the *pid*, the `ppid`, the `pgid`, the `sid` and the array which collects the ids off all thread of the process. In Linux each running process can be identified by an unique number called `pid` stored in the corresponding field of the `task_struct` structure. The process ids increase sequentially until the upper limit is reached, the kernel reuses a used `pid` when the corresponding process terminate. Also other informations contained in the `pstree` image file can be collected directly from the `task_struct` structure, theese data are very similar to the `pid` and represent:

- `ppid`: it is the process identifier of the process's parent.

```
def dump_fd_info(self, task, sockets_type):


        fdinfoFile = open("fdinfo-2.json", "w")
        entries = []
        for file, fd in task.lsof():
                ...
                element = {"id":0, "flags":0, "type":"", "fd":int(fd)}
                entries.append(element)


        data = {"magic":"FDINFO", "entries":entries}
        fdinfoFile.write(json.dumps(data))
        fdinfoFile.close()
```

Figure 4.11. Generation of fdinfo file in the BackToLife plugin.

```
#Build pstree file for CRIU with info about process and his threads
def buildPsTree(self, task):
    pstreeData = {"magic":"PSTREE", "entries":[{
                                                "pid":int(str(task.pid)),
                                                "pgid":int(str(task.pid)),
                                                ...
                                                }]}
    threads = []
    for thread in task.threads():
        threads.append(int(str(thread.pid)))

    pstreeData["entries"][0]["threads"] = threads

    ...
```

Figure 4.12. Pstree image file construction.

- `pgid`: it represent the *process group id*, it is equal to the `pid` of the group leader. The concept of processes group permits to the kernel to know which processes are working together.
- `sid`: it is the session identifer. The `sid` is the `pid` of the process which had created the current session, known as session leader

Generation of file: mm-PID.img

The `mm-PID.img` image file contains information about virtual memory of the process. Analysing the file using `CRIT` tool the first part of the binary file contains information about starting and ending addresses of several memory regions such as *code* segment, *data* segment, *stack* segment, the *heap*, the command-line *arguments* and the *environment* varialbes. For each regions there are several information that will be explained in this paragraph.

The *code* segment, also known as text segment contains the machine instructions of the program

that will be executed by the processor. The text segment of an executable object file is often read-only segment that prevents a program from being accidentally modified. The *data* segment contains the static data of the program, i.e. the variables that exist throughout program execution. Data segment is further divided into four sub-data segments: initialized data segment, uninitialized or .bss data segment, stack, and heap. This kind of division is useful to store variables depending upon if they are local or global, and initialized or uninitialized. Initialized data or simply data segment stores all global, static, constant, and external variables that are initialized beforehand. Contrary to initialized data segment, uninitialized data or .bss segment stores all uninitialized global, static, and external variables (declared with extern keyword). Global, external, and static variable are by default initialized to zero. The use of the term .bss to denote uninitialized data is universal. It was originally an acronym for the "Block Storage Start" instruction from the IBM 704 assembly language. The *stack* segment contains the system stack, which is used as temporary storage. Stack segment is used to store all local variables and is used for passing arguments to the functions along with the return address of the instruction which is to be executed after the function call is over. The stack is a simple data structure with a LIFO (last-in first-out) access policy. Items are only added to or removed from the top of the stack. Implementing a stack requires only a block of memory (e.g. an array in a HLL) and a stack pointer which tells where the top of the stack is. In the MIPS architecture, the $sp register is designated as the stack pointer. Adding an element to the top of the stack is known as a push, and retrieving an item from the top is known as a pop. The stack is a dynamic structure that dynamically change depending on the context and the local variable. The *heap* segment is a pool of memory used for dynamically allocated memory, such as with `malloc()` of `calloc()` in C or new in C++ and Java. The stack and heap are traditionally located at opposite ends of the process's virtual address space. The Figure 4.13 represent a scheme of different memory area.

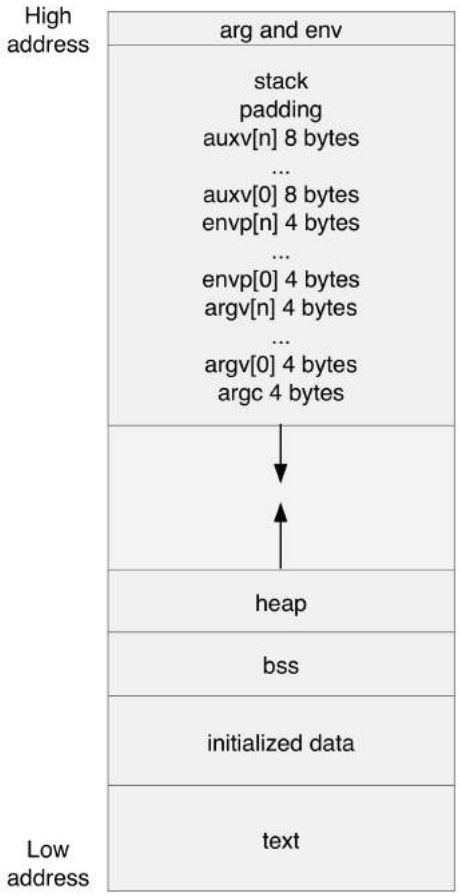

Figure 4.13. Memory layout of a program.

In order to find the starting and ending addresses for each memory region it has been analysed through the `Volshell` the `task_struct` struct which contains at the address `0x2d0` the pointer to the `mm_struct`. The `mm_struct` is a 944 bytes Linux kernel struct that contains information needed such as `start_code`, `end_code`, `start_data`, `end_data`, `start_brk`, `brk`, `start_stack`, `arg_start`, `arg_end`, `env_start` and `env_end`. In the mm file there is also the `mm_saved_auxv` that is an array that contains values of the auxiliary vector.

As reported in [68] ELF auxiliary vectors are a mechanism to transfer certain kernel level information to the user processes. When a program is executed, it receives information from the operating system about the environment in which it is operating. The form of this information is a table of key-value pairs. Some of the data is provided by the kernel for libc consumption, and may be obtained by ordinary interfaces, such as sysconf. However, on a platform-by-platform basis there may be information that is not available any other way. An example of such an information is the pointer to the system call entry point in the memory (`AT_SYSINFO`). This kind of information is dynamic in nature and is only known after kernel has finished up loading. The information is passed to the user process by binary loaders which are part of the kernel subsystem itself. Binary loaders convert a binary file, a program, into a process on the system. Usually there is a different loader for each binary format. The most used binary format is the ELF format. The ELF binary loader is defined in the file `/usr/src/linux/fs/binfmt_elf.c`. The ELF loader parses the ELF file, maps the various program segments in the memory, sets up the entry point and initializes the process stack. It puts ELF auxiliary vectors on the process stack along with other information like `argc`, `argv`, `envp`. After initialization, a process stack is similar to that one depicted in the Figure ẽffig:memory. The structure of an auxiliary vector is defined in `/usr/include/elf.h` and is represented here below:

```
typedef struct
{
  uint32_t a_type;              /* Entry type */
  union
    {
      uint32_t a_val;           /* Integer value */
    } a_un;
} Elf32_auxv_t;
```

As reported in [69] the primary customer of the auxiliary vector is the dynamic linker (`ld-linux.so`). In the usual scheme of things, the kernel's ELF binary loader constructs a process image by loading an executable into the process's memory, and likewise loading the dynamic linker into memory. At this point, the dynamic linker is ready to take over the task of loading any shared libraries that the program may need in preparation for handing control to the program itself. However, it lacks some pieces of information that are essential for these tasks: the location of the program inside the virtual address space, and the starting address at which execution of the program should commence. In theory, the kernel could provide a system call that the dynamic linker could use in order to obtain the required information. However, this would be an inefficient way of doing things: the kernel's program loader already has the information because it has scanned the ELF binary and built the process image and knows that the dynamic linker will need it. Rather than maintaining a record of this information until the dynamic linker requests it, the kernel can simply make it available in the process image at some location known to the dynamic linker. That location is, of course, the auxiliary vector. It turns out that there's a range of other information that the kernel's program loader already has and which it knows the dynamic linker will need. By placing all of this information in the auxiliary vector, the kernel either saves the programming overhead of making this information available by implementing a dedicated system call. Among the values placed in the auxiliary vector and available via the `getauxval()` system-calls there are important values such as:

- `AT_PHDR` and `AT_ENTRY`: The values for these keys are the address of the ELF program headers of the executable and the entry address of the executable. The dynamic linker uses this information to perform linking and pass control to the executable.

- AT_SECURE: The kernel assigns a nonzero value to this key if this executable should be treated securely. This setting may be triggered by a Linux Security Module, but the common reason is that the kernel recognizes that the process is executing a set-user-ID or set-group-ID program. In this case, the dynamic linker disables the use of certain environment variables and the C library changes other aspects of its behavior.

- AT_UID, AT_EUID, AT_GID, and AT_EGID: These are the real and effective user and group IDs of the process. Making these values available in the vector saves the dynamic linker the cost of making system calls to determine the values. If the AT_SECURE value is not available, the dynamic linker uses these values to make a decision about whether to handle the executable securely.

- AT_PAGESZ: The value is the system page size. The dynamic linker needs this information during the linking phase, and the C library uses it in the implementation of the malloc() and calloc() functions.

- AT_PLATFORM: The value is a pointer to a string identifying the hardware platform on which the program is running. In some circumstances, the dynamic linker uses this value in the interpretation of rpath values.

- AT_SYSINFO_EHDR: The value is a pointer to the page containing the Virtual Dynamic Shared Object (VDSO) that the kernel creates in order to provide fast implementations of certain system calls.

- AT_HWCAP: The value is a pointer to a multibyte mask of bits whose settings indicate detailed processor capabilities. This information can be used to provide optimized behavior for certain library functions.

- AT_RANDOM: The value is a pointer to sixteen random bytes provided by the kernel. The dynamic linker uses this to implement a stack canary.

The whole list of possible key values is defined in the header files /usr/include/linux/auxvec.h and asm/auxvec.h.

In order to extract and generate the correct output for the mm-PID.img file researchers developed a Volatility plugin that is able to walk through the stack and extract each key-value pairs for the auxiliary vector. The plugin is called linux_dump_auxv. The most significant part of the linux_dump_auxv plugin is reported in the listing řeffig:dumpauxv. The function read_auxv called in the render_text is responsible for walking through the stack and gather the values.

Another fundamental vector that is present in the mm-PID.img file is the list containing information about each memory regions. Such information include the start and end virtual address of the memory area, flags, status, the pgoff value, the prot string and the shmid value.

In order to obtain those values, the vm_area_struct kernel struct has been analysed using the Volshell. CRIU expects values in a certain format for flags, status and prot. Specific translation function has been developed for this purpose in the linux_backtolife plugin. Functions protText(), statusText() and flagsText() convert values found in the vm_area_struct in the CRIU expected format.

The method getShmid(self, progname, current_name, dic, task) of the linux_backtolife plugin has been developed for generating the shmid number. It takes the maximum file descriptor id and assigns it to program. For all other files (e.g. shared libraries) the *id* is returned incrementally.

Generation of file: pagemap.img

The pagemap.img image file contains information about virtual memory of the process and the number of sequential pages starting from a certain virtual address. The output generated by CRIT contains for each memory address the nr_pages starting from that virtual address. The

```
class linux_dump_auxv(linux_pslist.linux_pslist):
    """Read Auxiliary Vector of a process"""

    def __init__(self, config, *args, **kwargs):
        linux_pslist.linux_pslist.__init__(self, config, *args, **kwargs)

    def calculate(self):
        if not self._config.PID:
            debug.error("You have to specify a process to dump. Use the
    option -p.\n")

        #Retrieve the task_struct of the process
        tasks = linux_pslist.linux_pslist.calculate(self)
        for task in tasks:
            yield task

    #Method for reading an address range in memory dump
    def read_addr_range(self, task, start, size):
        proc_as = task.get_process_address_space()
        segment = proc_as.zread(start, size)
        return segment

    def render_text(self, outfd, data):
        global auxv
        for task in data:
            print "Auxiliary Vector for process: {0}".format(self._config.PID)
            aux = self.read_auxv(task)
            self.table_header(outfd, [("Key", "16"), ("Value", "#018x")])
            for i in range(0, 46, 2):
                key = aux[i]
                value = aux[i + 1]
                self.table_row(outfd, auxv[key], value)
                if key == 0:
                    break
```

Figure 4.14. Part of the developed linux_dump_auvx plugin.

`pagemap.img` file is used by `CRIU` in order to understand the composition of the file `pages-1.img`. The researchers developed the part of `linux_backtolife` plugin that generates `pagemap.img` image file exploiting the `linux_proc_maps` Volatility plugin that gives information about memory regions. The virtual addresses that are taken in account for the generation of `pagemap.img` file are all virtual addresses of those memory regions that are readable and/or writeable. Researchers understood that `CRIU`, for shared libraries, wants only not executable segments (only `r--` or `rw-`). It is important to note that for dynamic regions such as stack or heap the `linux_backtolife` plugin generates the maximum number of pages for those regions. This is done because of simplicity avoiding to find the minimum number of pages. The consequence of this fact is that the `pages-1.img` file is bigger than `CRIU` generated file because `CRIU` dumps only the minimum size of the stack and the heap. Another part that is not taken into account is the VDSO and vsyscalls area. Before explaining why those regions are not considered into the `pagemap.img` file it is important to understand what kind of information those regions contain. As reported in [70] The vDSO (Virtual Dynamic Shared Object) is a shared library that the kernel automatically maps into the address space of all user-space applications. Applications usually do not need to concern themselves with these details as the vDSO. The developer writes code and the C library will take care of using any functionality that is available via the vDSO. There are some system-calls that are used frequently

that the kernel maps into that region for raising the performance. Making system calls can be slow. The instruction that trigger a software interrupt can be very expensive. Rather than require the C library to figure out if this functionality is available at run time, the C library can use functions provided by the kernel in the vDSO. The base address of the vDSO is passed by the kernel to each program in the initial auxiliary vector via the `AT_SYSINFO_EHDR` key. The vDSO is not mapped at any particular location in the user's memory map. The base address will usually be randomized at run time every time a new process image is created. This is done for security reasons, to prevent the *return-to-libc* attacks. For some architectures, there is also an `AT_SYSINFO` tag. This is used only for locating the vsyscall entry point and is frequently omitted or set to 0. This tag is a throwback to the initial vDSO and its use should be avoided. The *vsyscall* was added as a way to execute specific system calls which do not need any real level of privilege to run. The classic example is `gettimeofday();` all it needs to do is to read the kernel's idea of the current time. There are applications out there that call `gettimeofday()` frequently, to the point that they care about even a little bit of overhead. To address that concern, the kernel allows the page containing the current time to be mapped read-only into user space; that page also contains a fast `gettimeofday()` implementation. Using this virtual system call, the C library can provide a fast `gettimeofday()` which never actually has to change into kernel mode. The reason why vsyscall area and vDSO is not mapped into the `pagemap.img` file is that is kernel-dependent and is not possible to pass to `CRIU` a vDSO area that is different on the resurrection machine. The same concept is applied to any other shared library. The `reg-files.img` contains references of files, shared libraries and binary. The `prepareMachine.py` will remap all shared libraries and files used by the program into the resurrection machine with the libraries and files in the new paths.

Generation of file: pages-1.img

The `pages-1.img` image file contains the raw memory of the process. In fact, this kind of file cannot be analysed using `CRIT` tool. if `CRIT` is launched, the message returned is:

```
>Unknown magic 0x464c457f. Maybe you are feeding me an image with raw
    data(i.e. pages.img)?
```

The `pages-1.img` file is binary and researchers used `xxd` program in order to analyse the content of the `pages-1.img` file. The output of `xxd` is similar to:

```
0000000: 7f45 4c46 0201 0100 0000 0000 0000 0000  .ELF............
0000010: 0200 3e00 0100 0000 1004 4000 0000 0000  ..>.......@.....
0000020: 4000 0000 0000 0000 d812 0000 0000 0000  @...............
```

Several tools have been developed in order to analyse the binary `pages-1.img` file.

- `findBytes.py`: this script is able to find specific stream of bytes in raw binary file.
- `compareBinary.sh`: this script has been used for comparing different binary file and find differences between them.
- `pageSeparator.py`: this Python script, given a binary file, is able to separate it in pages of 4096 Byte counting how many pages are empty.
- `hexSeparator.py`: this Python script convert binary files using *xxd* command and divede the output in pages of 4096 Byte.
- `searchBytesHex.sh`: this Bash script is able to find hexadecimal pattern in a binary file that has been converted using `xxd` command.

The `pages-1.img` file is composed by all regions that are listed in the `pagemap.img` file. Code segment, Data segment, Heap region, Stack region, all shared libraries that are readable and writeable and not executable, vDSO, vsyscall and so on. In order to understand what regions are necessary for obtaining a correct `pages-1.img` file several test have been done. The most difficult part of the generation of this file was to understand that regions such as vDSO are generated on

```
#!/usr/bin/env python
from ctypes import *

for ln in open('/proc/self/maps'):
    if "[vdso]" in ln:
        start, end = [int(x,16) for x in ln.split()[0].split('-')]
        CDLL("libc.so.6").write(1, c_void_p(start), end-start)
        break
```

Figure 4.15. Generation of the vDSO.

the backtolife machine for correctness. In order to properly work the `pages-1.img` file as the last segment needs to have the vDSO area. The generation of the vDSO is done on the resurrection machine using the `prepareMachine.sh` script. This script calls `generateVDSO.py` that creates the vDSO area that is appended to the `pages-1.img`. The code of the `generateVDSO.py` script is reported in the listing 4.15.

Generation of file: reg-files.img

The `reg-files.img` image file contains information about the binary file of the program, all shared libraries and all regular files that are created by `open()` function.

The information that is needed is the path of the file, the size, flags and the fown (`uid`, `euid`, `signum`, `pid_type`, `pid`).

This file is generated by `linux_backtolife` plugin but is modified before the resurrection on the backtolife machine using the `prepareMachine.sh`. The `prepareMachine.sh` remaps files and shared library on the resurrection machine in order to properly works the resurrection. For each entry in the `reg-files.img` file is associated a `id` that is a general identifier for that resource. In all other files generated by `CRIU`, a reference to that resource is done using that `id`. Particular entries in `reg-files.img` file are the `/dev/pts/0` default pts device, the `/` root folder and the binary ELF file of the program. Another useful information is the `pid` of threads that are eventually created.

In order to generate `reg-files.img` the `linux_backtolife` Volatility plugin uses the `lsof()` method of the `task_struct` class. This method returns information about files that are used by the process. In order to get the path of a regular file or a shared library the function `linux_common.get_path(task, filp)` where `filp` is returned by the `lsof()` method. It is generated a file that is not directly useful for the resurrection because it is an intermediate file that reports information whether the file is a socket or a regular file or the binary or a shared library. The `prepareMachine.sh` script parsing this information is able to remap the correct entries and it generates the final file that is given to `CRIU` for a correct resurrection.

Generation of file: core.img

In the `core.img` file CRIU stores different informations about the process at the checkpoint time. It is one of the bigger image file, it is composed by three main sections which will be explained in details. CRIU generates one core file for each thread and store inside it a various kind of informations. The main section of this image file are the following:

- `thread_core`: it contains various informations about the context of the thread at the checkpointing time, for example, in it CRIU stores data about the security context, signals handler and other data regarding the context of the thread.
- `thread_info`: it is a section which contains informations about registers, in particular it stores register value, also for the *floating point unit*.

- `tc`: it contains informations about the thread state at the checkpointing time, data contained in this section can be easily collected because most of them represent field of the `task_struct` structure.

In order to understand how to retrieve data for generating the `core.img` file, the `volshell` was used and information were searched starting from the `task_struct` structure of the target process, then the `linux_backtolife` was upgraded for automatically gather target informations. The first section, `thread_core` contains many informations about the thread context, for example the security context of a task, contained in the `struct cred` of the process descriptor. Not all fields of the `cred` structure can be red directly by the `Volshell`, some array fields were not correctly extracted by Volatility, for these informations the `linux_backtolife` plugin read directly from the memory dump, parsing the target structures.

```
addr = int(thread.cred.cap_inheritable.__str__())
dataByte = self.read_addr_range(task, addr, 4)
for c in dataByte:
    reverse.insert(0, "{0:02x}".format(ord(c)))

value = ''.join(reverse)
threadCoreData["creds"]["cap_inh"].append(int(value, 16))
```

The second main part of the `core.img` file contains informations about CPU registers, specifically, it includes values of them at the checkpointing time. Processor registers are very important in order to restore the state of the process, through them, a program store data and do operations, CPU use them for execute a program and using them it is possible to get access to many memory area. CRIU, in order to successfully restore a checkpointed process needs in particular the *general purpose registers* and some of *special purpose registers*, CRIU stores also values regarding the *floating point unit*. The *general purpose registers* are available to store any kind of data required by the current program, their number and length in bit change with the architecture. *Special purpose registers* are registers used by CPU for specific operations, for example the *instruction register* which holds the instruction currently in execution, or the *progam counter* that holds the address in memory of the next instruction which have to be fetched from memory and executed. In order to retrieve values for these registers, a standard volatility plugin was used: `linux_info_regs`. This plugin tries to collect registers values from the memory dump, and in order to perform this operation it takes advantage of the *context switch*. Registers values are not normally stored in memory, when the CPU switch from a process (or a thread) to another a *context switch* is performed, a context is represented by all CPU registers and the *program counter*. During a switch, all registers of the old process are saved in the *kernel stack* and the registers values of the new one are restored. The `linux_info_regs` is able to search in the *stack* of target process the values stored at the last *context switch*. In order successfully perform a process resurrection, register values have to be coherent, this is why in the analysis model, explained in subsection 4.3.1, before the dump acquisition the target process have to be paused. Pausing a process, force it to perform a *context switch* ensuring that when memory dump will be analysed register values in the *kernel stack* will be coherent and updated with the process state before the memory acquisition. The main part of registers extraction is listed in Figure 4.16.

Floating point registers are not involved in the standard *context switch* and can not be recovered using the standard plugin. In the `thread` structure of the Linux kernel FPU registers are saved, if used, when the *context switch* occours, the `linux_backtolife` plugin search fpu registers values in that structure. In particular, the `thread` structure contains a *fpu* struct which have a *state* field. Inside it, through the `Volshell` it is possible to find the last saved state of the FPU. Furthermore, not only FPU registers values have to be stored, CRIU requires also others field regarding the FPU state, for example the `xmm_space` or the *ymmh_space*. These fields are in the same struct of *floating point registers* but Volatility does not correctly read them, the implemented plugin reads byte by byte their values directly from the memory dump. The lat part of the `core.img` file is `tc`, it contains generic informations about the thread, as for example his state, his exit code and his *rlimits*. The *rlimits* field, is the only one which is filled locally in the *restore machine* througth the `prepareMachine.py` script. This array of values is generated in the machine where

```
info_regs = linux_info_regs.linux_info_regs(self._config).calculate()
for task, name, thread_regs in info_regs:
    for thread_name, regs in thread_regs:
        if regs != None:
            regsData = {"gpregs": {
                 "r15": "{0:#x}".format(regs["r15"]),
                 "r14": "{0:#x}".format(regs["r14"]),
                 "r13": "{0:#x}".format(regs["r13"]),
                 "r12": "{0:#x}".format(regs["r12"]),
                 ...
```

Figure 4.16. Registers values extraction from the kernel stack.

restore is performed, because `rlimits` represent resources usage limit for a process. For each limit the operating system specifies the *Soft Limit* and the *Hard Limit*, the main limits regard for example, the stack size, the data size, number of opened files or pending signals. These data are locally collected in order to adapt the process to the new Operating System, it is performed in the `prepareMachine.py` as presented in Figure 4.17.

```
#Generation of rlimits for core-{pid}.json
limits = os.popen("cat /proc/1/limits").readlines()
rlimitsJson = {"rlimits":[] }
for line in limits:
    if "Units" in line:
        continue
    limit = re.sub(' +',' ',line)
    limitArray = new_line.split(" ")
    softLimit = limitArray[0]
    hardLimit = limitArray[1]

    rlimitsJson["rlimits"].append({"cur":softLimit,"max":hardLimit})
```

Figure 4.17. Rlimits extraction in the new machine.

Generation of file: fs.img

The `fs.img` file contains informations about the linux file system, it is one of the smaller image file and at checkpoint time CRIU store in it data about the current file system. Specifically, the `fs.img` file contains three different values:

- `cwd_id`: it represents the id of the working directory in the `reg-files.img` file. The working directory is where, in linux file system, the original process was started. It is an information which is setted in the machine where restored is performed in order to match the real directory where the process is restored.
- `root_id`: it is the id of the *root* directory of the file system in the `reg-files.img` file. It represents the root directory of the file system, also this information is setted in the new machine, and it is not extracted directly from the memory dump.

- `umask`: it is a permission representation often used in Unix system, it is similar to normal octal mode, but in the `umask`, when a bit have the value 1, the corresponding permission is disabled. In the `fs.img` image file it is stored the umask of the *root directory* of the file system.

All theese informations can not be extracted by the memory dump because CRIU need the real information regarding the machine where the resurrection will be performed, for this reason the `fs.img` file is entirely created through the `prepareMachine.py` script in the new machine.

Generation of file: tty.img and tty-info.img

In these image files CRIU stores informations about `tty`, in Unix, a `tty` is a device which can be used by the user for communicate with a command interpreter or other applications. At the beginning with the term `tty`, it was indicated the physical device composed by a keyboard and a monitor, nowadays, with te same term it is indicated also the special file, used by the operating system or other applications for talking with a peripheral device. In the `tty.img` file, CRIU saves the list of `tty` used by the target process, for each entry, it stores properties about the owner process, an `id` which is a reference of the object in the `reg-files` image file and another id (`tty_info_id`) as a reference in the second file: `tty-info.img`. The second file contains more detailed informations about each used `tty`. The main field is the *termios - Terminal Input Output Settings*, which contains the configuration parameters for the `tty`, as for example the transmission speed, special characters or various flags. In the info file for each entry it is also specified if it is a `pty`, this particular type of `tty` is a *pseudo terminal*. It is a pair of two virtual devices, the slave which emulates a terminal device and the master which gives to processes de possibility to control the slave. `Pty` is used for example in all emulated terminal like `xterm`. In the `tty-info.img` file, if an entry correspond to a `pty`, parameters about terminal window are stored. These two file are entirely generated locally through the `prepareMachine.py` script, collecting information from the operating system of the machine where the target process will be restored.

Generation of file: other static files

For the thesis development, and in particular for the first implementation of the `linux_backtolife` plugin, example programs were used and checkpointed through CRIU. Images files have been analysed in order to understand what they contain and how to retrieve informations. During the analysis it was understood that some files are static ore unused for CRIU. It is the case of the last two missing files: `ids.img` and `inventory.img`. These file are not extracted or generated, because they contain informations about the checkpointing procedure and not of the target process, they were used as static in all resurrection experiments.

4.3.4 A network-based process

With the objective of more complex processes resurrection, the next kind of programs which have been analysed are network based programs. In order to perform this analysis, two different programs were written, a client and a server. They communicate through *TCP* socket, the client sends to the server two integers, and receives as answer the result of the addiction of the two numbers. The *DACA* model was applied to the client program in order to understand how CRIU performs the checkpoint and restore of programs which use network. For performing the checkpoint of the client program CRIU have to be launched with a new parameter which informs that a *TCP* connection have been established.

```
> criu dump -t PID --shell-job --tcp-established
```

In the list of generated files, it is possible to distinguish two new files regarding the network connections, these images have been analysed in order to extract data from the memory dump.

Analysis of generated files

During the analysis of new generated files, it was clear that some data could not be extracted from the `volshell`, because it does not integrate some kernel data structures. This led developers to build a new Volatility profile which includes all needed kernel structures for generating necessary VTypes. Data extraction for network-based process is performed in a separated implemented plugin: `linux_dump_sock`.

Generation of file: `inetsk.img`

In the `inetsk.img` file CRIU stores all generic informations about used network sockets. For each entry, the image file, contains data relative to all kind of network socket. Generically, a network socket is a system resource which represents an internal endpoint used for sending or receiving data. Every process can use a socket referring to it using a *file descriptor*. Typically, sockets are associated with a specific socket address composed by the IP address and a port number, if the socket is connected, there is also a corresponding socket address at the peer node. When a developer wants to create a network socket, he has to choose the type, the family and setup the address. There are three different types of network socket:

- *Datagram* socket: connectionless type used for the *UDP* protocol.
- *Stream* socket: connection oriented, it is used in all connection based protocols as for example the *TCP*.
- *Raw* socket: this is a particular type, the address is composed only by the IP address because the transport layer is bypassed.

Network sockets can be also distinguished depending on their *family*, it specifies a communication domain, all various families are defined in the `socket.h` library of the Linux kernel, for the purpose of the thesis, the three main used families are:

- `AF_UNIX`: it corresponds to the *unix socket family*, used for local communication.
- `AF_INET`: it corresponds to socket used for all internet protocols, based on *IPv4*.
- `AF_INET6`: it is similar to the previous one but for *IPv6*.

In order to perform the collection phase, it is important to understand where the Linux kernel stores all informations regarding the sockets, three main structures are involved, and in particular data can be searched in: `struct sock_common`, `struct sock` and `struct socket` structures. Using the `volshell` it is possible to get all sockets used by a process with the method `task.netstat()` which returns an array of `socket` structures. Then it is possible to distinguish UNIX from network sockets through the *family* field. The result of `netstat()` function is an instance of a VType representing the `struct socket`:

```
>>> dt("socket")
 'socket' (48 bytes)
0x0   : state           ['Enumeration', {'target': 'int', 'choices': {0:
    'SS_FREE', ...}}]
0x4   : type            ['short']
0x8   : flags           ['unsigned long']
0x10  : wq              ['pointer', ['socket_wq']]
0x18  : file            ['pointer', ['file']]
0x20  : sk              ['pointer', ['sock']]
0x28  : ops             ['pointer', ['proto_ops']]
```

It is the socket representation in the Linux kernel, and contains an instance of the `sock` structure which with `sock_common` contains all important fields for the `inetsk.img` image file.

```
>>> dt("sock")
 'sock' (704 bytes)
0x0   : __sk_common     ['sock_common']
...
0xdc  : sk_rcvbuf       ['int']
...
0x124 : sk_sndbuf       ['int']
...


>>> dt("sock_common")
'sock_common' (112 bytes)
0x0   : skc_addrpair    ['unsigned long long']
0x0   : skc_daddr       ['unsigned int']
0x4   : skc_rcv_saddr   ['unsigned int']
0x8   : skc_hash        ['unsigned int']
0x8   : skc_u16hashes   ['array', 2, ['unsigned short']]
0xc   : skc_portpair    ['unsigned int']
0x10  : skc_family      ['unsigned short']
0x12  : skc_state       ['unsigned char']
...
```

Data extraction and automation of these data can be easily performed through the previously presented structures, and in particular it is performed in the `linux_dump_sock` plugin as showed in Figure 4.18.

```
import socket
import volatility.obj as obj
import volatility.plugins.linux.common as linux_common
import volatility.plugins.linux.pslist as linux_pslist


class linux_dump_sock(linux_pslist.linux_pslist):
    """Dumps info about opened network socket of a process"""


    def __init__(self, config, *args, **kwargs):
        linux_pslist.linux_pslist.__init__(self, config, *args, **kwargs)

    def get_sock_info(self, task, addrspace):
        for ents in task.netstat():
            if ents[0] == socket.AF_INET:
                (node, proto, sock_saddr, sock_sport, ...) = ents[1]


                sock_state = node.sk.__sk_common.skc_state
                sock_proto = node.sk.sk_protocol
                sock_family = node.sk.__sk_common.skc_family
                sock_type = node.sk.sk_type
                ...
```

Figure 4.18. Basic structure of `linux_dump_sock`.

Generation of file: `tcp-stream-[inode].img`

CRIU stores in the `tcp-stream-[inode].img` file all informations about the *tcp-stream* sockets, if a process uses multiple TCP sockets, CRIU generates a `tcp-stream-[inode].img` image for each socket. The name of the file changes because CRIU append to it the relative `inode` number in *base64* form. Analysis and extraction phases for generating this image file required many efforts because Volatility does not support this kind of extraction. All data contained in the image file are contained or calculated starting from fields of the `struct sock_info` of the Linux kernel. The VType for this particular structure is not included in Volatility because it is not in the file used for generating the profile, for the purpose of this thesis, a new Volatility profile have been created in order to generating all needed VTypes. It required analysis of kernel code in order to understand how this particular structure can be included in a kernel module. When the new profile was compiled in the `volshell` the relative VType can be used:

```
>>> dt("tcp_sock")
 'tcp_sock' (1800 bytes)
0x0   : inet_conn           ['inet_connection_sock']
0x4e0 : tcp_header_len      ['unsigned short']
0x4e2 : gso_segs            ['unsigned short']
0x4e4 : pred_flags          ['unsigned int']
0x4e8 : rcv_nxt             ['unsigned int']
0x4ec : copied_seq          ['unsigned int']
0x4f0 : rcv_wup             ['unsigned int']
0x4f4 : snd_nxt             ['unsigned int']
0x4f8 : snd_una             ['unsigned int']
0x4fc : snd_sml             ['unsigned int']
...
```

Also if the structure is included in Volatility, it does not mean that the framework will fill it with desired informations. In order to retrieve needed data, it was necessary to understand how CRIU collect them, from the open source code it was possible to understand that CRIU uses a particular system call which fill a `tcp_info` structure with all informations using a `tcp_sock` structure.

```
void tcp_get_info(struct sock *sk, struct tcp_info *info);
```

The behaviour of the system call has been replicated inside the `linux_dump_sock` plugin for gathering what is needed. The kernel code was studied and the function has been partially replicated in his main parts. In order to fill an instance of the `tcp_sock` structure, a `sock` structure have to be cast to the other one. Data contained in the structure regard the TCP connection, and all data required to manage a connection based protocol. Not all needed informations are included in the new used structure, some field was directly calculated by CRIU, these calculations have been replicated also in the Volatility plugin starting from CRIU source code. For example the values `inq_len` e `outq_len` which represent the length of input and output buffers, are calculated directly inside the `linux_dump_sock`. The input length is calculated as the difference of the last received sequence number and the sequence number of the last packet extracted from the buffer. The same calculation is done for the output buffer.

```
tcp_sock = sock.cast("tcp_sock")
inq_len = tcp_sock.rcv_nxt - tcp_sock.copied_seq
outq_len = tcp_sock.write_seq - tcp_sock.snd_una
```

A problem encountered in the composition of the `tcp-stream-[inode].img` is related to the input and output buffers. When CRIU performs a checkpoint, it is able to get all data in the input buffer performing a `recv()` system call on the target socket and it writes data in *base64* format in

the `tcp-stream-[inode].img` image file. At restoring time, it performs a `send()` system call in order to fill again data in the input buffer of restored program. When a process is extracted from a memory dump, it is possible to get the input buffer through Volatility but the extracted data are in a rough format, in particular it is not possible to extract only the payload but Volatility extract the entire packets. For the purpose of this thesis, the implemented plugin does not perform the extraction of the buffers and if a process is resurrected during a data transfer, it is possible that some data will be missing. For an investigation process, it could not be a problem, because the process will anyway receive the other part of data. Another problem is encountered during the analysis, it was not possible to find a particular value, called `timestamp`, it is not stored in any kernel structure because it is calculated inside a particular system call. It is an optional addition to the TCP layer to provide information on round-trip times and to help with sequencing.

Problems related to network processes

Network processes can be successfully extracted with the `linux_backtolife` plugin, but some problems has been encountered. CRIU when perform a checkpoint of a process use a particular trick for keeping the TCP connection opened, before killing the process, it set some `iptables` rules in order to do not kill also the connection. The `iptables` command, indeed, performs filtering operations and block the `EOF` string that one peer send to the other before closing the connection. This strategy is very smart because if there are not timeout on the connection, when a process is restored, the `iptables` rules are removed and the TCP connection can be re-established. When a network process is extracted from a memory dump, it is very difficult to find it in a similar state. Because of this, with `linux_backtolife` it is possible to perform a resurrection of a TCP based process, but conditions which permits to completely restore the network connection are very uncommon. The assumption is that most common processes are able to manage inside the connection, re-establishing it if a closed socket is found. Another problem regards the *timestamp* field, during test phases, developers understood that it is a very important field, it can not be set to a 0 value, because the connection could be closed. There is the need of an heuristic for setting that value as estimated, for example using the running time of the process. The last important problem is related to the network configuration, as mentioned before, a network connection is based on addresses, when a process is restored, it is important to replicate the previous network environment in order to make possible the connection re-establishment.

4.3.5 A unix-based process

A Unix domain socket or IPC socket is a communication method that is used for exchanging data between processes executing on the same host operating system. The communication occurs entirely within the operating system kernel. The `AF_UNIX` socket family is used to communicate between processes on the same machine efficiently. Traditionally, UNIX domain sockets can be either unnamed, or bound to a filesystem pathname. Valid socket types in the UNIX domain are: `SOCK_STREAM`, for a stream-oriented socket. `SOCK_DGRAM`, for a datagram-oriented socket that preserves message boundaries. `SOCK_SEQPACKET`, for a sequenced-packet socket that is connection-oriented, preserves message boundaries, and delivers messages in the order that they were sent.

As reported in 2.3.2 `CRIU` does not support every kind of unix socket. In particular named unix sockets with stream/seqpacket options cannot be dumped or restored, as once it is dumped one end, the other one will see EOF on the socket and may close it. Unnamed unix sockets created with socketpair() system call can be dumped and restored. One particular example is the datagram sockets with one-way connection from the client to the server. These connections are thus uni-directional. In this case it is possible to dump a program with the client-side socket and on restore the socket will be reconnected back to the original server. There are some particular workaround in order to dump a client that uses a socket stream. During the dump it is possible to specify an inode of a socket that will be closed. The server will receive a EOF. The client will be dumped. On restore, it is needed to create a new socket and call `CRIU` in order to ask him to inherit it. If a socket stream is used `CRIU` returns the error reported below:

```
> Error (sk-unix.c:707): sk unix: Can't dump half of stream unix connection.
> Error (cr-dump.c:1614): Dumping FAILED.
```

For the experiment has been developed a client-server application that uses stream socket. The program has been used as sample program in order to develop the part of backtolife that extract and generate `CRIU` files about unix socket. The communication between client and server is done on the path `/echo_socket`. The logic of the application is:

1. The server send a message to the client asking for a number.
2. The client send the number received on the stdin.
3. The server send a second message to the client asking for a second number.
4. The client send the second number received on the stdin.
5. The server send back to the client the sum of the numbers received.

During the experiment the memory dump has been done between the third and the fourth step. In order to understand what kind of information are needed for the restore has been done also a `CRIU` dump. Since stream sockets have been used it is needed to tell to `CRIU` to close the connection. For the identification of the socket inode number it is possible to use the following command. Supposing the `PID` of the client equals to 1838:

```
> lsof -p 1838
> client  pid=1838 root 3u  unix 0xffff88007c1feb80 0t0 inode=18622 socket
```

In order to dump the program using `CRIU` launch the command:

```
> criu dump -t 1838 -j --ext-unix-sk=18622
```

A new file called `unixsk.img` is created and new information are needed in files described above. In particular there is a new entry in the `fdinfo-2.img` file that describes the unix socket. The file `unixsk.img` describe all unix sockets used by the dumped program. For each socket there are two entries. Each entry describe one peer of the socket. The program reported in the listing 4.19 uses one socket, therefore in the `unixsk.img` file there are two entries. The `id` is the reference present in the `fdinfo-2.img` file. The `ino` is the inode number, the same that has been passed to `CRIU` during the dump.

In order to extract information about unix socket researchers have been developed a Volatility plugin called `linux_dump_unix_sock` that returns all information about unix sockets. In the listing 4.20 is reported the main part of the plugin in order to show what kind of structures have been used and analysed for the extraction. The listing is not complete and some parts have been cut in order to show just the most significant part. As described in the TCP socket the structure that is used is the `node` object. One important aspect is that for retrieving information about the peer it is needed to get the value in `node.peer` attribute and using the `obj.Object()` method is returned an instance of the class `unix_sock`. The `obj.Object()` class is used because *node.peer* contains just a pointer that points to a particular memory address. The `obj.Object()` method follows the address and interprets those bytes as a `unix_sock` struct.

The `linux_dump_unix_sock` Volatility plugin is used in the `linux_backtolife` Volatility plugin in order to get all information about unix sockets used by a process. This information is formatted using syntax of `unixsk.img` and converted in image file using `CRIT` tool.

One aspect that have to be taken in consideration is the problem related to the state of the process. Since `CRIU` during the dump close the connection to the server, the state of the server is different to the state of the client. When the client is restored, a new connection is created. The server launch a new thread that waits for the first operand. But the client is waiting for the second operand from the stdin. When the operand is sent to the server, the server waits for the second operand but the client waits for the result of the operation. Supposing that a memory dump is given, the analyst should have done a study of the case creating a server that is in a consistent

```
{
            "id": 2,
            "ino": 18622,
            "type": 1,
            "state": 1,
            "flags": "0x2",
            "uflags": "0x0",
            "backlog": 0,
            "peer": 18620,
            "fown": {
                "uid": 0,
                "euid": 0,
                "signum": 0,
                "pid_type": 0,
                "pid": 0
            },
            "opts": {
                "so_sndbuf": 212992,
                "so_rcvbuf": 212992,
                "so_snd_tmo_sec": 0,
                "so_snd_tmo_usec": 0,
                "so_rcv_tmo_sec": 0,
                "so_rcv_tmo_usec": 0,
                "reuseaddr": false,
                "so_priority": 0,
                "so_rcvlowat": 1,
                "so_mark": 0,
                "so_passcred": false,
                "so_passsec": false,
                "so_dontroute": false,
                "so_no_check": false
            },
            "name": ""
}
```


Figure 4.19. One-side unix socket description in *unixsk.img* file.

state for correctly working with the client process. The analyst could restore the server from the dump but he should to lead the server in the consistent state for properly work with the client. After that the client can be restored from the memory dump using BackToLife. Another way could be to restore the client and the server simultaneously from the memory dump. But `CRIU` does not support this kind of event at the same time. This supposition is based on the theory and it has not been tested.

4.3.6 Examples and problems related

The purpose of this section is to report the most significant examples that have been tested using BackToLife. Some of these examples perfectly worked just launching BackToLife, others needs to modify the environment in order to properly work.

```
class linux_dump_unix_sock(linux_pslist.linux_pslist):
    """Dumps infos about opened unix socket of a process"""

    def get_sock_info(self, addr_space_arg, task):
        # ... main part ...
        for filp, fdnum in task.lsof():
            if filp.f_op == sfop or filp.dentry.d_op == dfop:
                iaddr = filp.dentry.d_inode
                skt = self.SOCKET_I(addr_space, iaddr)
                inet_sock = obj.Object("inet_sock",
                offset = skt.sk,
                vm = addr_space)
                if inet_sock.protocol in ("TCP", "UDP", "IP", "HOPOPT"):
                    family = inet_sock.sk.__sk_common.skc_family
                    if family == 1: # AF_UNIX
                        node = obj.Object("unix_sock",
                        offset = inet_sock.sk.v(),
                        vm = addr_space)
                        sock_ino = iaddr.i_ino
                        if node.addr:
                            name_obj = obj.Object("sockaddr_un",
                            offset = node.addr.name.obj_offset,
                            vm = addr_space)
                            name   = str(name_obj.sun_path)
                        else:
                            name = ""
                        # ... parameters extraction ...
                        sock_state = node.sk.__sk_common.skc_state
                        sock_type = node.sk.sk_type
                        sock_backlog = 0
                        sock_sndbuf = node.sk.sk_sndbuf
                        sock_rcvbuf = node.sk.sk_rcvbuf
                        if node.sk.__sk_common.skc_reuse == 0:
                            sock_reuseaddr = False
                        else:
                            sock_reuseaddr = True
                        sock_priority = node.sk.sk_priority
                        sock_rowlat = node.sk.sk_rcvlowat
                        sock_mark = node.sk.sk_mark
                        sock_flags = filp.f_flags
                        sock_id = fdnum -1
                        if name != "":
                            element["name"] =
    base64.b64encode((name+'\0').encode('ascii')) + '\n'
                        # ... peer extraction ...
                        if node.peer:
                            peer = node.peer
                            peerNode = obj.Object("unix_sock",
                            offset=peer.v(),
                            vm=addr_space)
                        # ... continue peer extraction ...
```

Figure 4.20. Unix socket backtolife plugin.

Nano

One of the first sample used in this research is nano. GNU nano is a text editor for Unix-like computing systems or operating environments using a command line interface. Researchers launched nano just typing `nano` in the terminal window and wrote something on the editor. Without saving the document a memory dump is done. Before dumping the memory it is needed to pause the process (CTRL+Z). The list of all the steps for using BackToLife and resurrect the nano on the resurrection machine are reported below:

1. On the target machine launch nano and type something on the editor.
2. On the target machine pause the process in order to save CPU registers in the kernel structures.
3. On the target machine dump the memory using a memory extractor (`LiME` has been used during the research).
4. On the resurrection machine identify the PID of the process using the command:

   ```
   > volatility -f [file_dump] linux_pslist | grep nano
   ```

5. On the resurrection machine lauch the BackToLife Volatility plugin in order to generate files that are used by CRIU for the resurrection.

   ```
   > volatility -f [file_dump] linux_backtolife -p [PID_nano]
   ```

6. On the resurrection machine launch the *prepareMachine.py* script in order to prepare the environment.

   ```
   > ./prepareMachine.py
   ```

7. On the resurrection machine launch the command for resurrecting the process in a new terminal window:

   ```
   > BackToLife.sh -t [PID_nano] -j
   ```

As it is possible to see the editor has been resurrected at the same point of the dump with the same content. A particular things that suggest the correct resurrection is the pointer in the editor.

Lynx

Another program that has been tested using BackToLife is Lynx. Lynx is a highly configurable text-based web browser for use on cursor-addressable character cell terminals. The steps needed for the resurrection are the same presented for the resurrection of nano. As it is possible to test, this resurrection does not properly work. The browser is active and resurrected but it does not work as expected. In particular surfing on the web and following some links Lynx returns the error:

```
> Alert! Can't open temporary file.
```

Researchers discovered that Lynx check the presence of a folder under the path `/tmp/` that is used for storing temporary files. In order to properly works the resurrection is needed to create a directory under `/tmp/`. The name of the directory changes every time that Lynx is launched. In order to find the name of the directory and all files that are used by the process Lynx it is needed to launch Volatility using the `linux_find_file` with the option L.

```
> volatility -f [dump_file] linux_find_file -L | grep lynx
```

```
Volatility Foundation Volatility Framework 2.5
---------------                0x0 /root/.lynx-keymaps
---------------                0x0 /root/.lynxrc
---------------                0x0 /root/lynx.lss
---------------                0x0 /root/.lynx
         665697 0xffff880079fc4c68 /tmp/lynxXXXXIwuRRu
         665712 0xffff880079fcb4a8 /tmp/lynxXXXXIwuRRu/L1877-8626TMP.html.gz
        1707108 0xffff880079fd40c8 /usr/share/locale/en/LC_MESSAGES/lynx.mo
---------------                0x0
    /usr/share/locale/en_US/LC_MESSAGES/lynx.mo
        1458069 0xffff880079f96888 /usr/bin/lynx
---------------                0x0 /usr/sbin/lynx
---------------                0x0 /usr/local/bin/lynx
---------------                0x0 /usr/local/sbin/lynx
        1180599 0xffff880079fd8c68 /etc/lynx-cur
---------------                0x0 /etc/lynx-cur/opaque.lss
---------------                0x0 /etc/lynx-cur/mild-colors.lss
---------------                0x0 /etc/lynx-cur/midnight.lss
---------------                0x0 /etc/lynx-cur/bright-blue.lss
---------------                0x0 /etc/lynx-cur/blue-background.lss
        1180600 0xffff88005a8424a8 /etc/lynx-cur/lynx.lss
---------------                0x0 /etc/lynx-cur/local.cfg
        1180601 0xffff88005a8420c8 /etc/lynx-cur/lynx.cfg
```

Figure 4.21. Output of `linux_find_file` Volatility plugin.

The output of the command is reported in the listing 4.21. Several files are used by the process but in the test that researchers have done no one was necessary except the directory that contains all the files. The directory that is needed for the resurrection is the `/tmp/lynxXXXXIwuRRu` in this particular example. After that the directory is created it is possible to launch BackToLife using the command:

```
> BackToLife.sh -t [PID_lynx] -j
```

Now the program works correctly and it is possible to navigate through the links in the page.

The problem related to folder suggests that if the resurrection does not work it could be useful to make a study on the process. In particular it could be important to understand if the program was working on some files or directories before the dump. Several tests have been done and not in all cases that folder was listed by the `linux_find_file` Volatility plugin. Probably because when the dump has been done the process was not writing or reading in that folder. The analyst should analyse and troubleshoot about the environment before launch the resurrection.

Python shell

Another test that researchers have done is the resurrection of a Python shell starting from a physical memory dump. Researchers opened a terminal with a Python shell on the target system. A function `increment()` is defined. The function increment a global value `counter`. The Python shell is paused and the dump is done on the target machine. On the resurrection machine is launched BackToLife opening a new terminal window with an interactive shell. In order to test that the resurrection has been done correctly it is possible to print the value of the counter. The shell return the value of the counter before the dump on the target system. A correct result is obtained also if the analyst call the function `increment()`. The shell execute the function meaning that the definition has been correctly extracted from the memory dump. This is another example in which the BackToLife works just launching it.

Vim

Another test that researchers have done is the resurrection of Vim. Vim is a highly configurable text editor built to make creating and changing any kind of text very efficient. It is included as `vi` with most UNIX systems and with Apple OS X. Suppose to have a memory dump where Vim was running. Launching BackToLife the resurrection does not work. Researchers discovered that Vim cannot be restored easily. In order to understand the problem researchers launched on a target system Vim. The purpose is to dump Vim using `CRIU` in order to find differences in the generation of the files. If the dump with `CRIU` is launched it fails returning the error:

```
> Error (sk-unix.c:698): sk unix: External socket is used. Consider using
    --ext-unix-sk option.
> Error (cr-dump.c:1614): Dumping FAILED.
```

In order to understand which are the unix sockets used it is possible to launch:

```
> lsof -p $(pidof vim) | grep socket
> vim     1779 root    5u  unix 0xffff88007c264880      0t0   21528 socket
> vim     1779 root    7u  unix 0xffff88007966f800      0t0   21534 socket
```

In this case `Vim` uses two sockets 21528 and 21534. Researchers have tried to directly close the connection during the dump using the command:

```
> criu dump -t $(pidof vim) -j --ext-unix-sk=21528,21534
```

The dump is performed but it is not possible to restore again the process because criu returns:

```
>1719: Error (sk-unix.c:1186): sk unix: External socket found in image.
    Consider using the --ext-unix-sk option to allow restoring it.
>Error (cr-restore.c:2226): Restoring FAILED.
```

This error means that it is necessary to open the sockets before and call `CRIU` passing them in order to inherit them. In order to get information about the socket launch the command:

```
> ss -a --unix -p | grep 21528
> u_str  @/tmp/.ICE-unix/1307 21529  * 21528
    users:(("x-session-manag",pid=1307,fd=17))
> u_str            *          21528  * 21529   users:(("vim",pid=1779,fd=5))
```

The same has been done for the second socket.

```
> u_str  @/tmp/.X11-unix/X0 21535  * 21534   users:(("Xorg",pid=998,fd=42))
> u_str            *        21534  * 21535   users:(("vim",pid=1779,fd=7))
```

As it is possible to see `Vim` has two sockets: one used for communicating with Xorg for changing the title of the window, the other used for communicating with `x-session-manager`. The X session manager is a session management program, a program that can save and restore the current state of a set of running applications. From the point of view of an X session manager, a session is a "state of the desktop" at a given time: a set of windows with their current content. The most recognisable effect of using a session manager is the possibility of logging out from an interactive session and then finding exactly the same windows in the same state when logging in again.

In order to open the socket with the X session manager and Xorg researchers developed a Python script in the listing 4.22 that creates the sockets and calls `CRIU` passing them. The first argument passed through the command line is the random number that is generated for the path of the unix socket used for the communication with the X session manager.

Despite the Python script correctly open the sockets and `CRIU` inherit them after some seconds the X session manager close the connection. Probably because Vim is in a state that is different and not consistent with the state of the X session manager. The socket with Xorg still remains opened. In order to test that the problem was just with the X session manager researchers found a way to compile a new version of Vim removing the functionality of the X session manager. In order to disable the X session manager functionality it is needed to edit the makefile in `vi/src/Makefile` and comment the lines:

```
#!/usr/bin/env python
import socket
import sys
import os

if len(sys.argv) != 3:
    print sys.argv[0] + "x-session-manager-PID"
    exit()

PATH_ICE="/tmp/.ICE-unix/" + sys.argv[1]
PATH_XORG="/tmp/.X11-unix/X0"
sock = sys.argv[2]
# Create a UDS socket for ICE
ice = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
#Create UDS socket for Xorg
xorg = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
try:
    ice.connect(PATH_ICE)
    xorg.connect(PATH_XORG)
except socket.error, msg:
    print "Exception connecting"
    sys.exit(1)

print "Connection Established"
fileno = xorg.fileno()
os.system("criu restore -D /root/vi -j -x --inherit-fd=fd[" + str(fileno) +
    "]:socket:["+sock+"]")
```

Figure 4.22. *UnixConnect.py* script.

```
#CONF_OPT_XSMP = --disable-xsmp
#CONF_OPT_XSMP = --disable-xsmp-interact
```

Recompiling Vim and running it again it is possible to see that now Vim uses just one socket with Xorg. It is possible to create just one socket passing it to `CRIU` using the `UnixConnet.py` script. The resurrection with `CRIU` correctly works. The final test is to dump the RAM of the target machine where it is present an instance of the modified version of Vim. After that is launched the Volatility `linux_backtolife` plugin and the `prepareMachine.py` on the resurrection machine, it is possible to restore the process with `UnixConnect.py` script. The script open a socket with the Xorg on the resurrection machine that is different from the Xorg on the target system. The Vim has been restored and properly works.

Open-SSH client

Another test that researchers have done is the resurrection of Open-SSH client. SSH client is a program for logging into a remote machine and for executing commands on a remote machine. It is intended to replace `rlogin` and `rsh`, and provide secure encrypted communications between two untrusted hosts over an insecure network.

The configuration of this test consists in the use of three systems. One is the target system (VM1) that is a virtual machine running Kali Linux. The second system is the resurrection machine (VM2) that is a virtual machine running Kali Linux with the same kernel of the target machine. The two virtual machines are Kali GNU/Linux 2.0 (sana) 64-bit with kernel 4.0.0-kali1-amd64. The third system is the host (HST) of the two virtual machines. The host is a mac OSX running

OSX El Capitain version 11.4 with a Darwin Kernel Version 15.4.0. It has been created a local network in which each system has own IP address. The local network is composed by the three systems HST, VM1 and VM2.

The test consists in opening a ssh connection from the VM1 to the HST executing some commands. After that is necessary to pause the ssh client and dump the RAM memory of the VM1. After that the dump is completed is necessary to set the IP address of the resurrection machine with the same IP address of the target system. The target system network should be turned off. Subsequently it is necessary to run BackToLife and resurrect the process on the VM2 (resurrection machine). As it is possible to see the resurrection is correctly completed and the connection is still open. The VM2 can control the HST as VM1 before the dump. This is another case in which BackToLife correctly works.

It is important to note that the connection works because the time between the dump and the resurrection is not so long. In a more real case it could happen that pass a long time before the analyst can have the memory image file. In such a case probably the connection will be opened but the ssh server will not receive any commands. This could happen because of the value of the TCP timestamp. If the client uses a too old value the server will start to drop packets and the ssh client will not work properly.

4.4 X-processes

The last objective for this thesis was to perform the resurrection with a process with a graphical user interface. When a good amount of command line programs are been supported, researchers started to analyse the possibility to perform the same operations also with processes with a GUI. In this section the used strategy will be presented, there are not new Volatility plugins, but it is an explanation of what kind of approach was implemented.

4.4.1 CRIU and X-processes

The first problem encountered during the designing phase was that CRIU does not support the graphical applications. They make use of the `X-server` which stores a part of applications state, when this thesis is developed, CRIU is not able to dump informations contained in the graphical server. The implementation of this feature is in the *TODO* list of CRIU developers, but is categorized as a hard implementation that probably requires modifications of `X-server`.

The solution proposed by CRIU developers is to use a `VNC` server as secondary graphical server, and dump an entire process tree containing also this server. It is supported by CRIU, and it is the only manner for checkpointing and restore graphical applications. `Xvnc` is the Unix `VNC` server, it is based on a standard X server, applications can use it as a normal `X-display`, but they will actually appear on any connected `VNC viewers`. A viewer is an application which talk with the real `X-Server` with the `X-Protocol` and with `Xvnc` with a particular protocol. Its objective is to give to users the possibility to watch end interact with applications. The architecture used by `VNC` is displayed in Figure 4.23

The CRIU approach is to use these tools to perform the dump of one, or multiple applications with a GUI. Firstly a particular `C` program is used: `newns.c`. Its task is to start the `Xvnc` server as child process, it is available on *CRIU Project* page [71]. Specifically, the user has to create a script (`vnc-server.sh`) which will be launched by the previously mentioned program for starting `Xvnc` with particular settings, as illustrated in Figure 4.24.

Then it is possible to start the process tree with the command:

```
./newns ./vnc_server.sh icewm
```

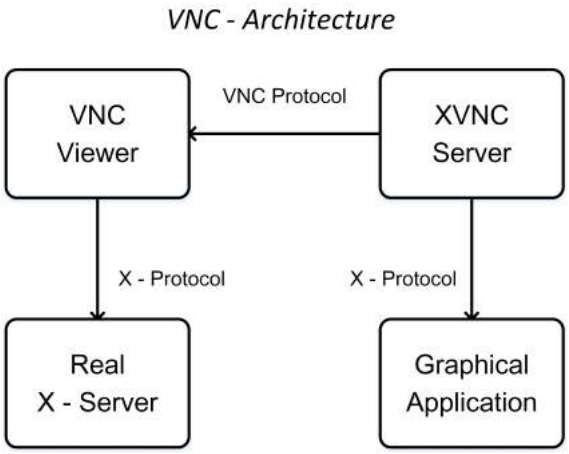

Figure 4.23. VNC architecture.

```
#!/bin/bash
set -m
Xvnc :25 -v -geometry 800x600 -i 0.0.0.0 -SecurityTypes none &
pid=$!
trap "kill $pid; wait" EXIT
sleep 3
DISPLAY=:25 $@
```

Figure 4.24. `vnc-server.sh` script.

The `icewm` indicates which *window-manager* will be used inside the `Xvnc` server. A *window-manager* is a particular component which manage the position and appearence of windows in a graphical desktop. `Icewm` is a simple manager, it is not the most used in UNIX systems but it is good for the purpose of this thesis because its smallness and simplicity. When the previous command is run, in the list of active process, a tree like the following will appear:

```
1654 ?  Ss  0:00 ./newns ./vnc-server.sh icewm
1655 ?  Sl  0:00  \_ Xvnc :25 -v -geometry 800x600 -SecurityTypes none
1663 ?  R   0:00  \_ icewm
```

When the showed process tree is started, it is possible to connect a `vncviewer` to the server, in order to display the desktop, each application which will be started inside the viewer desktop will be a child process of the `icewm` instance. This trick permits to perform a checkpoint of the entire tree, starting from the `newns` process. When the restoration of the checkpointed tree will be performed, the user will be able to display again the graphical application reconnecting a viewer to the `Xvnc` server. This is the only way for performing checkpoint and restore of application with graphical components.

4.4.2 The implementation model

For the purpose of restoring a graphical process developers thought about a particular strategy in order to use CRIU without modify it. Specifically it was implemented a model in order to use `Xvnc` as `X-server` in the restoring phase.

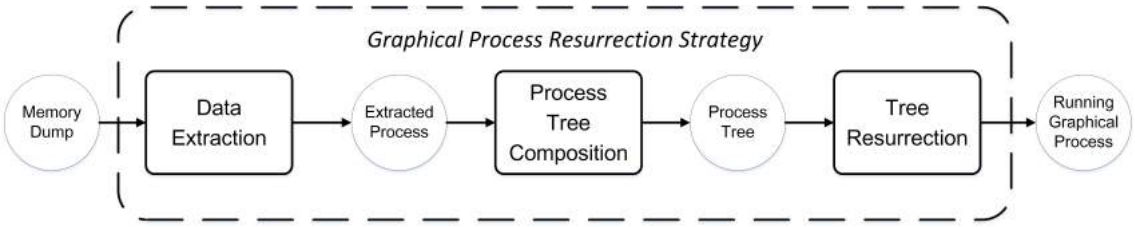

Figure 4.25. Proposed strategy for the resurrection of a graphical process.

From a forensic point of view, the actions which have to be performed are the same presented in the previous sections, the implementation instead, changes in particular for the resurrection process. Changes are needed because it is necessary to adapt the forensic output, extracted using Volatility, to a resurrection of a graphical process with CRIU. As illustrated in Figure 4.25, a new phase is inserted in the implementation model previously presented in Figure 4.2. This new phase is the connection point between the forensic extraction output and the way supported by CRIU for restoring a graphical process. This particular stage is called *tree composition*, its objective is to insert an extracted process into an existent checkpoint. Specifically, a process tree which contains `Xvnc` and `icewm` is checkpointed using CRIU, then the process extracted from the memory dump is inserted in the tree as child of `icewm`. The insertion is performed through image files modification, in order to build a valid and consistent checkpoint. The memory dump is analysed and the target process is selected, through `linux_backtolife` it is extracted and images file are created. Later, extracted files are modified with the checkpointed tree in order to recreate an environment for the graphical target process similar to the previous one, substituting the `X-server` with `Xvnc`. Specifically, `UNIX_SOCKET` connection have to be re-established, many references have to be setted in order to generate a valid checkpoint which can be resurrected by CRIU. A high level of what is performed in this new phase is illustrated in Figure 4.26.

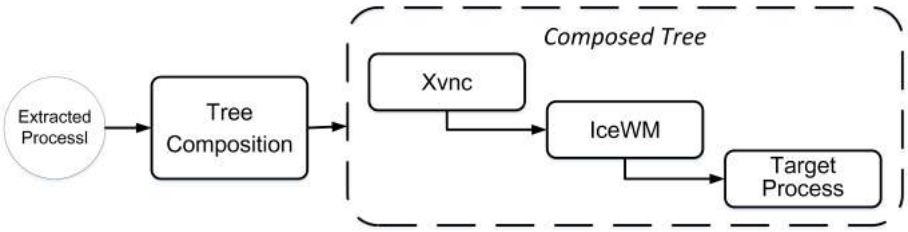

Figure 4.26. Proposed strategy for the resurrection of a process.

Supporting tools

As mentioned before, graphical applications communicate with the `X-Server` with a dedicated protocol through a `UNIX_SOCKET`. In chapter 3 section 2.3.2 it was presented how processes which use these kinds of sockets can be checkpointed with CRIU. Specifically, `SOCK_STREAM` sockets external to the process tree have to be specified to CRIU which interrupt them during the checkpoint. When an entire tree of processes have to be checkpointed, it could be very difficult to find all external sockets, this is the reason which pushed researchers to develop a particular tool able to draw a graph representing a process tree, with hierarchy and sockets connections. The tool is entirely written in Python using the `pygraphviz` library for generating a `PNG` file. The script `processTree.py`, accept only one parameter, the `PID` of the root process, then it analyses all nodes of the tree generating the hirarchy. For each process, unix sockets are collected and stored with relative peers. When the analysis is completed, results are written on terminal, and the graph is recursively drawn. Main parts of the implemented tools are illustrated in Figure 4.27.

The output graph contains various informations. Firstly the hierarchy is shown, each node contains the PID and its common name. Unix connection are displayed through unidirectional connections between nodes. Extern processes are highlighted in order to understand if a socket

```
def visualize(data, processes, root):
    G = pgv.AGraph(directed=False, strict=False)
    G.node_attr['shape']='box'
    for pid, p in processes.iteritems():
        ...


    G.draw("graph.png")


if __name__ == "__main__":


    #Get Root process from PID
    root = psutil.Process(int(root_pid))


    #Get Children
    processes[root.pid] = root
    for c in root.children(True):
        processes[c.pid] = c


    #Get UNIX connections
    for pid,c in processes.iteritems():


        for i in os.popen("lsof -e /run/user/1000/gvfs -p " + str(pid) + " |
    grep unix").read():
            ...


    #Search for peers
    for inode,pid in unix_sock.iteritems():

        for i in os.popen("ss -xp | grep " + str(inode)).read():
           ...


    #Draw Graph
    visualize(data, processes, root)
    exit(0)
```

Figure 4.27. Basic structure of `processTree.py`.

connection is external to the target tree. On terminal the list of all processes is shown, for each one of them informations about potential socket connections are displayed.

4.4.3 Implementation

In this subsection it will be explained how the implementation model was tested, and in particular, how researchers tested the resurrection phase. Specifically, it will be explained what is needed for

inserting an application in a checkpointed tree, and how it is possible to establish unix connections with the `X-Server`.

Development environment

In order to develop and test the proposed idea, some software is needed. In order to build the environment it is important to install the *window manager* `icewm`, also `Xvnc` and `vncviewer` are needed. The process tree illustrated in subsection 4.4.1 is built and checkpointed many times, in order to test if and how is possible to modify images files for changing the checkpointed process tree. Each modified checkpoint have been tested in order to understand if it is valid and if it can be restored by CRIU.

The forwarder process

As described in the previous section, the main idea is to start from a set of files containing a dumped environment with a Xvnc server. Subsequently, the analyst should extract an X-process starting from a physical memory dump and implant the process into the process tree in the dumped files. After that the files will contain a new environment and the process tree can resurrect using `CRIU`. Since that the environment of the memory dump is completely different from the environment of the dumped file, it is needed a contact point (layer) between the two environments. The purpose of this section is to describe this layer that it is called `forwarder`. The idea is to modify the dumped file in order that when the process tree will be resurrected the target process thinks to be in the old environment. The forwarder it is needed because in the old environment the target process communicates directly with the X-server but in the dumped file the Xvnc server does not contain the same information of the X-server in the old environment. The function of the `forwarder` is to put in communication the target process with the Xvnc server in order to properly work the communication between the processes.

The first step towards the implementation of the `forwarder` is to create a basic process that is just able to forward the bytes stream from the target process to the Xvnc server and vice-versa. Researchers have done several tests in order to make it works this model. This development model consists of several steps:

- Creating the environment starting Xvnc server and icewm window manager: in this step researchers created the environment that will host the target process. This environment will be dumped using `CRIU` and will be the starting point of the research.
- Starting the forwarder: in order to have the forwarder in the dumped files in it needed to launch it and dump it with the process tree.
- Starting a target process: this target process is useful to simulate the target process that will arrive from the physical memory dump used by the analyst.
- Dump the environment: in this step researchers dump all the process tree in order to have a set of files containing all the environment used for the resurrection. In this step it is fundamental the use of the tool for the generation of the communication graph. The uses of the tool suggest which are the socket to close for dumping the tree with `CRIU`.
- Modification of files: since the target process selected in the previous step communicate with the Xvnc server directly, it is needed to modify some files in the dumped files in order to put in communication the target process with the *forwarder* and the forwarder with Xvnc server.
- Resurrection: the last step is to use `CRIU` for the resurrection of the tree. After the resurrection the process will communicate with the *forwarder* and the `forwarder` will forward the byte stream directly to the Xvnc server and vice-versa.

The Figure 4.28 shows the environment before the dump. The process socat is used to open a connection with the `forwarder` in order to have an open socket that will be used by the target

process in order to communicate with the forwarder. The forwarder open a connection with the Xvnc in order to exploit the creation of the socket. This socket will be subsequently modified. The creation of the socket with Xvnc it is needed otherwise the process will not be able to forward bytes to Xvnc.

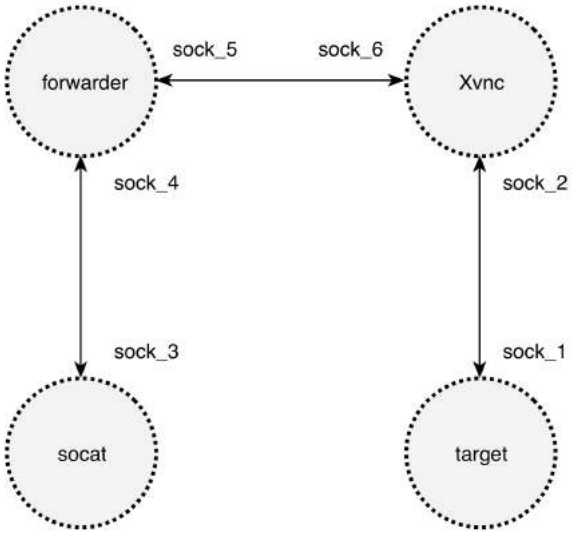

Figure 4.28. Environment before the dump.

The Figure 4.29 shows the environment after that the files generated by `CRIU` are modified. The modification consists in modifying the file that describes sockets. In particular is modified the *unixsk.img* binary `CRIU` file. The modification consists in the following steps:

- Close the socket *sock_3* removing the entry in the *unixsk.img* file.
- Close the socket *sock_6* removing the entry in the *unixsk.img* file.
- Change the peer of the socket *sock_1* with the socket *sock_4* and change the peer of the socket *sock_4* with the socket *sock_1*.
- Change the peer of the socket *sock_5* with the socket *sock_2* and change the peer of the socket *sock_2* with the socket *sock_5*.

After the modification of the `unixsk.img` file is done, it is needed to resurrect the tree using `CRIU`. The target process thinks to communicate directly with the Xvnc server but the communication has been modified and all the bytes sent to Xvnc are sent to the `forwarder` and the `forwarder` will send them to Xvnc. The same is done for the Xvnc server. All bytes sent from Xvnc to the target process are sent to the `forwarder` that will forward the bytes to the target process.

Researchers have done several tests in order to test this model. In particular the target process that is selected for these tests was Mozilla Firefox. The Figure 4.30 shows all the processes involved in the practical example with Firefox. The black arrows represent the hierarchy of processes. The red arrows represent the unix sockets between processes. The blue arrow identifies the unix socket that goes out of the environment. That particular socket should be closed before the dump, otherwise `CRIU` will not dump the tree. Several test have been done and the closing of that socket does not influence the functionality and the stability of the system.

The code of the first version of the `forwarder` is reported in the listing 4.31. The `threaded_function()` is the function that forward bytes from the *src* to the *dst*. The function is executed in two different

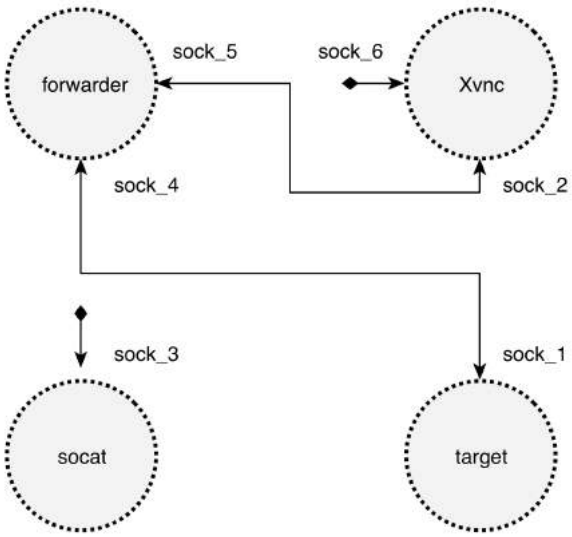

Figure 4.29. Environment after the modification of *unixsk.img* file.

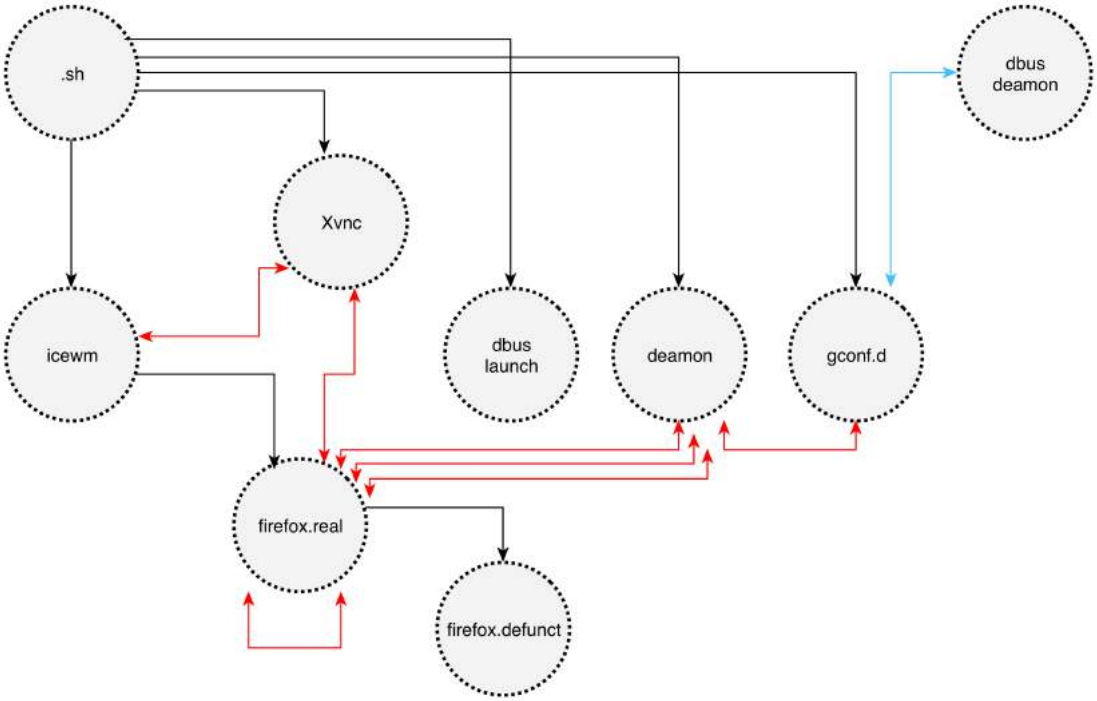

Figure 4.30. Hierarchy and unix socket of the tested environment.

thread. One thread is responsible for the forward from the target process to the Xvnc server. The other thread is responsible for the communication from the Xvnc server to the target process.

```
#!/usr/bin/env python

import socket
import sys
import os
import struct
from threading import Thread

PATH_XORG="/tmp/.X11-unix/X25"
server_address = '/tmp/.forwarder'

def threaded_function(src, dst, srcName):

    while True:
        data = (src).recv(1)
        dst.sendall(data)


# Make sure the socket does not already exist
try:
    os.unlink(server_address)
except OSError:
    if os.path.exists(server_address):
        raise
sock = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)

# Bind the socket to the port
print >>sys.stderr, 'starting up on %s' % server_address
sock.bind(server_address)

# Listen for incoming connections
sock.listen(1)
while True:
    # Wait for a connection
    print >>sys.stderr, 'waiting for a connection'
    connection, client_address = sock.accept()
    xorg = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
    xorg.connect(PATH_XORG)
    thread = Thread(target = threaded_function, args = (connection, xorg,
    "Firefox"))
    thread.start()
    threaded_function(xorg, connection, "Xvnc")
```

Figure 4.31. The *forwarder*.

Improved forwarder process

When researchers was sure that it is possible to change a checkpoint state modifying unix connections, they tried to understand if it is possible to force a graphical application to draw in an existent window, different from the original one. It must happen without that neither `Xvnc` nor the application knows that. It is a necessary test, because in the mentioned resurrection model, the extracted application will be restored in a completely new environment, with a new `X-Server` and a new window to draw.

As previously illustrated, the forwarder process can be completely transparent to the others

involved in the communication, and the idea is to make use of it not only as a forwarder but also for modifying received packets before forwarding them. The followed model is the same used for the implementation of the simple forwarder, but in order to have a window to draw, different from the original one, a second graphical process is added to the process tree. The idea consists in creating the process tree, checkpointing it with CRIU, modifying images files and restoring the entire tree. After the restoration, the expected result is that the target process draws its graphics in the window of the other graphical process. The new environment that have to be dumped is illustrated in Figure 4.32, it contains one more graphical process, only for the purpose to create a graphical window which can be used by the target process during the restoration. When the modified checkpoint is restored, the environment is similar to the one in Figure 4.29, for firsts tests the second graphical process is not modified.

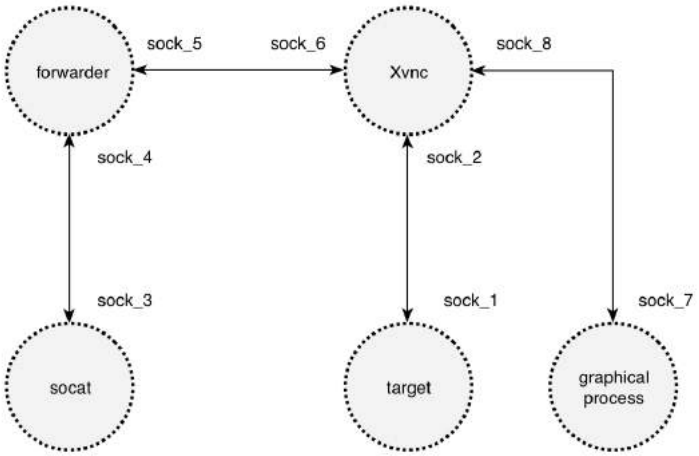

Figure 4.32. Environment before the dump with a second graphical process.

In order to force the target process in another window, different from the original one, the first thought solution is to change the `window-id`, it is a univocal number used by processes for identifying a certain window. Specifically, when a graphical process wants to draw in a specific window, it has to send the related `window-id` in the *draw request*. The new implementation of the forwarder, has to not only forward packets, but also to modifying them changing the `window-id` with the id of the new target window. The ids of graphical windows, were taken before the dump using the unix program `xwininfo`, which writes on terminal the id of a target window. The `threaded_function` of the forwarder is modified in order to search the old `window-id` and substitute it with the new one. Data are received in blocks of 32 bytes, and each occurrence of the old id is changed. Then the packet is forwarded to the other peer. Researchers employ effort for code optimization if this function, because the forwarder must be very fast in order to keep the unix connection alive. A piece of the modified forwarder is shown in figure Figure 4.33. The function receives now as parameter also the old and new `window-id`.

After the checkpoint modifications as for the simple implementation of the forwarder, the tree is restored by CRIU. In the first time the result seemed very positive, the target process was really drawing all its graphics inside the other window. The problem is encountered when researchers tried to interact with the application, when the mouse is clicked on the window, the original process drawn again all the window and the target process can not be visualized any more. It can be a good output of the test in order to give the possibility to an analyst of visualizing what a graphical process was performing, but for the purpose of this thesis, researchers wants an interactive restored graphical process. The problem is that target process draw its graphics on the target window, but when there is an interaction on the window, the `X-server` notifies the process which owns that window which redraw its own graphics again, removing the old one. The obtained results brought developers to think that the original owner of the target process have to be disconnected from the

```
def threaded_function(src, dst, toFind, toModify):


    while True:
            data = (src).recv(32)


            if toFind in data or toFindChild in data:


                tofindB = False
                tofindchildB = False
                startI = 0
                startIchild = 0


                if toFind in data:
                    startI = data.index(toFind)
                    tofindB = True


                for i in range(0, len(data)):
                    if i in range(startI, startI+4) and tofindB:
                        tosend = toModify[i-startI]
                        dst.sendall(tosend)
                    else:
                        dst.sendall(data[i])
            else:
                dst.sendall(data)
```

Figure 4.33. `threaded_function` with `window-id` changes.

graphical server, and that each unix packet direct to it have to be forwarded to the target extracted process. In order to perform these operations, the target process must talks with the `X-server` through the socket owned by the disconnected applications, it requires new modifications to the original checkpoint, and the restored process tree is illustrated in Figure 4.34.

After some attempt, it was immediately clear that modifications to the `window-id` is not enough for creating a valid connection. Studying more the `x-protocol`, another important field was discovered, it is the `sequence-number`. It is a control number which counts a certain kind of requests and it is incremented when the client performs a specific request to the `X-server`. In order to study how packets are composed and how sequence number are incremented, a more complex forwarder was developed. Developers added to it the ability of dump on a file what is received and sent through it. Simultaneously it was able also to log operations performed in order to give the possibility to researchers of analyse what happens for think about a solution. This analysis can be represented with four different phases:

- The process tree presented in Figure 4.32 was created and checkpointed with CRIU.
- Starting from the original checkpoint two different versions have been created, the first which simply forward packets not modifying it, the second which is able to modify packets forwarding them to the unix socket relative to the target window.
- Analysis of the first checkpoint output in order to find the last used `sequence-number`.
- Restore of the second checkpoint for changing `sequence-number` and `window-id`.

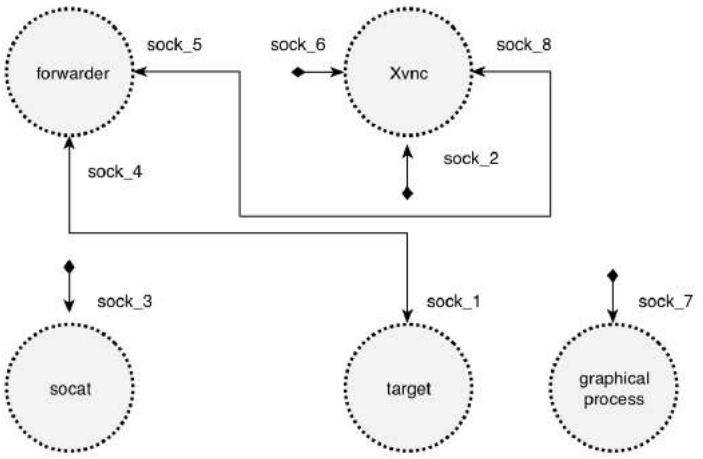

Figure 4.34. Environment after the dump with a second graphical process.

These phases, and in particular the last one, are very difficult to be performed, they have to be executed on the same checkpoint, but the sequence number is known only after analysis and it has to be used by the forwarder process in order to modify the original one. Researchers developed a forwarder able to read at runtime the new `sequence-number` in order to modify it when needed. After many tests researchers understand that it is very difficult to change the number in the right moment, and every time the `X-server` received a wrong number the connections have been closed. In order to increment the number correctly it is needed a more specific comprehension of what each packet contains, performing a real parsing of `x-protocol` packets.

Future works

As reported in the previous section, the `forwarder` should be able to comprehend every kind of packets sent by the `X-Server` to the target process and vice-versa. Instead of filtering just some packets as researchers have done, it could be a better approach to develop a `forwarder` that interpret the X-protocol in its completeness.

As mentioned in the previous section, just filtering and changing the `window-id` produced an important result because the process was able to draw its graphical interface on another window. If a complete X-protocol has been developed, the interaction with the user would be possible. In such a case it could be possible to claim that the target process can communicate with another `X-Server`, completely different from the original `X-Server` that was present on the target machine. The next needed step is the application of memory forensics techniques in order to extract all the information and the memory areas about the target process. As researchers have completely done for processes without graphical user interface, it is needed to compose all data in such a way can be interpret by `CRIU`. After that the extraction is completed it is needed to substitute files about target process into the set of files pre-created which contain the environment with the Xvnc and icewm.

Subsequently, it would be possible to resurrect the process with GUI in the analyst environment using `CRIU` and the set of files pre-created. The process will communicate with the `forwarder` and the `forwarder` will translate the packets to the `X-Server` in the new environment.

Summarising the steps needed for a complete resurrection of a process with graphical user interface:

- Dump an entire process tree with a pre-created environment with Xvnc, icewm and a process that will be replaced with the target process.

- Modification of the `forwarder` with a complete implementation of the X-protocol.
- Application of memory forensics techniques used in BackToLife tools in order to extract memory areas of the target process starting from the physical memory dump.
- Composition of extracted data and implantation of the process in the pre-created subset of files containing the default process tree with Xvnc and icewm.
- Modification of the files and connection with the *forwarder*.
- Resurrection of the tree using `CRIU`.

The Figure 4.35 shows that the set of files created dumping an entire process tree contains an environment with Xvnc, icewm and a process that will be replaced with the target process extracted from the physical memory dump.

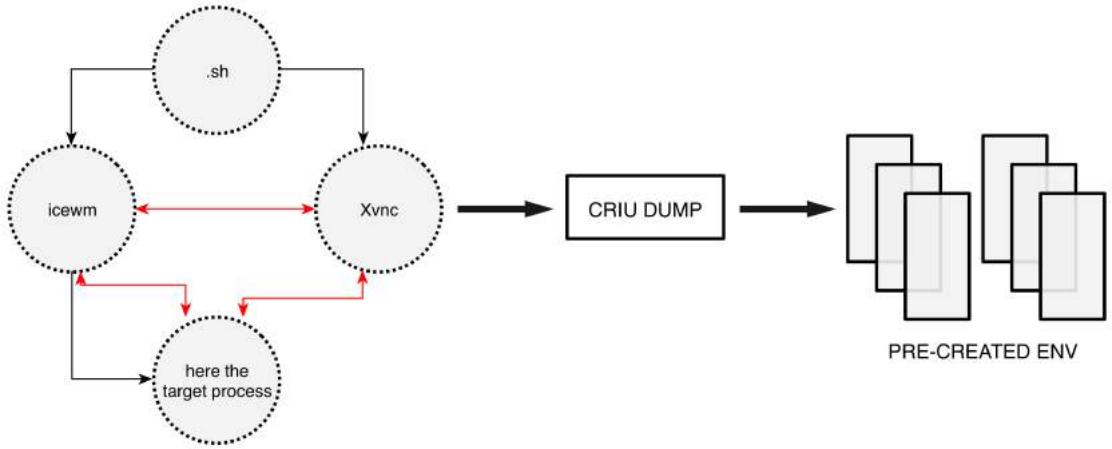

Figure 4.35. Pre-created environment.

The Figure 4.36 shows that for a complete resurrection of a process with graphical user interface the idea is to extract the target process memory areas from the physical memory dump using the BackToLife tool. After that, will be needed to compose the data in an interpretable format by `CRIU`. The set of pre-created files will be modified in order to have a process tree containing the target process.

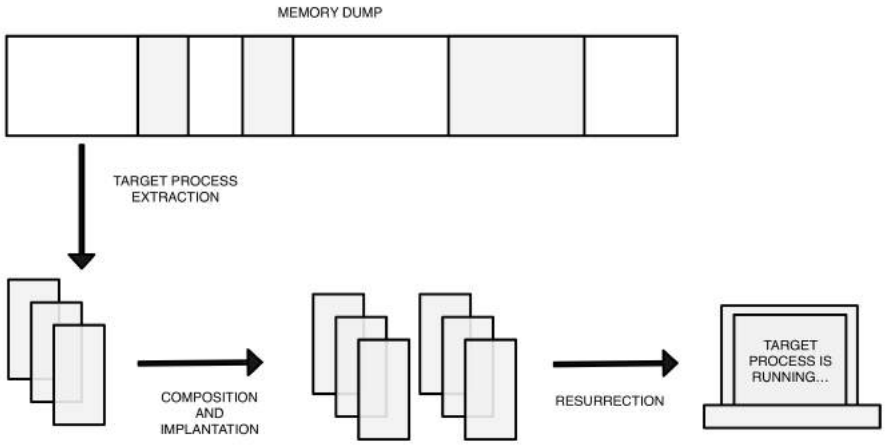

Figure 4.36. Procedure for the resurrection.

The processes tree that will be resurrected using `CRIU` is depicted in the Figure 4.37. As it is possible to see the target process after the resurrection will communicate with the *forwarder*. The

forwarder will modify all packets in order to properly work the communication between the target process and the `X-Server`. The black arrows represent the hierarchy of the processes. The red arrows represent the unix sockets used for the communication between processes.

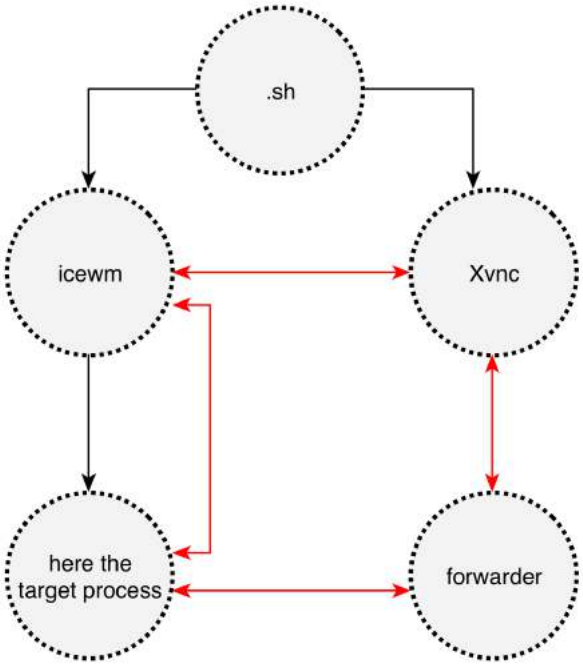

Figure 4.37. The processes tree after the resurrection.

Chapter 5

Conclusions

The project BackToLife, presented in this thesis, aims to improve memory forensic techniques. BackToLife is the first forensic project that combines memory analysis techniques with check pointing technologies.

The analyst is able to analyse processes in volatile memory of a target computer. Using BackToLife the static memory analysis turns into live analysis.

BackToLife is able to revive several kinds of processes starting from a physical memory dump. The analyst has just to select a target process. After that BackToLife has analysed the memory, the target process is extracted from the dump and resurrected on the analyst's computer in order to perform the investigation.

Several tests have been made using BackToLife with different kind of processes. Single-thread programs, multi-thread programs, TCP socket-based and Unix socket-based programs can be resurrected using BackToLife. The resurrection is also possible for programs that use files and communicate with other processes through signals. Main kinds of command line programs can be resurrected using this project.

The research has gone beyond command-line-based processes resurrection. The study also covered programs with graphical user interface. In this research has been developed a model that aims to resurrect the Firefox browser into the analyst machine. Firefox is an X-process which communicate with the X-server. Since `CRIU` currently does not support the dump and restore of X-processes, the model became more complex. The structure of the model is to implant the Firefox memory areas into a set of files that are sent to `CRIU` for the resurrection. The resurrection of the X-process is possible because of a workaround that exploits an X-server and a window manager in the user address space. Since the Firefox extracted from the dump was living in an environment that is different from the resurrection environment, a bridge program is needed. In particular, has been developed a forwarder that is able to filter the communication between the X-client and the X-server in a transparent way. The resurrection of the browser has not been completed because of the complexity of the X-protocol. The forwarder should completely understand the dialogue between the client and the server, filtering data in a proper manner. A possible future development could delve into this part. Another possible evolution could be waiting the `CRIU` implementation for X-based processes. In this way the model will be easier because the process extracted from the memory dump does not need to be implanted into a complex process tree.

Since BackToLife is the first project in this field, several developments are possible. The project has been developed on Linux. A porting on different operating systems could spread this kind of technique also on Mac OS X and Windows machines. Since smart phones are widely used in the world, another interesting research could be a porting on mobile operating systems. Since BackToLife assumes that the analyst's machine has the same kernel and the same libraries of the target machine, another possible development could regard the possibility to run BackToLife on a machine with different characteristics from the target.

The purpose of this research was to understand the feasibility of resuscitating a process from a memory dump onto another machine with similar features of the investigated machine. Results

presented in this thesis proved that is possible to bring processes back to life allowing memory analysis with a different approach.

Appendix A

User Manual

The output of this thesis is a pool of file which can be used for resurrecting processes. In this section it will be explained how these files have to be used for reaching the purpose. Specifically there are two different categories of files, the first contains the scripts and volatility plugins for the resurrection, the second includes tools for development and continue the research.

A.1 Environment setup

In order to build the environment it is necessary to start from a physical or virtual machine running a Linux operating system. During the development of this thesis, a *Kali Linux 2.0* with kernel version 4.0 and a *Debian Jessie* with kernel 3.16 were used and deeply tested. When the operating system is setted up, needed tools can be installed on the system. This manual explains how to set up the environment on a *Kali* system, but for other distributions steps are very similar.

In order to perform the resurrection process both `CRIU` and `Volatility` have to be installed on the machine, they can be downloaded from their official repositories, and installed through following steps. Firstly dependences have to be installed with the right version, therefore packages sources have to be correctly setted on the system to be sure to get access to all available versions of each package.

```
#Setting up packages repository

>> echo "deb http://old.kali.org/kali sana main non-free contrib" >>
    /etc/apt/sources.list

>> echo "deb-src http://old.kali.org/kali sana main non-free contrib" >>
    /etc/apt/sources.list

>> apt-get update

>> apt-get upgrade
```

Regarding `CRIU`, the version 2.5 should be installed in order to guarantee the compatibility with `backtolife plugins`. In newer version images file format could change, and it is possible that generated images do not work. This version needs a particular set of packages, and with the following command they will be installed on the machine at the same version used for development of this project. If on the system there are installed newer versions of target packages, they will be downgraded.

```
# Installing dependencies

>> apt-get install -y --allow-downgrades \
```

```
        build-essential \
        libprotobuf-dev=2.6.1-1 \
        libprotobuf-c1=1.0.2-1 \
        libprotobuf-c-dev=1.0.2-1 \
        protobuf-c-compiler=1.0.2-1 \
        protobuf-compiler=2.6.1-1 \
        python-protobuf=2.6.1-1 \
        libnet1-dev=1.1.6+dfsg-3 \
        pkg-config=0.28-1 \
        libnl-3-200=3.2.24-2 \
        libnl-3-dev=3.2.24-2 \
        python-ipaddr=2.1.11-2 \
        libcap2=1:2.24-8 \
        libcap-dev=1:2.24-8 \
        libaio1=0.3.110-1 \
        libaio-dev=0.3.110-1 \
        python-yaml=3.11-2
```

When all packages are successfully installed, it is possible to proceed installing `CRIU`.

```
# In root (/) directory
# Download of CRIU sources in version 2.5

>> wget https://github.com/xemul/criu/archive/v2.5.tar.gz

# Sources extraction
>> tar zxvf v2.5.tar.gz

# Moving in CRIU directory and compiling it
>> cd /criu-2.5 && make
```

At the end of the procedure, `CRIU` and its companion tool `crit` will be compiled and available on the machine. Then, it is possible to move on the *Volatility Framework* configuration. It is all written in `python` and specifically, it requires the version 2.7. In the thesis development the version 2.6 of this framework was used and tested but all version should be supported.

```
# Installing python and other dependencies
>> apt-get install -y \
        dwarfdump \
        pcregrep \
        libpcre++-dev \
        python-dev \
        python-crypto=2.6.1-5+b2 \
        python-distorm3=3.3-1kali2

# Download of Volatility on root directory and sources extraction
>> wget https://github.com/volatilityfoundation/volatility/archive/2.6.tar.gz
>> tar zxvf 2.6.tar.gz

# Main volatility script has to be executable
>> chmod +x /volatility-2.6/vol.py
```

When both tools are configured on the analysis machine, it is necessary to configure the `$PATH` environment variable in order to use all the previously installed tool from every path. It can be done for example with the following command:

```
# Configuring PATH
>> echo "PATH=$PATH:/volatility-2.6:/criu-2.5/criu:/criu-2.5/crit" >>
    ~/.bashrc
```

```
# Setting alias for vol.py script.
>> echo "alias volatility=\"vol.py\"" >> ~/.bashrc
```

Then a community volatility plugin has to be downloaded from a *github* repository, it is needed for extracting the `ELF` file of the target process from a memory dump [72]. This particular plugin needs a `bash` global variable called `LD_BIND_NOW` which has to be setted to the value 1. It is necessary for permitting the execution of the extracted `ELF` file as declared by the author of the `ElfDump` plugin. The plugin can be configured as follows:

```
# Cloning the repository
>> git clone https://github.com/ecanzonieri/ElfDump

# Installing plugin inside volatility directory
>> mv ElfDump/linux_elf_dump /volatility-2.6/volatility/plugins/

# Setting global variable
>> echo "LD_BIND_NOW=1" >> ~/.bashrc
```

When all tools are setted up, it is possible to proceed with configuration of *BackToLife tools*. These tools can be downloaded directly from the *github* repository. They require two python packages, also `volatility` plugins have to be inserted in the right installation folder, in the path `volatility/plugins`. It is possible to configure these tools using the following commands:

```
# Installing dependencies
>> apt-get install -y \
        python-psutil \
        python-pygraphviz \

# Download of source code
>> wget https://github.com/lukdog/backtolife/archive/v2.0.tar.gz

# Extracting it in root directory
>> tar zxvf v2.0.tar.gz

# Setting PATH variable for using backtolife tools
>> echo "PATH=$PATH:/backtolife-2.0" >> ~/.bashrc

# Installing developed volatility plugins
>> mv /backtolife-2.0/volPlugins/* /volatility-2.6/volatility/plugins
```

When all is setted up, it is possible to check if all is correctly configured running following commands:

```
# Check volatility installation
>> volatility --info | grep linux

# It is important to check if installed plugin are included in the output of
    previous command

# Check criu installation
>> criu --version
```

A.2 How to perform a resurrection

When the setup of plugins is completed, the volatility profile of the target machine have to be generated and moved in `volatility/plugins/overlay/linux`. For the generation of the target

profile, the modified `module.c` file has to be used in order to generate all needed VTYpes, it can be found in `BackToLife/linux_module`.

```
# Moving modified module in right folder
>> cp /backtolife-2.0/linux_module/module.c /volatility/tools/linux

# Compiling module file
>> cd /volatility/tools/linux && make

# Generating the profile
>> zip /volatility/volatility/plugins/overlays/linux/kali.zip
    /volatility/tools/linux/module.dwarf /boot/System.map-4.0.0-generic
```

Previous command can be run on the development machine, but it is important to remember that each profile is related to a particular operating system or kernel version. Profiles have to be generated on target machine or on a identical installation.

When all is setted up, it is possible to perform the resurrection of a process through a simple sequence of commands:

```
# Target process identification
>> volatility --profile=[profile_name] -f [dump_file] linux_pslist | grep
    [target_process]

# Process extraction
>> volatility --profile=[profile_name] -f [dump_file] linux_backtolife -p
    [PID]

# Image files generation
>> prepareMachine.sh

# Process resurrection
>> criu restore [params]
```

At the end of the last command the process should be restored and it is possible to interact with it using the command line. Another possibility is to use the `BackToLife.sh` instead of directly calling `CRIU` for the restoring phase. This script accepts the same parameters of `CRIU` but perform the restoration of the process in a new terminal window, it supports only `gnome-based` operating systems. Sometimes the process which have to be restored makes use of libraries not present on the analyst's machine. In this situation, the `prepareMachine.sh` script will fail and the restoration can not be performed. It is possible to solve the problem installing the right version of the target libraries listed in the `procfiles.json` file. Desired libraries can be installed, setting the right package repository and using the following commands:

```
# Checking if desired version is in the setted repository
>> apt-cache policy [package_name]

# Installing desired version
>> apt-get install [package_name]=version
```

A.3 Development tools

This category of tools includes all scripts developed for this research. In order to use all of them, the `BackToLife` path has to be included in the environment variable `$PATH`. Most of them can be easily launched for understanding what parameters are needed, and in this section the most useful will be explained. Development tools could be divided in groups based on the context of use. Main scripts were written for searching patterns, or working on a raw file, for example in a memory dump or a section of it.

- `findBytes.py`: searches all occurrences of a sequence of bytes in hexadecimal format in a raw file, the output is a list of addresses where the pattern is found. It receives in input the raw file and the sequence of bytes.
- `pageSeparator.py`: splits a raw file in many pages raw file with the size of a page. It receives as parameter the raw file.
- `compareBinary.sh`: compares two different binary files printing the differences.

Other scripts were written for working faster with `CRIU` file images.

- `convertImgJson.sh`: it has not parameters and it is able to covert all images from binary to `json` format using `crit`.
- `diffCriu.sh`: it takes as parameter two different paths containing two `CRIU` checkpoint converted in `json` format. It prints all differences of the two checkpoint.
- `findPatternCrit.sh`: searches for a pattern in all `json` image files in the current directory.

Remaining tools were developed during the research for the resurrection of graphical processes. In this group there is the `forwarderUnix.py` which is the implemented forwarder for the `X` protocol, or the `processTree.py` which was explained in the previous sections.

Bibliography

[1] A. Aljaedi, D. Lindskog, P. Zavarsky, R. Ruhl, and F. Almari, "Comparative analysis of volatile memory forensics: live response vs. memory imaging", Privacy, Security, Risk and Trust (PASSAT) and 2011 IEEE Third Inernational Conference on Social Computing (SocialCom), 2011 IEEE Third International Conference on, Boston (MA), 9–11 October 2011, pp. 1253–1258, DOI 10.1109/PASSAT/SocialCom.2011.68

[2] F. Adelstein, "Live forensics: diagnosing your system without killing it first", Communications of the ACM, vol. 49, February 2006, pp. 63–66, DOI 10.1145/1113034.1113070

[3] S. Mrdovic, A. Huseinovic, and E. Zajko, "Combining static and live digital forensic analysis in virtual environment", Information, Communication and Automation Technologies, 2009. ICAT 2009. XXII International Symposium on, Sarajevo (Bosnia), 29–31 October 2009, pp. 1–6, DOI 10.1109/ICAT.2009.5348415

[4] A. Case, L. Marziale, C. Neckar, and G. G. Richard, "Treasure and tragedy in kmem_cache mining for live forensics investigation", Digital Investigation, vol. 7, 2010, pp. S41–S47, DOI 10.1016/j.diin.2010.05.006

[5] I. O. Ademu, C. O. Imafidon, and D. S. Preston, "A new approach of digital forensic model for digital forensic investigation", Int. J. Adv. Comput. Sci. Appl, vol. 2, no. 12, 2011, pp. 175–178, DOI 10.14569/ijacsa.2011.021226

[6] B. Carrier and E. H. Spafford, "An event-based digital forensic investigation framework", Digital forensic research workshop, Baltimore (MD), 11–13 August 2004, pp. 11–13

[7] Volatility Foundation, "Volatility framework: Volatile memory artifact extraction utility framework", http://www.volatilityfoundation.org/

[8] S. Kiltz, T. Hoppe, and J. Dittmann, "A new forensic model and its application to the collection, extraction and long term storage of screen content off a memory dump", Digital Signal Processing, 2009 16th International Conference on, Santorini, Greece, 5–7 July 2009, pp. 1–6, DOI 10.1109/ICDSP.2009.5201189

[9] B. Saltaformaggio, Z. Gu, X. Zhang, and D. Xu, "Dscrete: Automatic rendering of forensic information from memory images via application logic reuse.", USENIX Security, San Diego (CA), 20–22 August 2014, pp. 255–269

[10] B. Saltaformaggio, R. Bhatia, Z. Gu, X. Zhang, and D. Xu, "Guitar: Piecing together android app guis from memory images", Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver (CO), 12–16 October 2015, pp. 120–132, DOI 10.1145/2810103.2813650

[11] B. Saltaformaggio, R. Bhatia, X. Zhang, D. Xu, and G. G. Richard III, "Screen after previous screens: Spatial-temporal recreation of android app displays from memory images", Proc. USENIX Security Symposium (Security 16), Austin (TX), 10–12 August 2016

[12] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations", Digital Investigation, vol. 1, no. 1, 2004, pp. 50–60, DOI 10.1016/j.diin.2003.12.001

[13] A. R. Arasteh and M. Debbabi, "Forensic memory analysis: From stack and code to execution history", digital investigation, vol. 4, 2007, pp. 114–125, DOI 10.1016/j.diin.2007.06.010

[14] P. Movall, W. Nelson, and S. Wetzstein, "Linux physical memory analysis.", USENIX Annual Technical Conference, FREENIX Track, Anaheim (CA), 10–15 April 2005, pp. 23–32

[15] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory", Digital Investigation, vol. 3, no. 4, 2006, pp. 197–210, DOI 10.1016/j.diin.2006.10.001

[16] J. D. Kornblum, "Using every part of the buffalo in windows memory analysis", Digital Investigation, vol. 4, no. 1, 2007, pp. 24–29, DOI 10.1016/j.diin.2006.12.002

[17] A. Savoldi and P. Gubian, "Towards the virtual memory space reconstruction for windows live forensic purposes", Systematic Approaches to Digital Forensic Engineering, 2008. SADFE'08. Third International Workshop on, Oakland (CA), 20–22 May 2008, pp. 15–22, DOI 10.1109/SADFE.2008.21

[18] W. Cui, M. Peinado, Z. Xu, and E. Chan, "Tracking rootkit footprints with a practical memory analysis system.", USENIX Security Symposium, Boston (MA), 13–16 June 2012, pp. 601–615

[19] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang, "Mapping kernel objects to enable systematic integrity checking", Proceedings of the 16th ACM conference on Computer and communications security, Chicago (IL), 9–13 November 2009, pp. 555–565, DOI 10.1145/1653662.1653729

[20] S. Rahman and M. Khan, "Review of live forensic analysis techniques", International Journal of Hybrid Information Technology, vol. 8, no. 2, 2015, pp. 379–388

[21] Š. Balogh and M. Pondelik, "Capturing encryption keys for digital analysis", Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on, Prague (Czech Republic), 15–17 September 2011, pp. 759–763, DOI 10.1109/idaacs.2011.6072872

[22] Google, "Rekall", http://www.rekall-forensic.com/

[23] A. Bianchi, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Blacksheep: detecting compromised hosts in homogeneous crowds", Proceedings of the 2012 ACM conference on Computer and communications security, Raleigh (NC), 16–18 October 2012, pp. 341–352, DOI 10.1145/2382196.2382234

[24] J. T. Sylve, V. Marziale, and G. G. Richard, "Pool tag quick scanning for windows memory analysis", Digital Investigation, vol. 16, 2016, pp. S25–S32, DOI 10.1016/j.diin.2016.01.005

[25] G. G. Richard and A. Case, "In lieu of swap: Analyzing compressed ram in mac os x and linux", Digital Investigation, vol. 11, 2014, pp. S3–S12, DOI 10.1016/j.diin.2014.05.011

[26] A. Silberschatz, P. B. Galvin, G. Gagne, and A. Silberschatz, "Operating system concepts", vol. 4, Addison-wesley Reading, 1998

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

[28] I. Enrici, M. Ancilli, and A. Lioy, "A psychological approach to information technology security", Human System Interactions (HSI), 2010 3rd Conference on, Oakland (CA), May 2010, pp. 459–466, DOI 10.1109/SADFE.2011.7

[29] J. Stüttgen and M. Cohen, "Anti-forensic resilient memory acquisition", Digital investigation, vol. 10, 2013, pp. S105–S115, DOI 10.1016/j.diin.2013.06.012

[30] B. Schatz, "Bodysnatcher: Towards reliable volatile memory acquisition by software", digital investigation, vol. 4, 2007, pp. 126–134, DOI 10.1016/j.diin.2007.06.009

[31] J. Stuettgen, "On the viability of memory forensics in compromised environments.". PhD thesis, University of Erlangen-Nuremberg, 2015

[32] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory", Journal of the ACM (JACM), vol. 40, September 1993, pp. 873–890, DOI 10.1145/153724.153741

[33] S. Vömel and F. C. Freiling, "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition", Digital Investigation, vol. 9, no. 2, 2012, pp. 125–137, DOI 10.1016/j.diin.2012.04.005

[34] M. Gruhn and F. C. Freiling, "Evaluating atomicity, and integrity of correct memory acquisition methods", Digital Investigation, vol. 16, March 2016, pp. S1–S10, DOI 10.10161/j.diin.2016.01.003

[35] J. Stüttgen and M. Cohen, "Robust linux memory acquisition with minimal target impact", Digital Investigation, vol. 11, 2014, pp. S112–S119, DOI 10.1016/j.diin.2014.03.014

[36] LiME, "LiME - Linux Memory Extractor", https://github.com/504ensicsLabs/LiME

[37] A. Reina, A. Fattori, F. Pagani, L. Cavallaro, and D. Bruschi, "When hardware meets software: a bulletproof solution to forensic memory acquisition", Proceedings of the 28th annual computer security applications conference, Orlando (Florida), 03–07 December 2012, pp. 79–88, DOI 10.1145/2420950.2420962

[38] J. Bauer, M. Gruhn, and F. C. Freiling, "Lest we forget: Cold-boot attacks on scrambled ddr3 memory", Digital Investigation, vol. 16, March 2016, pp. S65–S74, DOI 10.1016/j.diin.2016.01.009

[39] M. Gruhn and T. Muller, "On the practicability of cold boot attacks", Availability, Reliability and Security (ARES), 2013 Eighth International Conference on, 2–6, September 2013, pp. 390–397, DOI 10.1109/ARES.2013.52

[40] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys", Communications of the ACM, vol. 52, May 2009, pp. 91–98, DOI 10.1016/j.diin.2006.06.005

[41] S. Lindenlauf, H. Höfken, and M. Schuba, "Cold boot attacks on ddr2 and ddr3 sdram", Availability, Reliability and Security (ARES), 2015 10th International Conference on, 24–27 August 2015, pp. 287–292

[42] F. N. Dezfouli, A. Dehghantanha, R. Mahmoud, N. F. B. M. Sani, and S. bin Shamsuddin, "Volatile memory acquisition using backup for forensic investigation", Cyber Security, Cyber Warfare and Digital Forensic (CyberSec), 2012 International Conference on, 26–28 June 2012, pp. 186–189, DOI 10.1109/CyberSec.2012.6246108

[43] S. Willassen, "Forensic analysis of mobile phone internal memory", IFIP International Conference on Digital Forensics, Florida, 13–16 February 2005, pp. 191–204

[44] A. Distefano and G. Me, "An overall assessment of mobile internal acquisition tool", digital investigation, vol. 5, 2008, pp. S121–S127, DOI 10.1016/j.diin.2008.05.010

[45] D. Irwin and R. Hunt, "Forensic information acquisition in mobile networks", Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on, 23–26 August 2009, pp. 163–168, DOI 10.1109/PACRIM.2009.5291378

[46] J. Sylve, A. Case, L. Marziale, and G. G. Richard, "Acquisition and analysis of volatile memory from android devices", Digital Investigation, vol. 8, no. 3, 2012, pp. 175–184, DOI 10.1016/j.diin.2011.10.003

[47] S. Vömel and J. Stüttgen, "An evaluation platform for forensic memory acquisition software", Digital Investigation, vol. 10, 2013, pp. S30–S40, DOI 10.1016/j.diin.2013.06.004

[48] Microsoft, "Microsoft Support - KB 920730", https://support.microsoft.com/en-us/kb/920730

[49] R. Harris, "Arriving at an anti-forensics consensus: Examining how to define and control the anti-forensics problem", digital investigation, vol. 3, 2006, pp. 44–49, DOI 10.1016/j.diin.2006.06.005

[50] K. Lee, H. Hwang, K. Kim, and B. Noh, "Robust bootstrapping memory analysis against anti-forensics", Digital Investigation, vol. 18, 2016, pp. S23–S32, DOI 10.1016/j.diin.2016.04.009

[51] J. Rutkowska, "Beyond the cpu: Defeating hardware based ram acquisition", Proceedings of BlackHat DC, vol. 2007, 2007

[52] N. Zhang, K. Sun, W. Lou, Y. T. Hou, and S. Jajodia, "Now you see me: Hide and seek in physical address space", Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, Singapore, 14–17 April 2015, pp. 321–331, DOI 10.1145/2714576.2714600

[53] Volatility Foundation, "Volatility Community plugins Official Repository", https://github.com/volatilityfoundation/community

[54] M. Foster and J. N. Wilson, "Process forensics: A pilot study on the use of checkpointing technology in computer forensics", International journal of Digital Evidence, vol. 3, no. 1, 2004

[55] P. Liang and L. K. NG, "N1ge6 checkpointing and berkeley lab checkpoint/restart", Sun Microsystems, Dec, 2004

[56] Jonathan Corbet, "Checkpoint/restart (mostly) in user space", https://lwn.net/Articles/452184/

[57] W. Li and A. Kanso, "Comparing containers versus virtual machines for achieving high availability", Cloud Engineering (IC2E), 2015 IEEE International Conference on, 9–13 March 2015, pp. 353–358, DOI 10.1109/IC2E.2015.79

[58] CRIU Project, "CRIU - Checkpoint Restore In User-space", https://www.criu.org

[59] CRIU Project, "CRIU - Comparison", https://www.criu.org/Comparison_to_other_CR_projects

[60] K. Mandeep, K. Navreet, and K. Suman, "A literature review on cyber forensic and its analysis tools", International Journal of Advanced Research in Computer and Communication Engineering, vol. 5, 2016, pp. 23–28, DOI 10.17148/IJARCCE.2016.5106

[61] B. Carrier, E. H. Spafford, *et al.*, "Getting physical with the digital investigation process", International Journal of digital evidence, vol. 2, no. 2, 2003, pp. 1–20

[62] B. J. Nikkel, "The role of digital forensics within a corporate organization", May 2006, IBSA Conference, Vienna, Vienna, Austria, May 2006

[63] J.-R. Sun, M.-L. Shih, and M.-S. Hwang, "A survey of digital evidences forensic and cybercrime investigation procedure.", IJ Network Security, vol. 17, no. 5, 2015, pp. 497–509

[64] B. Carrier, "Defining digital forensic examination and analysis tools using abstraction layers", International Journal of digital evidence, vol. 1, no. 4, 2003, pp. 1–12

[65] N. L. Beebe and J. G. Clark, "A hierarchical, objectives-based framework for the digital investigations process", Digital Investigation, vol. 2, no. 2, 2005, pp. 147–167, DOI 10.1016/j.diin.2005.04.002

[66] L. Wang, R. Zhang, and S. Zhang, "A model of computer live forensics based on physical memory analysis", Information Science and Engineering (ICISE), 2009 1st International Conference on, Nanjing (China), 26–28 December 2009, pp. 4647–4649, DOI 10.1109/icise.2009.69

[67] F. Olajide and S. Misra, "Forensic investigation and analysis of user input information in business application", Indian Journal of Science and Technology, vol. 9, no. 25, 2016, DOI 10.17485/ijst/2016/v9i25/95211

[68] Manu Garg, "Auxiliary vector", http://articles.manugarg.com/aboutelfauxiliaryvectors

[69] Michael Kerrisk, "Auxiliary vector explanation", https://lwn.net/Articles/519085/

[70] "VDSO", http://man7.org/linux/man-pages/man7/vdso.7.html

[71] CRIU Project, "CRIU - VNC", https://www.criu.org/VNC

[72] Enrico Canzonieri, "ElfDump", https://github.com/ecanzonieri/ElfDump