Cloud Forensic Framework For IaaS With Support for Volatile Memory

Matúš Baňas

Submitted as part of the requirements for the degree
of MSc in Cloud Computing
at the School of Computing,
National College of Ireland
Dublin, Ireland.

September 2015

Supervisor Dr. Horacio González-Vélez

Abstract

Cloud computing is attracting large base of users and organisations. However, lack of trust in public cloud providers, especially their legal responsibility towards the legislation in the countries of their origin sometimes complicates the move to the public cloud. Organisations, and communities such as research, education, government, and healthcare incline towards private or community cloud solutions. In those cases organisation, or the members of the community take role of the cloud provider. Similarly to the traditional IT infrastructure, cloud platforms also suffer from potential security related incidents. Regardless of the incident being internal, external, malicious, or accidental, it should be investigated, understood, and prevented from happening again in the future. However, underlying architecture of cloud introduced new challenges in digital forensics and made majority of the traditional forensic tools, and techniques irrelevant. Providers of private and community clouds are often kept in dark, with very limited ability to collect relevant evidence from their platforms. Cloud users on the other hand rely fully on the cooperation of the provider to provide the relevant data. This led to substantial research in the area of cloud forensics over last few years, trying to identify the efficient ways to successfully perform cloud forensics either as a end user, or a provider.

Focus of this thesis is to provide Self-service Forensic Framework for IaaS platforms, and determine the importance of volatile memory forensics in Cloud environment. Our framework allows cloud users, and cloud providers to extract disk, and memory images for forensic investigation in efficient, effective, and secure way.

Dedication

Kristínke

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Horacio González-Vélez for his support, and guidance in writing this thesis.

I would like to thank my friend Alex for pointing me in the right direction, and his feedback.

The biggest "thank you" to my wife Kristínka for her patience, and all her moral support.

... and thanks to everyone whom I have forgotten.

Contents

List of Figures

Chapter 1

Introduction

Volatile memory plays crucial role in digital forensics, and it can often lead to uncovering evidence hidden by anti-forensic techniques. Sophisticated attackers, or malicious applications can hide completely in the memory. Live OS environments often run entirely in memory, without touching the disk. In that case the only potential evidence could be collected from network, or firewall logs, and the volatile memory of that particular Virtual Machine instance (VM).

Crime happens every day, and while cloud computing offers numerous benefits, it still is just another platform that can be exploited by malicious users, or a cyber-criminals. In the recent years cloud computing attracted large base of users. Mainly for the convenience of outsourcing maintenance overhead to the public cloud provider, and the pay-as-you-go model, which also enables them to convert their capital expenses $CAPEX$ to operating expenses $OPEX$(Armbrust, Fox, Griffith, Joseph, Katz, Konwinski, Lee, Patterson, Rabkin, Stoica et al., 2010). On the contrary, many organisations who are not willing to give their data to the third-party providers, build their own private or community clouds. This way they can gain greater control over underlying hardware, software, and security procedures (Subramanian, 2011). Apart from technical challenges, security is still major concern associated with providing such infrastructure. Malicious software, malicious users, compromised accounts, accidental damage are only the tip of the iceberg, as even the most secure systems can be abused by the insiders who misuse their privileges (Denning, 1987).

Due to decentralised, distributed, and multi-tenant nature of the cloud traditional digital forensics procedures often cannot be applied (Birk and Wegener, 2011). Unknown location of the data, and shared resources between multiple clients also often complicate successful investigation. Many traditional techniques are becoming irrelevant, as the data is often spread across vast number of servers often located in multiple datacentres

across different countries around the world (Zawoad and Hasan, 2013). Regardless of the particular deployment model used, inability to perform successful digital forensic investigation often leaves cloud users in the dark, relying on the cloud provider to present relevant evidence. While in case of private, and community cloud, providers often lack appropriate tools, and features to perform adequate investigation (Dykstra and Sherman, 2012).

While cloud forensics is becoming a field of digital forensics, it still presents numerous challenges. Although many authors offered solutions for enhancing cloud forensics in their research, there is still a lack of implementation in cloud products.

This thesis aims to expand current solutions and provide efficient, and effective cloud forensic framework with the support for volatile memory forensics.

Chapter 2

Background

2.1 Digital Forensics

Definition of digital forensics vary from author to author. Vacca describes it as a principle of reconstructing activities, that identify 'what was done?' and 'how was it done?' (Vacca, 2002). Zargari and Benford refer to it as a process of analysing digital data, while preserving its integrity and validity, by gathering, preservation, validation, analysis, interpretation and documentation of the digital evidence. Authors refer to the digital evidence as any information transmitted or stored digitally, and this information holds probative value (Zargari and Benford, 2012). However, all the definitions generally conclude digital forensics as a science of recovering and investigating digital evidence. Increasing number of criminal activities involving computers over last decade is the main driver for forensic investigations (Fei, 2007). Investigation is typically performed either on digital resource such as computer, or server that was used to commit the crime, or was a target of crime (Prosise, Mandia and Pepe, 2003). Nevertheless, digital forensic is also used to investigate internal organisational policy violations, troubleshooting various operational issues, recovering from accidental system damage, reconstructing security and technical incidents, or for verifying regulatory compliance of the organisation. Therefore, every organisation needs to have capabilities to perform some sort of digital forensics (Kent, Chevalier, Grance and Dang, 2006).

2.1.1 Forensic Investigation Process

The National Institute of Standards and Technology (NIST) special publication 800-86, identifies four phases of forensic investigation process as collection, examination,

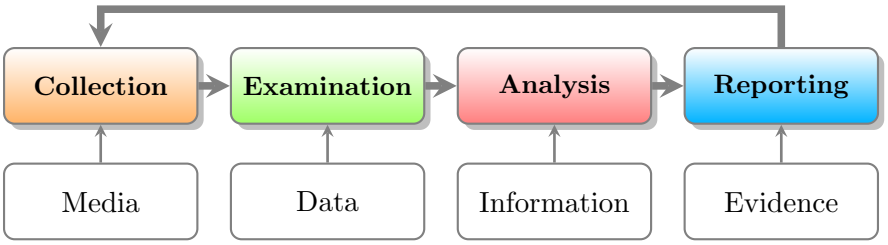

Figure 2.1: Forensic Process

analysis and reporting (Kent et al., 2006). Figure 2.1 shows the flow of the process and the objects associated with each phase. Integrity of the evidence must be preserved during the entire process of investigation. Purpose of the *Collection* phase is to identify the relevant data sources, secure the media, and record data for examination. During *Examination* phase relevant information is extracted from collected media, while preserving its integrity. Digital evidence can be very fragile and volatile, therefore it needs to be handled with care. This is usually done by creating bitwise copy of the original source or a media (Birk and Wegener, 2011). Examination is typically never performed on the original media to avoid potential damage during the process. *Analysis* of the examination results is done using legally justifiable techniques of extracting relevant information or the evidence. Purpose of the *Reporting* phase is to produce a report containing documentation of evidence and the results of the analysis. This report may also contain the detailed description of the tools and procedures used for extracting those results. If the evidence is not sufficient, or gaps in the report are identified, or it is recommended by the report then the forensic investigation process starts again from the collection, as shown in figure 2.1 (Kent et al., 2006)

2.1.2 Data Sources

From the technical point of view data can exist in three states: at rest; in motion; and in execution. Data *at rest* has allocated space on the storage media, whether as a file in a specific format, or as a data in the database. If the data is transferred over the network from one device to other, then it is referred to as a data *in motion*. Data *in execution* is loaded into volatile memory and executed as a process (Birk and Wegener, 2011). Before the data is collected, the sources of data must be identified. There are many potential sources that can provide valuable information during forensic investigation. Current wider use of digital technology for personal and professional purposes have expanded the number of data sources. Desktop and laptop computers, servers, and

network storage as a traditional data source, are now accompanied by other 'smart' devices capable of storing and exchanging data. Smartphones, tablets, digital media players, digital audio and video recorders are only some of them. All of these systems typically contain some sort of internal storage and support external storage media such as different types of memory cards, CD/DVD media, or USB thumb drives. Some also contain volatile memory where data is stored temporarily, and typically lost when system is rebooted. Therefore, it could be only extracted while system is running. System and application logs from the systems, but also from various network devices and monitoring systems are also valuable source of information (Kent et al., 2006). In 2007, Fei referred to two types of data that can reside on the digital media as the active data and the residual data. The *Active Data* that is available and visible to the operating system and to its users. This includes any documents that users have saved, application data, operating system files, temporary files, cache files of web browser, and any other files ready to be accessed. The *Residual Data* typically exist on the media after the files were deleted. Deletion process typically removes only information that points to the data on the physical media, but the data still exist until they are overwritten. Operating system, applications, or users, are not able to access this data directly. However, tools and techniques to recover residual data exist and are often used in forensic investigation (Fei, 2007). Cloud computing complicates the matter of identifying the media where the data is located, and in many cases obtaining the data or recovering residual data is almost impossible as the data is by design distributed randomly across multiple nodes (Birk and Wegener, 2011).

2.1.3 Memory Forensics

Memory became popular data source for forensic investigation. In 2009 Halderman et. al. published their research, covering their discovery that computer memory holds the information for couple of minutes after the computer is shut down, before it began to degrade. It enabled researchers to obtain the memory image by rebooting running target machine, and booting their custom software from either USB device, or from network using PXE boot. They were able to gain access to the encrypted disk drives by extracting the encryption keys from memory images (Halderman, Schoen, Heninger, Clarkson, Paul, Calandrino, Feldman, Appelbaum and Felten, 2009). Memory holds other valuable forensics data that can be extracted by variety of available tools. While memory structure varies between OS versions, or even kernel versions, it will still contain similar sources of information. Typically, memory contains list of running processes, open network connections, arp table containing list of cached IP addresses and associated MAC addresses of recently connected devices, history of the executed commands, the

content of files that was recently loaded by the OS, fragments of recently deleted files, and much more (The Volatility Foundation, n.d.). Malicious users often deploy various anti-forensic tools, and techniques in order to hide their activities. Modern malware, viruses, and rootkits are often hidden in memory without writing anything to the disk. Jahankhani et. al. covered different anti-forensic tools and techniques such as use of live CD/DVD environments, or use of a Ramdisk. The whole Live OS resides in the volatile memory, and the ramdisk is a file system fully located in volatile memory. Both techniques make disk forensics irrelevant, as no data is written into physical disks, and will disappear when computer is shut down or restarted (Jahankhani and Beqiri, 2010).

2.2 Cloud Forensics

Cloud computing introduced new challenges in digital investigation. In 2010 Garfinkel called the years from 1999 to 2007 the "Golden Age" for the digital forensics. During those years platforms were mostly uniformed and examinations were done on single devices. Forensic examinations were focusing on a small number file formats such as Microsoft Office documents, JPEG graphics and AVI video formats. Storage devices were relatively small size and typically of a standard interface type (IDE/ATA), that enabled fast bitewise cloning for examination. Nowadays, technology expanded into scales that complicates the process of forensic investigation (Garfinkel, 2010). Many authors agree that cloud environment contributed to the complexity of the digital forensic investigation by the distributed nature of the underlying technology, and sharing resources between multiple users and organisations (Birk and Wegener, 2011)(Zargari and Benford, 2012)(Ruan, Carthy, Kechadi and Crosbie, 2011)(Dykstra and Sherman, 2012)(Biggs and Vidalis, 2009)(Chen, 2014)(Thethi and Keane, 2014)(Patrascu and Patriciu, 2013)(Guo, Jin and Shang, 2012). Cloud forensics is forming as a science field, and while it is still at its early stage, it is increasingly gaining relevancy and attracting more research (Thorpe, Grandison and Ray, 2012).

2.2.1 Cloud Computing

Numerous different definitions of cloud computing were created over the years (Geelan, 2009). The wider technical community have accepted definition provided by NIST Special Publication 800-145. According to NIST, cloud computing is a model for enabling convenient, ubiquitous, on-demand network access to a shared pool of configurable resources (such as applications, servers, storage, networks, and other services), that can be rapidly provisioned and used with minimal management effort

or service provider interaction. Authors refer to four deployment models as private cloud, public cloud, hybrid cloud, and community cloud. Each model may have same characteristics and offer same services, but the organisational ownership is different. *Public cloud* is deployed for use by general public, and available to anyone who is willing to pay for it. It is owned by cloud provider and typically exists on the premises of the provider. *Private cloud* is provisioned for exclusive use by single institution and can be either owned and operated by the organisation, or the cloud provider. *Community cloud* is similar to private cloud, but is provisioned for exclusive use of community with shared interests such as mission, or security concerns. It can be owned and operated by its member organisations, or a third party, and it can exist on or off the premise of the provider. Where the *Hybrid cloud* is a combination of at least two distinct clouds that are interconnected and synchronised, but remain unique entities (Mell and Grance, 2009).

Traditionally cloud computing is delivered in three service models, *Software as a Service* (Saas), *Platform as a Service* (PaaS), and *Infrastructure as a Service* (IaaS). Although, some providers choose to expose their services through more than one model. Each service model offers different levels of control to the users and to the cloud provider. *IaaS model* exposes interface for provisioning virtual servers, networks, storage and other equipment. Users need to build their infrastructure before they can deploy their applications. This model offers great control and dynamic on-demand scalability of deployed infrastructure. However, users also inherit large maintenance overhead, as the servers need to be patched, configured, monitored and secured. Those maintenance tasks are gradually shifted to the cloud provider in the other two models. *PaaS model* exposes high-level environment for building, testing and deploying custom applications. Generally there are restrictions on what type of software can be deployed and how the application is written in order to support application scalability. Some of the operational overhead such as maintenance, management and patching of the operating system and the middleware is now shifted to the cloud provider. PaaS users can focus on development and maintenance of the applications. *SaaS model* removes the maintenance overhead completely as it delivers special-purpose applications owned and managed by the provider. This applications are accessed directly through the Internet and SaaS users have no control over the application itself (Foster, Zhao, Raicu and Lu, 2008).

2.2.2 Private and Community Clouds

For the scope of this research we are focusing on IaaS in private and community cloud scenarios. While challenges of public IaaS deployments would be very similar, there is very little information about internal practices and implementations of public cloud providers, mostly guarded by non-disclosure agreements. Many institutions, especially related to research and education, are choosing open source cloud platforms such as OpenStack, Eucalyptus, or CloudStack (The OpenStack Foundation, n.d.)(Eucalyptus Systems, Inc., n.d.)(The Apache Software Foundation, n.d.*a*). Transparency and ability to customise the system based on their needs is the main driving factor (Nurmi, Wolski, Grzegorczyk, Obertelli, Soman, Youseff and Zagorodnov, 2009). Regardless of the benefits of open source solutions, there are many gaps in those products that require technical knowledge to overcome. Apart from the technical challenges of building and managing private or community cloud infrastructure, providers are facing various security issues. Even they suffer from security threats and attacks such as Denial of Service (DoS), Distributed Denial of Service (DDoS), IP spoofing, Man in the middle (MITM), DNS spoofing and other threats. Nicanfar et. al., Rosche et. al., Lo et. al. are offering solutions to mitigate those threats by implementing intrusion detection (IDS), and intrusion prevention (IPS) systems (Nicanfar, Liu, Talebifard, Cai and Leung, 2013)(Roschke, Cheng and Meinel, 2009)(Lo, Huang and Ku, 2010). While, IDS and IPS may improve the overall security of the infrastructure, we should not fall under impression that they will solve all our problems. Incidents will happen if human factor is involved. New attack patterns are invented, and security holes detected daily. While researchers are trying to solve the issues of forensic investigations in cloud environments, they are often focusing on the aspect of forensic investigation from the view of organisations as a cloud users (Dykstra and Sherman, 2012)(Biggs and Vidalis, 2009)(Zargari and Benford, 2012)(Ruan et al., 2011)(Chen, 2014). On the contrary, cloud providers have better visibility of the underlying infrastructure and access to the greater areas of the platform. Some challenges discussed by the authors apply also to the cloud providers, but in many cases they are easier to solve. However, tools and frameworks are often unavailable (Saibharath and Geethakumari, 2014).

2.2.3 Challenges of Cloud Forensics

Recent research of many authors suggests that main challenge of forensic investigation in cloud is the process of obtaining data, and preserving the evidence (Chen, 2014)(Saibharath and Geethakumari, 2014)(Shaw, Bordbar, Saxon, Harrison and Dalton, 2014)(Thethi and Keane, 2014)(Guo et al., 2012)(Dykstra and Sherman,

2012)(Zargari and Benford, 2012)(Birk and Wegener, 2011)(Delport, Köhn and Olivier, 2011)(Ruan et al., 2011)(Biggs and Vidalis, 2009). Investigation in cloud is particularly difficult. Logs and data of multiple users and organisations may be co-located on same storage devices, and are spread across multiple hosts and locations that are constantly changing. While cloud provider has access to most components of the platform, and can possibly gather more forensic data, it is still complex and challenging task (Guo et al., 2012).

This thesis focuses on IaaS model, as there are different forensic challenges in PaaS, and SaaS models. IaaS contains many sources of valuable forensic data. This data can be captured at any of its three states, such as at rest, in motion, and in execution. However, due to the virtualisation technology used in IaaS this data is located in various layers of the platform. Data resides on virtual machines (VMs), virtual networks, and virtual storage devices owned and controlled by the cloud user, or physical devices owned, and controlled by cloud provider. In most cases VMs are sources of the incidents, and therefore often the main source of evidence. VMs contain stored data, running processes, system and application logs, and much more. Cloud provider typically have no access to the VM. There is a potential to gain access to the runtime state of the VM via the hypervisor, and snapshot technology, that enables freezing the VM in specific state (Birk and Wegener, 2011). Delport, Oliver, and Köhn discuss the need to isolate a crime scene in a cloud, rather than isolating single VM. Authors also proposed preservation of confidentiality, integrity, and availability (CIA) of other VMs, by relocating and isolating the suspicious VM to different hardware node.(Delport et al., 2011). Snapshot technology is also useful to prevent loss of evidence, when VM is deleted. On-demand feature of IaaS allows users to create virtual infrastructure, but also delete them without a trace. Via application programming interface (API) of most IaaS platforms this process can be easily automated to destroy the evidence after the malicious activity was completed. In convergence with Intrusion Detection Systems, snapshots can be created automatically when suspicious activity is detected (Grobauer and Schreck, 2010). Although, snapshot technology is available in many hypervisors, it is not available in most of the cloud platforms. Recent research has been conducted in this area, that mainly suggests development of the tools and frameworks for particular cloud products (Saibharath and Geethakumari, 2014)(Poisel, Malzer and Tjoa, 2013)(Delport et al., 2011).

Snapshot technology is not limited only to the VMs and hypervisors. Different filesystems have this capability built-in for many years. Birk and Wegener suggested

use of snapshots to solve the issue of unknown physical location of the data in the cloud (Birk and Wegener, 2011). Creating regular snapshots would enable forensic investigators to inspect the snapshot of the filesystem from particular time, without the need to recover the deleted data from the physical hardware.

In 2012 Dykstra and Sherman analysed the options of acquiring digital evidence from IaaS using popular forensic tools. Authors have concluded that while the tested tools were technically capable of the acquisition of the data, it was insufficient to produce trustworthy evidence. Their research led to the recommendation of implementing forensic framework into cloud management plane of cloud platform to produce fast and trustworthy evidence (Dykstra and Sherman, 2012).

Log files are often great source of information for such cases of forensic analysis. However, analysing logs of cloud platform proves to be a complex task. Logging is implemented on different layers of the platform and produced on various sources (physical nodes, network devices, storage nodes and devices, VMs, applications, etc.). Despite the relevance of log files for technical troubleshooting and monitoring of the platform and its components, filtering relevant information to a single user or institution is challenging. Some authors are proposing to simplify the forensic investigation in the cloud by creating logging framework designed for this purpose (Patrascu and Patriciu, 2014)(Thorpe, Grandison and Blake, 2014)(Sang, 2013). In 2013 Dykstra and Sherman continued their prior work, and designed forensic framework on OpenStack called FROST. This framework was a first attempt to integrate forensics into a cloud platform, by introducing additional logging facilities of the activities within OpenStack compute nodes. It enables the user to access the logs of their own cloud environment via OpenStack dashboard. FROST stores hashed logs in tree structure, enabling user to access OpenStack's firewall logs for only this particular user's infrastructure. In addition this framework enables user to export and download VM disk images for forensic investigation. Images are encrypted and signed by cloud providers and users public keys, so the consistency and validity of the disk image can be verified (Dykstra and Sherman, 2013). Year later Saibharath and Geethakumari designed their own forensic framework for OpenStack, while claiming that FROST would not give the whole picture of cloud activities as it resides on compute nodes OpenStack. Saibharath and Geethakumari proposed to use controller nodes instead to provide more accurate data (Saibharath and Geethakumari, 2014). While both frameworks focus on logging capabilities, and the disk image investigation, they are not taking volatile memory forensics into count.

Virtualization as a core component of cloud computing makes extracting memory from virtual machines easier comparing to the physical hosts, especially in IaaS. No special hardware, or software is required, except for appropriate permissions to the hypervisor. Most of the modern hypervisors such as KVM, VMWare, XEN, and Hyper-V support memory, and disk extraction. However, currently in IaaS those tools are not available to the cloud users. What make it impossible to obtain the disk and memory images without cloud provider's interaction.

2.3 Research Question

"Is Forensics of Volatile Memory Relevant in Cloud Forensics?"

This thesis will attempt to provide effective, efficient, and secure self serviced forensic framework, enabling cloud consumers to perform forensic investigation on the components of their own cloud infrastructure. As self-service being one of the characteristic of cloud computing, we believe that cloud users should be able to obtain disk, and memory images from their own infrastructure without cloud provider interaction.

For the scope of this thesis we focus on Infrastructure as a Service (IaaS) using OpenStack. We will attempt to extract digital evidence from various components of our cloud, and analyse importance of memory forensics in obtaining evidence in different real world scenarios. Main goal of our framework is providing access to the memory, and the disk images for forensic purposes, while protecting integrity, validity, and trustworthiness of the obtained evidence. Outcome of our research will provide self-service cloud forensics framework, that can be potentially extended to platforms other than OpenStack. Meanwhile, we will highlight the importance, and the value of the memory investigation in cloud forensics.
In this thesis, we evaluate the efficiency of our framework, and effectiveness of identifying, and recovering digital evidence of real-world scenarios, from the memory, and the disk images extracted by our framework.

The major IaaS providers refer to the Virtual Machines as instances, and we will refer to them as such throughout this thesis. Memory often holds the keys, and passwords to the encrypted disks, or recently decrypted files, list of open network connections, list of processes, fragments of open files, and other valuable data. Storing files on the

local disk can be avoided even with the use of the ramdisk, or the network storage. Some Linux distributions deploy ramdisk by default for storing temporary files. Files and applications stored in ramdisk will not be uncovered by off-line investigation of local disk images. Uncovering attached network storage in the cloud is also more difficult than in traditional network, where investigators would be aware of the storage device or a server. In the cloud, the storage may be located anywhere, even in cloud infrastructure offered by a different provider. Memory forensics can uncover the IP addresses of the storage devices, and lead to the further investigation of that network device.

Storage requirements are constantly growing, and the size or the volatile memory is typically much smaller than the disk. Investigation on few gigabytes of memory will take substantially less time than investigation of hundreds, or thousands of gigabytes of disk space. Meanwhile, detecting mounted network storage, uncovering active network connections is also faster, and easier by the memory forensics than searching through network logs, or capturing and analysing network packets. One of the issues our framework is addressing is ability of the user to obtain disk, and memory image of the instance without cloud providers interaction. We are also simplifying the task for cloud provider who needs to investigate instances of their customers.

Chapter 3

Design

Our design was inspired by work of Dykstra et. al. (Dykstra and Sherman, 2013), and Saibharath et. al. (Saibharath and Geethakumari, 2014). Both authors have created forensic frameworks, allowing users to obtain the firewall logs of their virtual infrastructures, and ability to extract instance disk images for forensic investigation. Our framework is expanding the functionality by adding memory extraction abilities, and proposing different approach of verifying integrity, and validity of obtained images. Similarly to the disk images, extracted memory must be securely delivered to the user, with the ability to verify the integrity of the file to prevent tempering with the evidence. Authors are proposing the use of hash values of the extracted images to validate that the image has not been tampered with. These values are then stored in the database of Cloud provider. User, and relevant authorities can then validate the hash of the image against the database. Meanwhile, both frameworks are encrypting and signing the extracted image, to ensure security of the content.

3.1 Proposed Design

Our approach of validation of the images eliminates the need for storing hash values of the images by Cloud Provider. We are creating two additional files, metadata file, and cloud providers signature. Metadata file will contain the hash value, and other information about the image, such as date, and time of extraction, user name, type of the image, instance name. Other information can be added in the future, which are out of scope of this thesis. Instead of encrypting and signing the images, we only encrypt the image, and sign the metadata file by Cloud Provider's GPG key. This approach will ensure that extracted image can be verified any time, and does not rely on availability

of the database used in previous approaches. We will also eliminate the need for signing the image files, which will take substantially longer than signing a simple text files.

3.1.1 Self Service

We believe that user should own their Cloud infrastructure. Therefore, they should also be able to extract their own disk, and memory images. Regardless, if it is forensic investigation, or any other reasons, such as backup, migration to their own virtualization platform, or migration to different Cloud Service. Our self-service framework enables the user to do that.

3.1.2 Memory Extraction

In comparison to the traditional digital forensics of physical machines, hypervisor as one of the core components of IaaS provides easier way to extract memory of instance without the need for specialised hardware, or an access to the running OS. We are proposing adding the ability to extract memory images of the instances. Our framework will execute memory extraction directly from the hypervisor by executing its native functions.

3.1.3 Storage Types

Technique of extracting the disk images differs between the types of the storage used. Different Cloud platforms support different hypervisors, and storage types. OpenStack's default hypervisor is KVM, and its native disk image format is QCOW2. However, it also supports Hyper-V, XEN, and VMWare hypervisors and their native image types, such as raw images, VMDK, VDH. In the meantime, instance disk can be theoretically provided by any storage system, such as LVM volumes, different SAN targets, or native block storage. Each storage type has its benefits and flaws. For example raw image offers slightly better performance, but does not support features such as snapshots. When designing forensic framework, we need to make sure to be able to extract each type of the image supported by our cloud platform.
While for extracting images from block storage devices, we rely on bite-wise copy of the original media, in case of native images such as QCOW2, VMDK, and VDH , we can improve extraction time by using the native tools provided for those images. Our framework will detect the image type and use relevant tools for extraction.

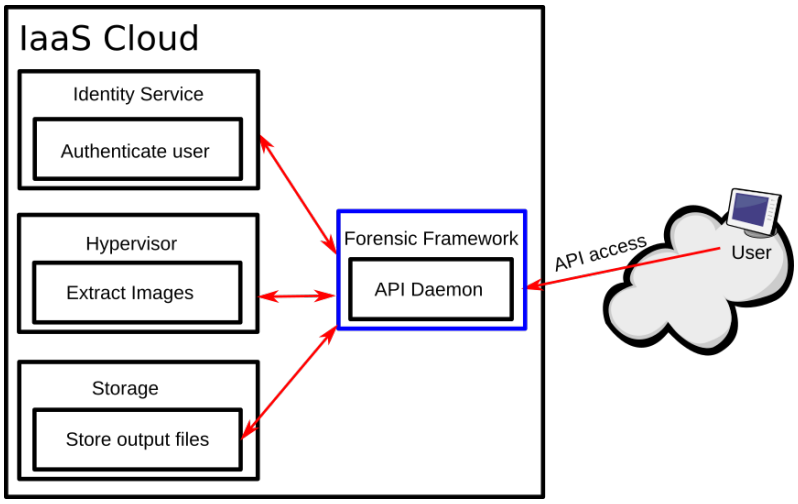

Figure 3.1: Forensic Framework Design

3.1.4 Image Extraction

The high level design in figure 3.1 shows the user interaction with our framework. First, the user requests extraction via web dashboard, or API call. The request is then authenticated against the identity service. If access is granted, the API server will execute memory, or disk image extraction from the hypervisor. Finally, the output files are made available to the user on the network storage.

3.1.5 Validation

After the image is extracted, the hash value of the image is written into metadata file. Our framework signs the metadata file with the Cloud Providers private GPG key. Validation of the image is done by comparing the hash value from the metadata file against the calculated hash value of the decrypted image. In order to ensure the validity of the metadata file, simple gpg signature verification can be done against the cloud provider's public GPG key, which can be accessed via API of our forensic framework.

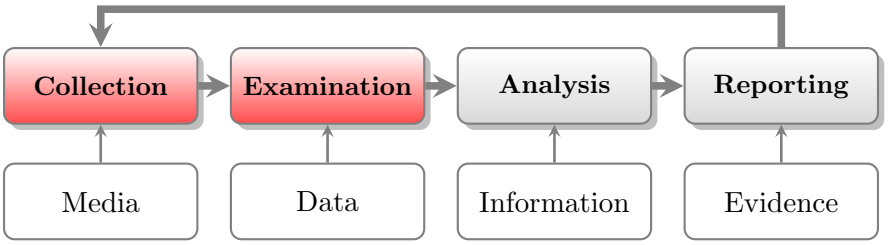

Figure 3.2: Forensic Process

3.2 Analysis Methods

Although, our framework operates in the collection phase of the forensic process, our evaluation will target both collection phase, and examination phase, as shown in figure 3.2. Successful collection of the media, in our case disk, and memory images, is crucial to the whole forensic process. Without the collection, remaining phases would not be possible.

Analysis of our solution is based on two part evaluation. In the first part of the evaluation we will compare extraction times of memory, and disk images of different instance sizes. Our evaluation should highlight efficiency of collection of the memory images, in comparison to the disk images in Cloud environment. Second part of the evaluation focuses on effectiveness of the examination of the memory images, in contrast to the disk images in different real world scenarios. We will attempt to recover evidence from different types of instances, while data will be located on local disk, network storage, and ramdisk. Details of each test and tools used are described in the evaluation chapter.

Chapter 4

Implementation

For implementation of our forensic framework, we have chosen OpenStack cloud platform (The OpenStack Foundation, n.d.) for its open source nature, large community of developers, active development, and constantly growing user base.

OpenStack has a pluggable architecture, and can be build with a combination of different components, that are providing additional features. Three core components shown in figure 4.1, are required to provide the core IaaS functionality. Users interact with OpenStack either using a dashboard called Horizon, or via application programmable interface (API). While dashboard presents convenient access via web interface, API offers more flexibility, and allows easier integration with external applications. Therefore, our main focus will be in delivering our framework via API, and potentially expanding it into dashboard. The main components of the OpenStack in production deployment are the compute engine Nova, the block storage Cinder, the object storage Swift, the network engine Neutron, and authentication engine Keystone.

4.1 OpenStack Overview

Nova is mainly responsible for running and managing instances, but it can also offer basic networking functionality if no other network engine is used. It supports various hypervisors such as Kernel-based Virtual Machine (KVM), XEN, VMWare, Hyper-V, and Linux Containers (LXC). We have chosen KVM, as it is OpenStack's default hypervisor.

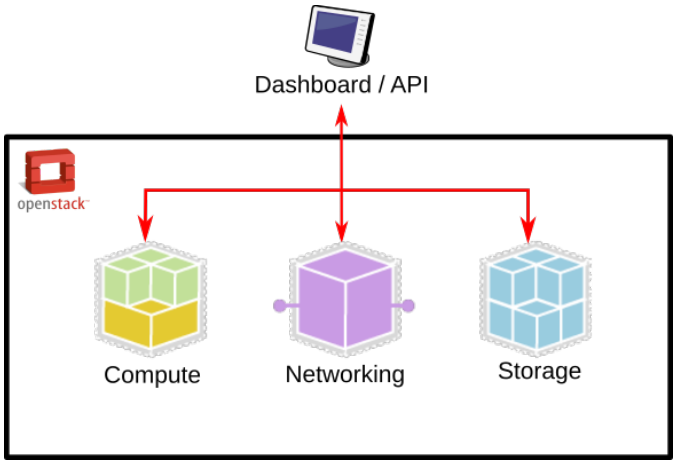

Figure 4.1: Core components of OpenStack

Cinder, and *Swift* are two storage engines, each responsible for storing different types of data. Cinder is a block storage, that provides raw disk storage, typically for storing the disk images of the instances. It uses iSCSI, Fibre Channel, NFS and other protocols for connection to back-end storage systems. On the contrary Swift is an object storage that stores data in a form of objects. In OpenStack it is mainly used for storing an instance deployment images (instance templates). Additionally, Swift provides storage service similar to Amazon S3 (Amazon S3, n.d.), where users store data as objects via HTTP protocol using PUT and GET commands. From the forensic perspective each of the storage systems can store potential forensic data. Swift and Cinder can also be used by malicious users to store the data, which can be classified as evidence during the forensic investigation. Without the memory forensics it may be difficult to identify data stored by instance in Swift object storage instead of local disk image.

Neutron provides network connectivity between virtual and physical interface devices that are managed by other OpenStack services, especially by Nova compute engine. It provides scalable, on-demand network abstraction. It supports plugins such as quality of service (QoS), Layer-2 in Layer-3 tunnelling, VPN as a service, Firewall as a service, IDS as a service, and many more. Neutron can also be a valuable source of forensic data. However, analysis of its logs can be complex task, and the events relevant to the forensic investigation may often not to be recorded.

Keystone is the identity service that manages access to all components of OpenStack. It provides authentication and authorization of users and services, and supports external authentication services, and mechanisms such as security assertion markup language

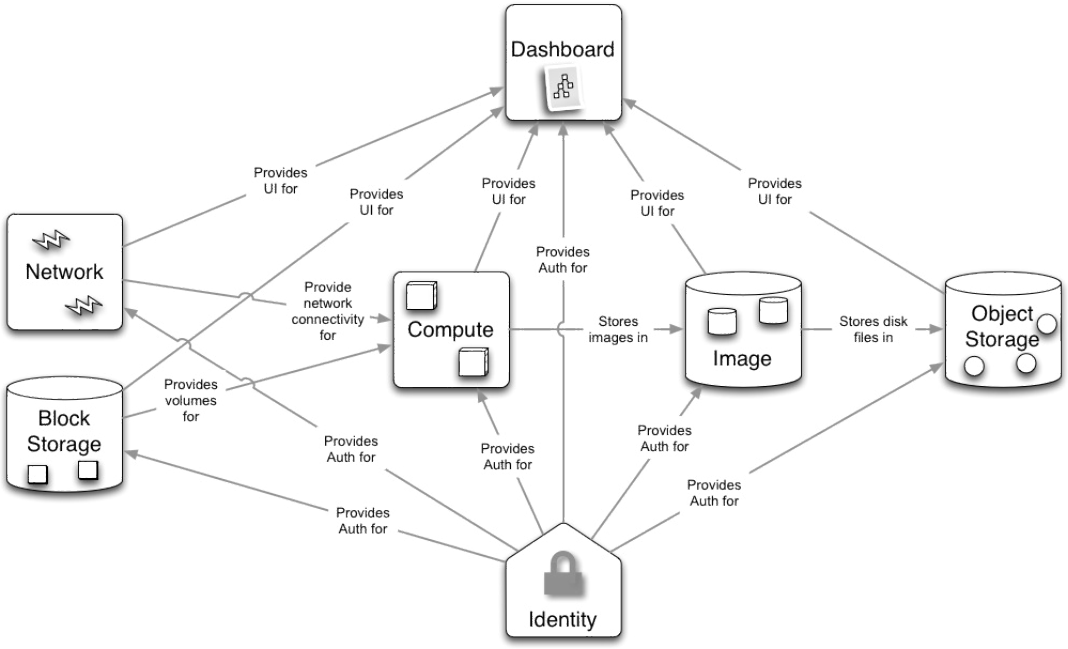

Figure 4.2: OpenStack diagram (The OpenStack Foundation, n.d.)

(SAML) (OASIS consortium, n.d.).
Interaction between OpenStack components in figure 4.2, shows the complexity of the platform (The OpenStack Foundation, n.d.). Investigation in such a complex environment can be difficult, especially without a forensic framework in place.

4.2 Forensic Framework Details

Our forensic framework consists of two parts, API server, and API client. The API server is a standalone application designed for OpenStack written in Python (Python Software Foundation, n.d.), and BASH (Free Software Foundation, Inc., n.d.). It interacts with OpenStack's native API for authentication, authorisation, and identification of the instances. It uses libvirt API of the KVM hypervisor for memory extraction, and directly access the filesystem, and the storage devices for the disk image extraction. The API client was written fully in Python, and uses HTTP, and JSON to communicate with the API server. Figure 4.3 shows diagram of our implementation and interaction between API client and server.

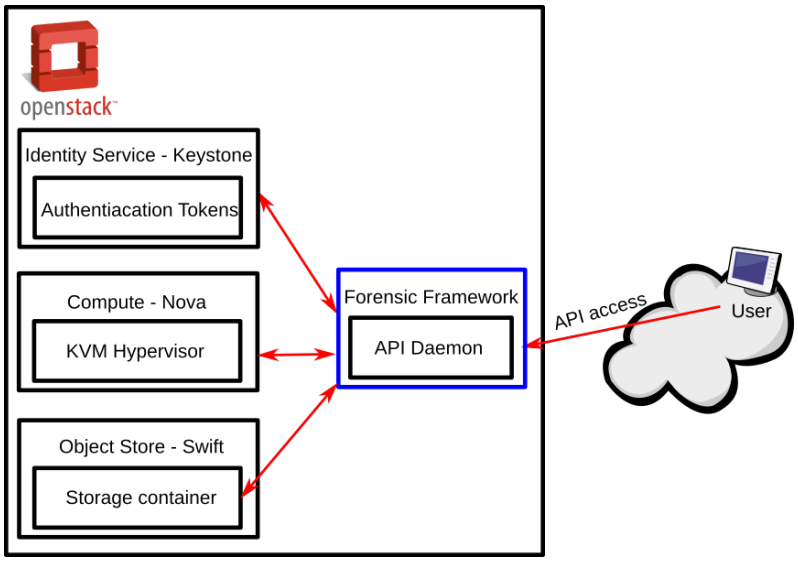

Figure 4.3: Forensic Framework implementation

When user sends the extraction command using API client, specifying the OpenStack credentials, the instance ID, tenant, and the name of the Swift container where the output files will be stored after extraction. Once the extraction is successful, user is presented with three files located in the desired location on the Swift object storage. File with the *"gpg"* extension is the extracted image file encrypted by user's OpenStack password. The *"metadata"* extension represents a metadata file, containing information about the image in simple text format. Apart from other useful information, this file contain the calculated hash value of the decrypted image, for verifying the image integrity and validity. The last file with a *"sig"* extension is the signature of the metadata file. It provides ability to verify the signature against the public GPG key of the Cloud Provider. Example of output files in figure 4.4 shows the example content of the Swift container, the encrypted image, the metadata file, and the signature file.

Figure 4.5 shows the process of the image extraction. Using the api client application, user passes the OpenStack credentials, instance ID, name of the storage container, name of the OpenStack tenant, and the type of the media to be extracted (either disk, or memory). API client will then send the authentication request using provided username and password to the authentication service (Keystone), and retrieve the authentication token. All the parameters, including the authentication token are then passed to the API server. Server then uses the username, and the authentication

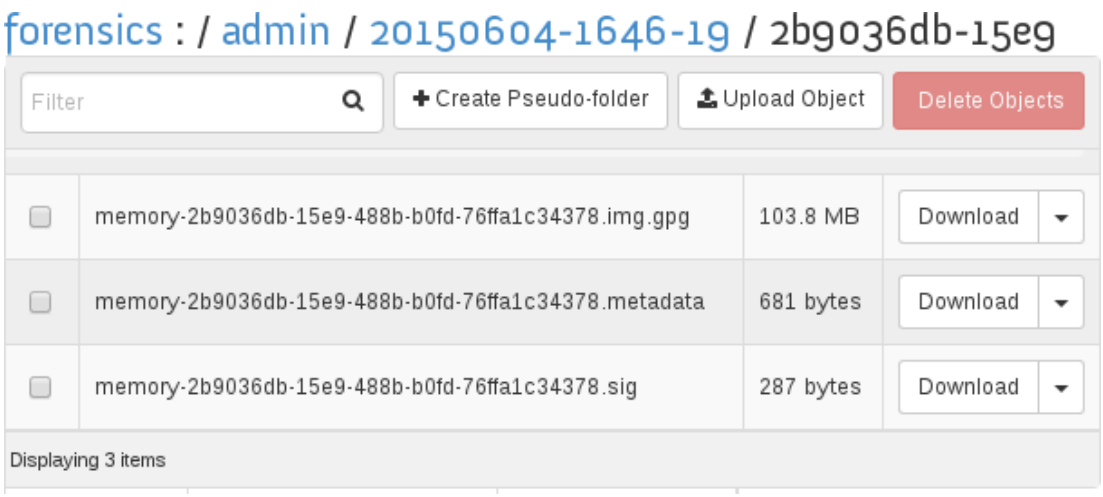

Figure 4.4: Swift Container containing output files

token to verify permissions to access the instance. Once the access is granted, API server creates the metadata file, containing username, extraction start time, and instance name. API server then identifies the name of the virtual machine of the KVM hypervisor corresponding to the OpenStack's instance ID, and starts the extraction process using the native tools for either disk, or the memory. Hash value of the extracted image is then generated. Our proof of concept implementation defaults to md5 hash, but code supports different hash types such as sha1, sha256, and can be extended to other algorithms. Hash value is next written into metadata file, and the disk image is encrypted using symmetric encryption against the user's OpenStack password. Final metadata file is signed by Cloud Provider's GPG key, and all files are uploaded to the specified container in Swift object storage. Once the transfer is finished, all files are deleted from the temporary location.

4.2.1 Memory Extraction Process

Memory extraction process slightly differs between hypervisors. We have chosen KVM because it is the default hypervisor in OpenStack. While OpenStack supports KVM on x86, ppc64, and ARM architectures, our solution is tested on x86 only, running 32bit and 64bit instances. However, libvirt API used by KVM supports all its architectures, therefore our framework should support them too (Libvirt virtualization API, n.d.). KVM uses modified version of an open source machine emulator and virtualizer QEMU to provide virtual hardware including volatile memory to the instance (QEMU emulator

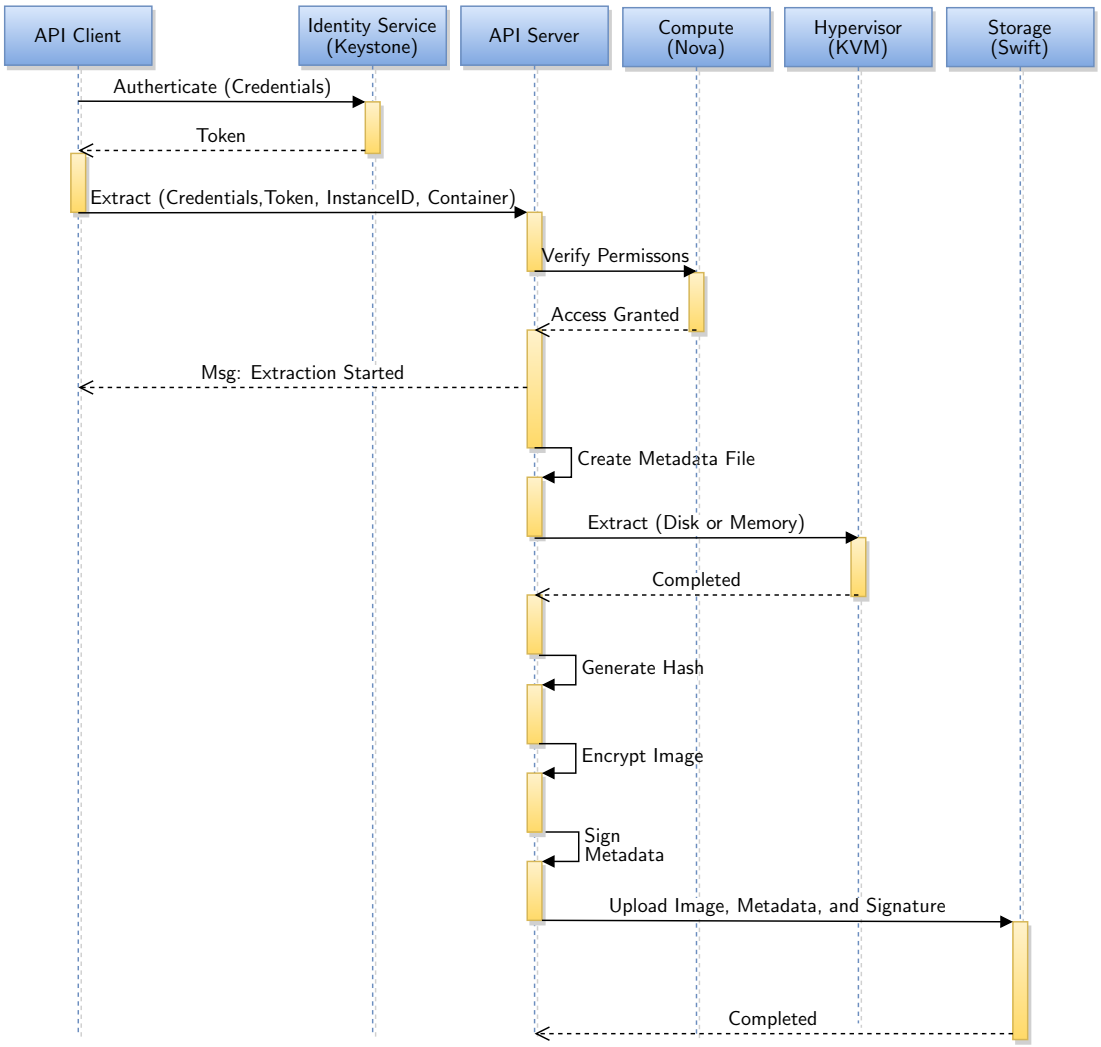

Figure 4.5: Forensic Framework Extraction Process

```
$cat disk-3510c4d3-fd55-4961-9984-40d8814f68ee.metadata
Extraction Date: Mon Jul 20 09:25:33 IST 2015
Extracted By: admin
Image type: Disk
Cloud Provider: Openstack NCI Demo
File Name: disk-3510c4d3-fd55-4961-9984-40d8814f68ee.img
File Hash (md5): 1806810d16486ae56017b7bbdc27de7e
```

Figure 4.6: Metadata Example

and virtualizer, n.d.). Our framework extracts volatile memory of the instance from KVM hypervisor using libvirt API (Libvirt virtualization API, n.d.).

4.2.2 Disk Extraction Process

Our API server supports QCOW2 image disk types (KVM Community, n.d.), which is default image type of KVM Hypervisor, used as default hypervisor in OpenStack. However, we have implemented hooks in our code, that supports adding different image types in the future. Our extraction process of QCOW2 images is similar to the concept of live migration. First the snapshot of the image using native QCOW2 tools, and then converted into raw image. During the conversion, new raw image is created from the differences of the original image and the snapshot. This will prevent the need for suspending the instance for the extraction period, and ensure that all data are extracted. That includes the empty disk space that may contain deleted files, which can also be a valuable source of evidence.

4.2.3 Integrity, Validity, and Encryption

Resources offered by IaaS are virtual, such as the virtual servers, the virtual desktops, the virtual networks, and other virtual devices. In comparison to the traditional forensics, where evidence is stored on physical device, that can always be a the "original" piece of evidence, we don't have such equivalent in the cloud. Finding the right physical disk containing the virtual instance in the data center will be difficult, and same disk may also contain the data of other. Cloud provider simply cannot afford to disrupting the operation of other cloud users. In order to ensure validity, and integrity of the extracted media is preserved, we propose to use the metadata file. Figure 4.6 shows example content of the metadata file. It contain information about the extracted image, such as username of the user who extracted it, date and time of the extraction, file name, and most importantly hash value of the media. In fact the file name is the "instance ID" of the VM in OpenStack. To prevent any tempering with the image,

metadata file is signed by Cloud Provider's private GPG key. Signature can be verified against the Cloud Provider's public GPG key. Meanwhile, the whole image is encrypted by user's password for improved security. This will prevent unauthorized users gaining access to the content of the image. Involved authorities are able to compare the hash of the original image against the hash in the metadata file, to ensure that evidence has not been tampered with.

4.2.4 API Server Details

The server part of the framework uses Flask microframework for Python to expose API endpoints of our Forensic framework (Flask, n.d.), and a BASH program that accept set of parameters from the API server and executes the image extraction.
When user request the image extraction, the API server will locate the instance name in the hypervisor, based on the instance ID from OpenStack provided by the user. The first instance started in OpenStack would be named in the hypervisor as "instance-00000001", and the number in the name is incremented with next instance. However, the name do not correspondent with the instance ID, which is a long random string such as "123456789-89d7-4231-abe6-fa5506a080b2". Our approach to solve this issue is extracting the location of the instance sorage from Nova's configuration file. The location contains folders that are named by instance ID. Each instance folder contains libvirt.xml file, which is a configuration file for that particular instance in KVM hypervisor. From this file we extract instance name, and the location of the associated disk images. API Server uses the instance name in conjunction with "virsh" command to extract the memory images. In case of the disk images, API server identifies the disk image type and apply relevant extraction command for particular disks. In our implementation only the QCOW2 images are extracted using "qemu-img". Details of the virsh and qemu-img commands can be found in the source of the extract.sh program in Appendix A.1.2. After extracting the images into temporary location, hash value of the images is calculated and stored in metadata file. Files are then encrypted by the user's password. We have chosen symmetric password encryption for the simplicity, but it can be exchanged for user's own GPG/PGP key for improved security. Metadata file is signed with Cloud Provider's Private GPG key. The last stage of the process, uploads the metadata file, the encrypted image, and the signature file into the desired Swift container, and deletes the temporary files from the local storage.

```
$curl localhost:9111
Openstack Forensic Framework API

Available API:
/gpg    : Public GPG key for signature verification
/disk   : Extract disk Image
/memory : Extract Memory Image
```

Figure 4.7: Usage info displayed by API call

4.2.5 API Server Installation and Startup

The server consists of two files, api_server.py and extract.sh. Detailed user manual, and the source code can be found in Appendix A.0.1, A.1.1 and A.1.2. Both files are standalone and by design they must be located in the same folder, and both files need to be made executable. In addition to python interpreter, that is installed by default on most of the main Linux distribution, we require installation of the Flask module for Python. Installation of the python modules varies from one distribution to another, and it won't be covered here.
The api_server.py requires superuser privileges, and should be executed from the root shell or using sudo command (sudo ./api_server.py). However, while it is out of the scope of this thesis, it is possible to improve security by creating a user account with privileges limited only to extracting the disk and memory images.
By default the API server is listening on port 9111, but it can be customized in the last section of the source code by changing the value of "port" variable.

4.2.6 API Server Usage

Once the server has been started, the API exposes three endpoints, and the help information displayed when the root of the API server is accessed. Example of the help message is shown in the Figure 4.7. More endpoints can be easily added for additional functionality.
The endpoints are split into two categories as extract, and verify. Verification is done by verifying the gpg signature of the metadata. GPG key can be accessed by HTTP GET methid against the "gpg" endpoint. Example of the GPG key returned by the API server is shown in Figure 4.8. The extraction is done by using the HTTP\POST method against either, "disk", or "memory" endpoint, and passing required parameters in JSON format (JSON, n.d.). While our API client application will post the data in proper format, user can use any HTTP client, such as curl (cURL, n.d.) to communicate with our API server. For the disk and the memory extraction, the server

```
$curl localhost:9111/gpg
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v2.0.22 (GNU/Linux)

mQENBFVYavkBCAC8zdakFOv/3dDmdxiZFLWt3fHeud4d26pDMupm9aOIXWHb68tN
3kYisycj3kvYuwr2ccP5YDgahysFrytAWBEdX+YQOCY92vWeg8AN6gkW54zjvrJt
COJ34avNOoo7tZ59yJD+aIYGSuUp7vYNuvxYXZQGUr9nCZbhksii2GKAuXAC/+uL
K1G58FIa8hWpT3PAFWg/JdDWMlg7jg5MLSMqC0yPejMPX5YsmOQoZJYq7UzfdNCf
NJjV9eAgEyY9G4UuHcEWol7/Y1b+octaDSM+PPux28kEESAeA7sN4VvzOyz/6/7r
wGZcNkR/6bKRbm/bKQlxuBjY6hOv59GsQAT/ABEBAAG0T09wZW5TdGFjayBDbG91
ZCBQcm92aWRlciAoS2V5IGZvciBzaWduaW5nIGZvcmVuc2ljIGRhdGEpIDxzZWN1
cml0eUBleGFtcGxlLm9yZz6JATkEEwECACMFAlVYavkCGwMHCwkIBwMCAQYVCAIJ
CgsEFgIDAQIeAQIXgAAKCRBEGtvYncpvrmUuB/4oElvIyhN9dEc6iyu2Cvh9lGOj
Jh++nzy1cvIAR84y/d1xCZXwBFI++BYgRzWZUalbGmxcivreA9XX1gAlGtO6yNhf
xPPldPmb/m2PnG3kItWIyCbUFRWkQ8a56ItaxWnEpEN5Ya201vBOBrwel/DST08w
U7+Hc/Fg26PnSCmLUzoq9AfX4y/1613MupXHT6rASEmoObRYL7B3MyJk3YxXTwC6
hNjnDb6fbCX15bekjjjOUOFvtsFsZsQj9F/6uM+ST/mMitYHGdiOtd48rzPTjCxf
0rggWCzieq3M7RfdHesATFLOW2nDidOinSxYw0QkAYIYh3bV40LvJDnUFrE7
=CQ2x
-----END PGP PUBLIC KEY BLOCK-----
```

Figure 4.8: Cloud Provider's public GPG key displayed by API call

```
$./api_client.py
Error: missing required options: host ahost action tenant username password container instnceID

Usage: api_client.py [options]

Options:
  -H HOST,       --host=HOST               hostname or IP of forensic API host (required)
  -A AUTH_HOST,  --auth-host=AUTH_HOST     hostname or IP of Keystone API host (required)
  -a ACTION,     --action=ACTION           disk or memory (required)
  -i INSTANCEID, --instance=INSTANCEID     Instance ID (required)
  -t TENANTNAME, --tenant=TENANT           Tenant name (required)
  -c CONTAINER,  --container=CONTAINER     Name of existing Swift container (required)
  -u USERNAME,   --username=USERNAME       Openstack username (required)
  -p PASSWORD,   --password=PASSWORD       Openstack password (required)
  -h,            -help                     Print this help
```

Figure 4.9: API Client Example1: No parameters specified

requires username and password for authentication, and the image encrypting; tenant name, and the instance ID of the extracted VM; and the Swift container name, where the encrypted image, metadata file, and signature file will be stored after extraction.

4.2.7 API Client Usage

API Client is a single python application called api_client.py. Executing the application without parameters will display the usage and required parameters, as shown in 4.9.

In addition, the API client was designed to take advantage of OpenStack's default environment variables such as OS_USERNAME, OS_TENANT_NAME, OS_PASSWORD, and OS_AUTH_URL. Our API client will import and use those variables, if they exist in user's shell and are not provided in the command line.

For more details refer to the user manual in Appendix A.0.1.

Chapter 5

Evaluation

In order to evaluate our approach, we have looked at multiple factors of our forensic framework. Our evaluation consists of two parts, efficiency, and effectiveness. The efficiency evaluation of the collection phase of the forensic process using our framework. We will compare the extraction time of the memory images, to the extraction of the disk images using our forensic framework. In the second part we are evaluating the effectiveness of memory forensics, in comparison to the disk forensics. In this stage of evaluation we are evaluating the examination phase of the forensic process, by attempting to extract evidence of simulated real-world malicious activities, and applying common anti-forensic techniques. Both evaluations aiming to prove that forensics investigation of memory is more efficient, and often ore effective than the disk.

5.1 Test Environment Details

Our test environment consists of single node installation of 11th release of the OpenStack (codename Kilo). Details of the Hardware and Operating systems used are listed in Figure 5.1.

5.1.1 Tools Used

There are many available forensic tools. Some of them are open source, some proprietary, and other available only to the law enforcing institutions. However, there are many standard tools used by system administrators for day to day work, that were not primarily designed as forensic tools, but can be used as such. As the memory image is

OpenStack Server:

CPU:	2x AMD Opteron(tm) Processor 6164 HE (12 Cores)
RAM:	256GB
DISK:	2x 160GB (RAID 1)
Network Storage:	5TB iSCSI volume (for storing disks of the instances)
OS:	Centos 7 (64bit)

Evaluated Instances:

OS:	Ubuntu 14.04 (64Bit)

Table 5.1: Hardware, and Software details

simply a binary file, in our evaluation we will be using tools like "HEX editor" for viewing, and searching the content of the raw binary files. The main tool used throughout this evaluation is an open source "Volatility Framework" (Volatility Foundation, n.d.). Additionally, we have also used other standard Linux, and BASH tools for searching strings in files using regular expressions, and we will refer to them through this chapter.

Volatility Framework

The Volatility framework is a set of tools written in Python, for extraction of digital artefacts from samples of volatile memory. It supports various operating systems, memory image types, and plug-ins for extending its functionality, such as listing running processes, listing open network ports, viewing internet browser history, listing of open files, and much more (Volatility Foundation, n.d.). We will discuss plug-ins used for this evaluation in the relevant sections.

5.2 Efficiency of the Media Collection

5.2.1 Scenario 1: Collection of the Media

In order to analyse efficiency we have measured the time it took to produce individual memory images. We have created multiple instances with different memory sizes, based on the predefined OpenStack instances shown in table 5.2, with exception of the "tiny" instance, as the size of the disk, and memory are too small to support tested OS, and to produce relevant results. Similarly, we have extracted the disk images of the same

instances in order to contrast the time impact of extracting disk, and memory images for for forensic investigation. In order to simulate the servers in production use, we have generated random data to fill up the disks to 90% of their capacity.

Instance	Memory (GB)	Disk (GB)
small	2	20
medium	4	40
large	8	80
xlarge	16	160

Table 5.2: Predefined OpenStack instance types

Results

The table 5.3, and the the chart in figure 5.1 represent the time of the memory, and the disk image extraction, before hash value is generated, and image is encrypted. The results show much faster extraction of the memory comparing to the disk images. This experiment shows that obtaining memory image is more efficient, especially in large instances. Those results were expected, as the memory is typically proportionally smaller than the disk images, and reflects the efficiency of memory forensics of Cloud instances in comparison to the disk forensics. As the disk space requirement are constantly growing, disk offered by cloud providers is increasing. Although, the memory capacity is increasing too, the size is typically substantially smaller. Some providers are offering terabytes of disk space available to the instances, and extraction of such disk image may not be always acceptable. Downloading extensively large images will result also in long download times, and additional storage requirement for storing them locally. In the meantime, typical offered memory of the cloud instance would not exceed 256GB, which can be managed more easily.

Results in the table 5.4, and their visualisation in figure 5.2 reflect the total time of extraction using our forensic framework. This result shows the total time including the hash value generation, and encryption of the images. While generating hash values approximately doubles the time of the extraction, encryption of the images in our environment took about ten times longer than the extraction itself. Potentially we could improve performance of the framework by disabling encryption, and storing the images in more secure location than the Swift block storage. However, it would involve

applying fine grained access control on the storage, that Swift does not support at the moment. With, or without encryption enabled, memory images can be extracted in reasonable time frames. In case of xlarge instance it takes about 10 minutes to extract, hash, and encrypt the memory image of 16GB using our framework. On the other hand, the disk image of the size of 160GB took almost 5 hours to do the same.

Instance	Memory	Disk
small	0.2	3
medium	0.4	7
large	0.8	13
xlarge	1	22

Table 5.3: Time to extract the memory/disk image (in minutes)

Instance	Memory	Disk
small	1	37
medium	3	71
large	5	145
xlarge	10	284

Table 5.4: Total time of extraction, hashing, encryption, and signing of the image (in minutes)

5.3 Effectiveness of the Data Examination

In order to analyse effectiveness of our framework, we have investigated extracted memory, and disk images in different simulated real-world scenarios. We have created multiple instances with Linux operating system. In the first scenario, set of instances runs the Trivial File Transfer Protocol (TFTP) server, simulating typical scenario where malicious user uses TFTP server for automatic uploading of stolen information by malware infected computers. In our test case we have created, and uploaded dummy credit card information, and attempted to recover them. In the second scenario, investigated instance host encrypted Tor website (Tor Project Inc., n.d.), simulating website hosting illegal content in form of JPEG (JPG) images. For each evaluated scenario, we have configured three instances with the same configuration, except for the application, and the data location. First we have stored the services and the data on the local disk of

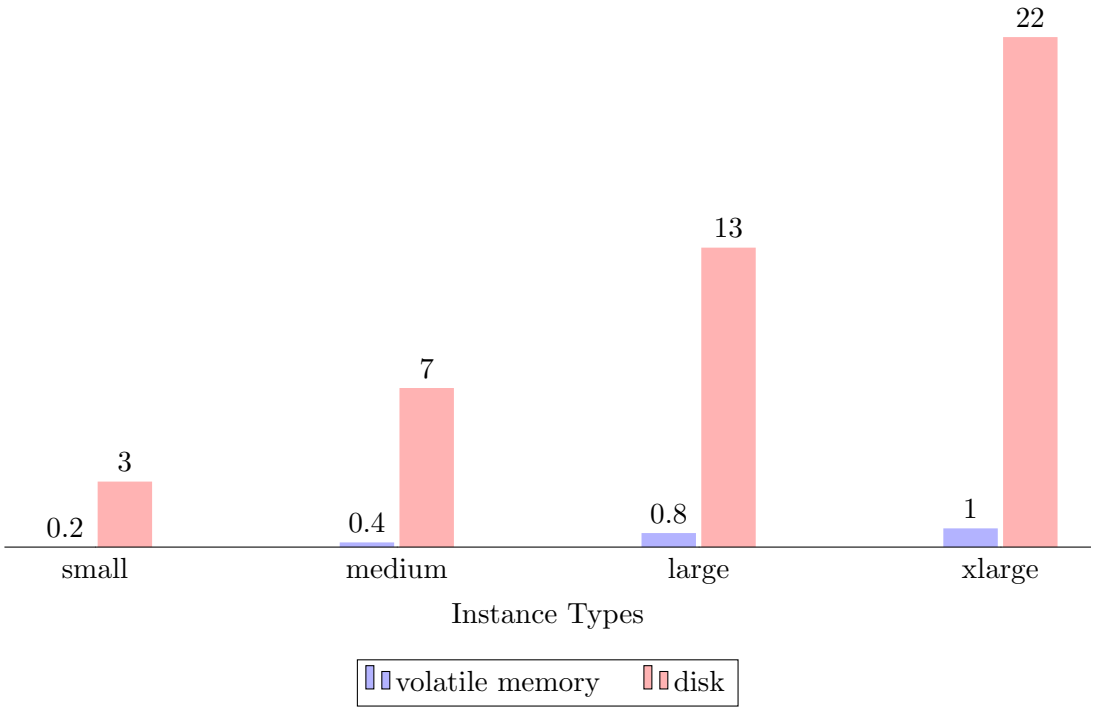

Figure 5.1: Time to extract the memory/disk image (in minutes)

the instance, without applying any anti-forensics techniques. Next instance stores the the service and the data in the network attached storage, and the last instance uses the ramdisk (tmpfs). Forensic investigation of the memory image was performed by the Volatility framework (Volatility Foundation, n.d.), and standard linux tools such as hex editors (for viewing and editing binary files), and pattern searching tools. In each evaluation we will compare the ability to recover evidence from the memory, and the disk images. Our test instances are running Ubuntu 14.04 64bit.

5.3.1 Scenario 2: Stolen Credit Cards

This scenario simulates the TFTP server collecting stolen credit card information from malware infected computers. We have generated two thousand fake credit card numbers, and uploaded them to the TFTP server from four client computers. The generated credit cards are in formats of Master card, and Visa card (16 digit numbers starting with number 5, and number 4). In the first case the data, and the TFTP server binary are located on the local disk in plain sight. For the second case we store it in a network attached storage using Network File System (NFS), and for the last case we have used the ramdisk. For each storage location we evaluate the ability to recover the

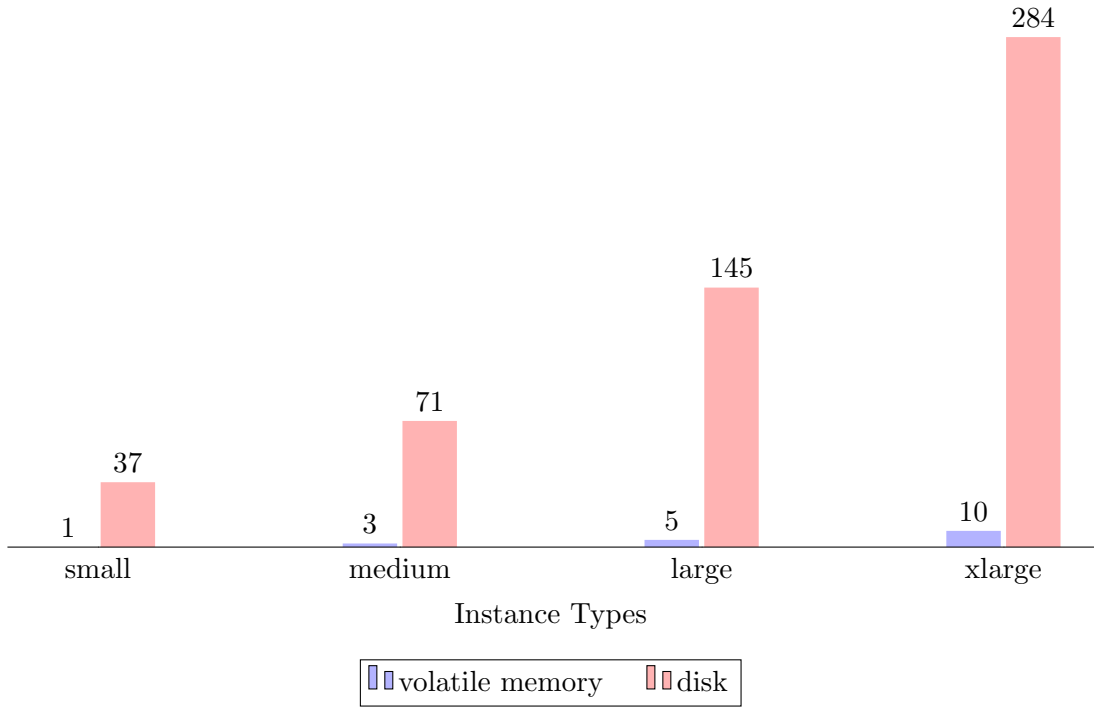

Figure 5.2: Total time of extracting, hashing, encrypting, and signing the image (in minutes)

evidence using memory, and disk forensics. Results shown in the table 5.5 indicates, that by examination of the memory images, we were able to identify the evidence of TFTP server collecting credit card information regardless of the location of the data. However, investigation of the disk images was helpful only in the case of data residing on the local disk.

Data location	Memory forensics	Disk forensics
Local disk	yes	yes
Network storage	yes	no
Ramdisk	yes	no

Table 5.5: Ability to find evidence of stolen credit cards

Details of the Examination

The examination process of the memory images was identical for all three cases of the data location, except for the additional step of extracting the ramdisk. Also, the steps

```
UNIX 11383                 sshd/2457
UNIX 11388                 sshd/2457  /tmp/ssh-J7r4lzpYM5/agent.2457
UNIX 11885                 sudo/2566
UNIX 11887                 sudo/2566
UNIX 11991             in.tftpd/2598
UDP      0.0.0.0         :51301 0.0.0.0         :  723              in.tftpd/2598
```

Figure 5.3: Output of the Volatility plug-in linux_netstat, TFTP

```
2458   1000   1000   -bash
2566   0      1000   sudo -i
2567   0      0      -bash
2598   0      0      /opt/12345/bla/in.tftpd -v -4 -L -a 0.0.0.0:51301 -c -s /opt/12345/out
```

Figure 5.4: Output of the Volatility plug-in linux_psaux

for examining the disk image were the same in all cases.

First we will look at the steps we have taken to evaluate the memory image. After extracting the images, we have used The Volatility Framework (Volatility Foundation, n.d.). We have used the linux_netstat plug-in to view all the ports that server is listening on, and identified that the TFTP service was running with a process ID (PID) 2598, and listening on non-standard port 51301, as shown in figure 5.3. Output of the linux_psaux plug-in in figure 5.4 shows the full command that started the TFTP server, including the location of the binary, output directory, and the PID 2598 that correspond to the previous output. Once we found the location of TFTP binary, and the output folder, we have used the linux_mount plug-in to verify if the folder is stored locally, or mounted from the network, or the ramdisk. In case of where local disk was used, output did not show any additional mount points. However, in case of the network storage, and the ramdisk, we have successfully identified the location, as shown in figure 5.5, and figure 5.6. At this stage we could start investigating NFS server, but this is out of the scope of this evaluation. However, when the ramdisk was used to store (hide) the data, we were able to extract full content of the ramdisk using Volatitity's linux_tmpfs command. In addition, using the linux_arp plug-in, we were able to print the ARP table that contains the IP addresses and MAC addresses of the devices recently connecting to the server, see figure 5.7.

Apart from the Volatility tool, we were able to search pattern of the 16 digit numbers starting with numbers 4 and 5, using standard linux tool "grep". We were able to identify between 1000 and 1600 numbers matching the pattern on each examined image, with false positives between 50 and 100. Recovered card numbers could be

```
none      /run/shm              tmpfs        rw,relatime,nosuid,nodev
udev      /dev                  devtmpfs     rw,relatime
none      /run/user             tmpfs        rw,relatime,nosuid,nodev,noexec
tmpfs     /opt/12345            tmpfs        rw,relatime
none      /sys/fs/pstore        pstore       rw,relatime
devpts    /dev/pts              devpts       rw,relatime,nosuid,noexec
none      /sys/kernel/security  securityfs   rw,relatime
```

Figure 5.5: Output of the Volatility plug-in linux_mount: identifying the Ramdisk

```
none                    /run/shm                tmpfs        rw,relatime,nosuid,nodev
proc                    /proc                   proc         rw,relatime,nosuid,nodev,noexec
192.168.122.212://export/temp /opt/12345              nfs          rw,relatime
udev                    /dev                    devtmpfs     rw,relatime
systemd                 /sys/fs/cgroup/systemd  cgroup       rw,relatime,nosuid,nodev,noexec
none                    /run/user               tmpfs        rw,relatime,nosuid,nodev,noexec
none                    /sys/kernel/debug       debugfs      rw,relatime
```

Figure 5.6: Output of the Volatility plug-in linux_mount: identifying the network storage

potentially validate against the card provider, and use it as an evidence.
The search pattern may need to be tweaked in real-world forensic examination to match different formats of card numbers, such as being stored in four groups of four digits separated by either space, hyphen, or other characters to gain the same results.

Examination of disk images was more time consuming, it can take hours to inspect Gigabytes of data on the disk. Disk investigation is mostly based on pattern matching, or manual inspection of the files. It is performed either on the disk content, recovered deleted data, or both. Our examination was done by searching the log files on the disk, and searching for the same search pattern of the credit card number as in previous case, but on the whole disk. However, we were only able to find the uploaded files, when the output folder it was located on the investigated disk image. As disk contained no evidence of TFTP server running, there was neither evidence of the TFTP command that would help identify the output folder. Therefore, we had to search the whole disk image to locate the uploaded files containing credit card details. When ramdisk, and network storage was used, we were not able to found any evidence of malicious activity. Although, in case of network storage, data still resides on the physical storage of the network device, we were not able to find any pointers leading to the location of the network storage, or even evidence that network storage has been used.

```
Volatility Foundation Volatility Framework 2.4
[ff02::2                                ] at 33:33:00:00:00:02    on eth0
[ff02::1:ff02:9967                      ] at 33:33:ff:02:99:67    on eth0
[ff02::16                               ] at 33:33:00:00:00:16    on eth0
[192.168.122.195                        ] at 52:54:00:2e:29:95    on eth0
[192.168.122.237                        ] at 52:54:00:28:f0:54    on eth0
[192.168.122.1                          ] at 52:54:00:05:33:11    on eth0
[192.168.122.218                        ] at 52:54:00:6d:6e:57    on eth0
[192.168.122.204                        ] at 52:54:00:0b:85:f3    on eth0
```

Figure 5.7: Output of the Volatility plug-in linux_arp

5.3.2 Scenario 3: Hidden Service

Our next scenario simulates website hidden by Tor (Tor Project Inc., n.d.). In real world that server would be equivalent of a server hosting illegal pictures. In our case those pictures are represented by random images of cats. Tor is a software that creates encrypted network, where each client acts as a node. Encrypted traffic is then routed through the random nodes, hiding the identity of the users. Tor also supports running hidden service without revealing its IP address, and the service can be only accessed via Tor network using the randomly generated hostname with the ".onion" top level domain (Tor Project Inc., n.d.). For this experiment we have compiled standalone Apache HTTP server (Web server) (The Apache Software Foundation, n.d.*b*), instead of the default package distributed by Ubuntu, in order to be able to hide it in ramdisk, and network storage. Web server is serving simple image gallery with 6 sample images in JPEG format. Once the server was started, we have accessed those images from our four client computers to simulate the web traffic. We have then extracted disk and memory images, and install two additional servers with identical configuration, but mounting a network storage, and a ramdisk, where we stored the web server, and all the website content. Similarly to the previous experiment, we have evaluated the ability to identify existence of the hidden web server, and obtain the evidence of illegal content.

Results in table 5.6 are identical to the previous evaluated scenario. Memory forensics helped to identify, and recover the evidence in all cases. On the contrary disk forensics was effective only in case of files stored on the investigated disk.

Details of the Examination

As in the previous experiment, we have used Volatility framework to identify the web server, and Tor service running on the instance. Example of the netstat output is displayed in figure 5.8. We were able to identify storage mount points, and detect

Data location	Memory forensics	Disk forensics
Local disk	yes	yes
Network storage	yes	no
Ramdisk	yes	no

Table 5.6: Ability to find evidence of illegal web content

```
UDP      0.0.0.0         :30332 0.0.0.0         :  737                          dhclient/634
UDP      ::              :57558 ::              :  949                          dhclient/634
UNIX 8905                 acpid/795
UNIX 8908                 acpid/795   /var/run/acpid.socket
TCP      0.0.0.0         :   22 0.0.0.0         :    0 LISTEN                      sshd/799
TCP      ::              :   22 ::              :52760 LISTEN                      sshd/799
TCP      192.168.122.211 :60801 5.39.89.124     :    0 ESTABLISHED                  tor/858
UNIX 9016                   tor/858
UNIX 9017                   tor/858
TCP      127.0.0.1       : 9050 0.0.0.0         :    0 LISTEN                       tor/858
UNIX 9021                   tor/858   /var/run/tor/control
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/920
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/929
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/930
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/931
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/932
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/933
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/1080
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/1083
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/1084
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/1085
TCP      127.0.0.1       :   80 0.0.0.0         :    0 LISTEN                  apache2/1087
```

Figure 5.8: Output of the Volatility plug-in linux_netstat, Hidden Service

location of the data, extract content of the ramdisk, and obtain the location of the network storage. However, as the instance was configured with Tor hiding traffic from, and to the Tor encrypted network, we were not able to find IP addresses or MAC addresses of the clients connecting to the web server. When web server was running, and storing data from ramdisk, using the linux_tmpfs command from Volatility Framework to extract full ramdisk and obtain all JPEG images served by the server.

The main difference in the examination of this scenario was the type of data we were looking for. While in the credit card scenario we have been searching for text patterns, the web server serves more than just text. Typical web site may contain different types of content, such as simple text, pictures, audio, video, and virtually any type of files. In this evaluation we are looking for JPEG images, or traces of the JPEG image files that could be used as an evidence of illegal content.

Content of the JPEG images is binary, and as most of standard file types, they contain binary signature. The binary signature is stored in the beginning of the file, helping

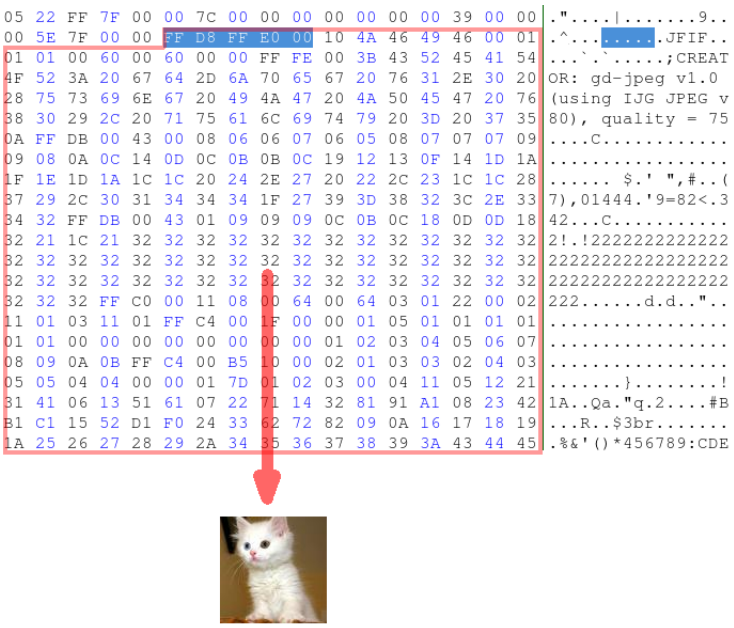

Figure 5.9: JPEG signature and content of the file found in the memory image

applications and operating systems to identify the file type. In case of JPEG images, the signature in hexadecimal format is "FF D8 FF E0". Based on this signature we were able to find images loaded by the web server in all evaluated memory images. Figure 5.9 shows the JPEG signature, and the content of the file found in the memory image, with the corresponding image below. Regardless of the data location in our three cases, using this technique we were able to obtain all images, that were accessed by our client machines simulating web traffic.

Examination of the disk images led to the results that we were expecting based on the previous experiment. Searching disk for image files resulted in thousands of false positives, including pictures, icons, wallpapers, and other artwork used in applications, and operating system. Locating the evidence required manual examination of every picture. In cases when ramdisk, or network storage was used, disk examination did not uncovered any evidence of neither web server, Tor service, or the files served by the web server (our simulated illegal content).

Chapter 6

Conclusions

Amount of information that can be acquired from the memory, makes it an attractive source of forensic data. Especially in cloud environments such as IaaS, where memory image can be relatively easily extracted from the running VM. Our evaluation showed that memory forensics improves the overall process of obtaining forensic evidence in cloud environment. Although, disk images can be also easily obtained, growing disk sizes and capacities of offered storage can make disk investigation very time, and resource consuming task. In cloud environment sizes of disk images can vary from few of gigabytes (10^9 bytes) to couple of terabytes (10^{12} bytes). Regardless of how long the investigation of large disks will take, downloading such an image from cloud may take hours, or days, and can generate extra cost. In the meantime, our evaluation have proven that in cases where anti-forensic techniques are used such as disk encryption, hiding data in ramdisk, or in network storage, disk forensics is often useless. We have proven that collection, and examination of the memory images is more efficient, and effective in the cloud environment than disk images. While disk may contain larger volumes of the data and potentially also more evidence, the evidence found in the volatile memory is often sufficient. In rare cases when memory does not contain the evidence, it generally contains valuable information that can lead to faster identification of the evidence in other locations, such as local, or network storage. Memory forensics may not replace other forensic techniques such as network forensics, log analysis, or even disk investigation, but it improves the overall forensic process. Memory images can substantially reduce the time of identifying, and obtaining evidence in cloud environment. Cloud providers, particularly the ones providing private, or community cloud have to rely on their own expertise in solving forensic issues. This could be an issue for smaller organisations with limited resources. Our Framework simplifies the media collection for both the cloud

provider, and the cloud user. While forensic frameworks should be developed as a part of the cloud platforms, ability to do the memory forensics should be definitely part of it.

6.1 Future work

Our API server has been written to be compatible with Horizon's design, and can be integrated without any modification into API server itself. However, dashboard integration is out of the scope of this thesis. We are planing to expand the framework into the dashboard and propose the whole product to the community for future integration into OpenStack platform.

We will investigate faster algorithms for hashing and encryption of the images in order to improve overall performance of the framework. Meanwhile, we plan to expand the support for different storage types, and hypervisors used in OpenStack. In order to provide complete solution, we will investigate potential integration of network forensics, and ability to access relevant OpenStack logs.

During the efficiency evaluation, we have observed negative effect on the extracted instances. In order to extract memory images, the hypervisor must suspend the instance for short period of time. In case of certain types of storage, the instance must be suspended for the whole duration of the extraction. Different memory, and disk sizes take longer to extract, and this can have negative effect on the instance. We have observed the issue with system time drifting when instance is suspended. This may have negative effect on instance, that is using time sensitive services, such as Kerberos (MIT Kerberos, n.d.), SAML (OASIS consortium, n.d.), and other time sensitive authentication services that rely on the system time. Time can corrected automatically if the instance has implemented NTP client (Network Time Protocol). However, better techniques should be researched, such as synchronising the system time using the hypervisor.

Bibliography

Amazon S3 (n.d.), 'Amazon S3 Website', http://aws.amazon.com/s3/. [Online; accessed 2015-03-21].

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I. et al. (2010), 'A view of cloud computing', *Communications of the ACM* **53**(4), 50–58.

Biggs, S. and Vidalis, S. (2009), Cloud computing: The impact on digital forensic investigations, *in* 'Internet Technology and Secured Transactions, 2009. ICITST 2009. International Conference for', IEEE, pp. 1–6.

Birk, D. and Wegener, C. (2011), Technical issues of forensic investigations in cloud computing environments, *in* 'Systematic Approaches to Digital Forensic Engineering (SADFE), 2011 IEEE Sixth International Workshop on', IEEE, pp. 1–10.

Chen, H.-Y. (2014), Cloud crime to traditional digital forensic legal and technical challenges and countermeasures, *in* 'Advanced Research and Technology in Industry Applications (WARTIA), 2014 IEEE Workshop on', pp. 990–994.

cURL (n.d.), 'cURL official website', http://curl.haxx.se/. [Online; accessed 2015-08-05].

Delport, W., Köhn, M. and Olivier, M. S. (2011), Isolating a cloud instance for a digital forensic investigation., *in* 'ISSA'.

Denning, D. (1987), 'An intrusion-detection model', *Software Engineering, IEEE Transactions on* **SE-13**(2), 222–232.

Dykstra, J. and Sherman, A. T. (2012), 'Acquiring forensic evidence from infrastructure-as-a-service cloud computing: Exploring and evaluating tools, trust, and techniques', *Digital Investigation* **9**, S90–S98.

Dykstra, J. and Sherman, A. T. (2013), 'Design and implementation of frost: Digital forensic tools for the openstack cloud computing platform', *Digital Investigation* **10**, S87–S95.

Eucalyptus Systems, Inc. (n.d.), 'Eucalyptus official Website', https://www.eucalyptus.com/. [Online; accessed 2014-12-16].

Fei, B. K. L. (2007), Data visualisation in digtial forensics, PhD thesis, University of Pretoria.

Flask (n.d.), 'Flask microframework for Python', http://flask.pocoo.org/. [Online; accessed 2015-08-05].

Foster, I., Zhao, Y., Raicu, I. and Lu, S. (2008), Cloud computing and grid computing 360-degree compared, *in* 'Grid Computing Environments Workshop, 2008. GCE'08', Ieee, pp. 1–10.

Free Software Foundation, Inc. (n.d.), 'Bourne Again SHell', http://www.gnu.org/software/bash/. [Online; accessed 2015-08-02].

Garfinkel, S. L. (2010), 'Digital forensics research: The next 10 years', *Digital Investigation* **7**, S64–S73.

Geelan, J. (2009), 'Twenty-one experts define cloud computing', *Cloud Computing Journal* **4**, 1–5.

Grobauer, B. and Schreck, T. (2010), Towards incident handling in the cloud: challenges and approaches, *in* 'Proceedings of the 2010 ACM workshop on Cloud computing security workshop', ACM, pp. 77–86.

Guo, H., Jin, B. and Shang, T. (2012), Forensic investigations in cloud environments, *in* 'Computer Science and Information Processing (CSIP), 2012 International Conference on', IEEE, pp. 248–251.

Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J. and Felten, E. W. (2009), 'Lest we remember: cold-boot attacks on encryption keys', *Communications of the ACM* **52**(5), 91–98.

Jahankhani, H. and Beqiri, E. (2010), 'Memory based anti-forensic tools and techniques', *Pervasive Information Security and Privacy Developments: Trends and Advancements: Trends and Advancements* p. 184.

JSON (n.d.), 'JSON official website', http://json.org/. [Online; accessed 2015-08-05].

Kent, K., Chevalier, S., Grance, T. and Dang, H. (2006), Sp 800-86. guide to integrating forensic techniques into incident response, Technical report, Gaithersburg, MD, United States.

KVM Community (n.d.), 'QCOW2: KVM Hypervizor Official Website', http://www.linux-kvm.org/page/Qcow2. [Online; accessed 2015-08-01].

Libvirt virtualization API (n.d.), 'Livbirt official Website', http://libvirt.org/. [Online; accessed 2015-03-21].

Lo, C.-C., Huang, C.-C. and Ku, J. (2010), A cooperative intrusion detection system framework for cloud computing networks, *in* 'Parallel Processing Workshops (ICPPW), 2010 39th International Conference on', pp. 280–284.

Mell, P. and Grance, T. (2009), 'The nist definition of cloud computing', *National Institute of Standards and Technology* **53**(6), 50.

MIT Kerberos (n.d.), 'Kerberos: The Network Authentication Protocol', http://web.mit.edu/kerberos/. [Online; accessed 2015-03-28].

Nicanfar, H., Liu, Q., Talebifard, P., Cai, W. and Leung, V. (2013), Community cloud: Concept, model, attacks and solution, *in* 'Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on', Vol. 2, IEEE, pp. 126–131.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L. and Zagorodnov, D. (2009), The eucalyptus open-source cloud-computing system, *in* 'Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium on', pp. 124–131.

OASIS consortium (n.d.), 'Security Assertion Markup Language Specification', http://saml.xml.org/saml-specifications. [Online; accessed 2015-03-28].

Patrascu, A. and Patriciu, V.-V. (2013), Beyond digital forensics. a cloud computing perspective over incident response and reporting, *in* 'Applied Computational Intelligence and Informatics (SACI), 2013 IEEE 8th International Symposium on', pp. 455–460.

Patrascu, A. and Patriciu, V.-V. (2014), Logging framework for cloud computing forensic environments, *in* 'Communications (COMM), 2014 10th International Conference on', pp. 1–4.

Poisel, R., Malzer, E. and Tjoa, S. (2013), 'Evidence and cloud computing: The virtual machine introspection approach', *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* **4**(1), 135–152.

Prosise, C., Mandia, K. and Pepe, M. (2003), *Incident response & computer forensics*, McGraw-Hill/Osborne.

Python Software Foundation (n.d.), 'Python', https://www.python.org/. [Online; accessed 2015-08-02].

QEMU emulator and virtualizer (n.d.), 'QEMU official Website', http://wiki.qemu.org/Main_Page. [Online; accessed 2015-03-21].

Roschke, S., Cheng, F. and Meinel, C. (2009), Intrusion detection in the cloud, *in* 'Dependable, Autonomic and Secure Computing, 2009. DASC '09. Eighth IEEE International Conference on', pp. 729–734.

Ruan, K., Carthy, J., Kechadi, T. and Crosbie, M. (2011), Cloud forensics, *in* 'Advances in digital forensics VII', Springer, pp. 35–46.

Saibharath, S. and Geethakumari, G. (2014), Design and implementation of a forensic framework for cloud in openstack cloud platform, *in* 'Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on', pp. 645–650.

Sang, T. (2013), A log based approach to make digital forensics easier on cloud computing, *in* 'Intelligent System Design and Engineering Applications (ISDEA), 2013 Third International Conference on', pp. 91–94.

Shaw, A., Bordbar, B., Saxon, J., Harrison, K. and Dalton, C. (2014), Forensic virtual machines: Dynamic defence in the cloud via introspection, *in* 'Cloud Engineering (IC2E), 2014 IEEE International Conference on', pp. 303–310.

Subramanian, K. (2011), 'Public clouds', *A whitepaper sponsored by Trend Micro Inc* .

The Apache Software Foundation (n.d.*a*), 'Apache Cloudstack official Website', http://cloudstack.apache.org/. [Online; accessed 2014-12-16].

The Apache Software Foundation (n.d.*b*), 'Apache HTTP Server Website', http://httpd.apache.org/. [Online; accessed 2014-12-16].

The OpenStack Foundation (n.d.), 'OpenStack official Website', http://www.openstack.org. [Online; accessed 2014-12-16].

The Volatility Foundation (n.d.), 'Volatility Foundation Official Website', http://www.volatilityfoundation.org/. [Online; accessed 2015-08-02].

Thethi, N. and Keane, A. (2014), Digital forensics investigations in the cloud, *in* 'Advance Computing Conference (IACC), 2014 IEEE International', pp. 1475–1480.

Thorpe, S., Grandison, T. and Blake, M. (2014), Cloud computing log forensics-the new frontier, *in* 'SOUTHEASTCON 2014, IEEE', pp. 1–4.

Thorpe, S., Grandison, T. and Ray, I. (2012), Cloud computing log evidence forensic examination analysis, *in* 'Proceedings of the 2nd International Conference on CyberCrime, Security and Digital Forensics'.

Tor Project Inc. (n.d.), 'Tor Project Website', https://www.torproject.org/. [Online; accessed 2015-03-22].

Vacca, J. R. (2002), *Computer forensics: computer crime scene investigation*, Charles River Media, Inc.

Volatility Foundation (n.d.), 'Volatility Foundation', https://github.com/volatilityfoundation. [Online; accessed 2015-03-22].

Zargari, S. and Benford, D. (2012), Cloud forensics: Concepts, issues, and challenges, *in* 'Emerging Intelligent Data and Web Technologies (EIDWT), 2012 Third International Conference on', IEEE, pp. 236–243.

Zawoad, S. and Hasan, R. (2013), 'Cloud forensics: a meta-study of challenges, approaches, and open problems', *arXiv preprint arXiv:1302.6312* .

Appendix A

Appendix

A.0.1 User Manual

Requirements

- OpenStack (Juno, or Kilo) with KVM hypervisor, and QCOW2 images
- Python version 3x
- BASH dependencies for API Server: gnugpg, xmllint
- Python modules for API Server: flask, request, jsonify, json, requests, subprocess, sys, os
- Python modules for API Client: requests, json, sys, getopt, os

Installation

Create new folder in desired location (for example /opt/ff), and create gpg folder within:

```
1 mkdir -p /opt/ff/gpg
```

Copy the api_server.py, and extract.sh into the newly created folder and make them executable:

```
1 cp api_server.py extract.sh /opt/ff
2 chmod +x /opt/ff/{api_server.py,extract.sh}
```

Create cloud providers GPG key pair for signing the metadata:

```
gpg --gen-key
```

Choose type of the key, select 3 or 4 if this key will be used for signing only.

```
gpg (GnuPG) 2.0.22; Copyright (C) 2013 Free Software Foundation, Inc.
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.

Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection?
```

Choose the keysize, larger key will improve security, but increase the signing time.

```
DSA keys may be between 1024 and 3072 bits long.
What keysize do you want? (2048)
```

Specify the validity period for the key (For security reasons key should have expiry):

```
Please specify how long the key should be valid.
        0 = key does not expire
     <n>  = key expires in n days
     <n>w = key expires in n weeks
     <n>m = key expires in n months
     <n>y = key expires in n years
Key is valid for? (0)
```

Specify the real name (Cloud Provider), the email address, and comments associated with the key. Review the entry and select "O" to indicate Okay:

```
GnuPG needs to construct a user ID to identify your key.

Real name: NCI Thesis OpenStack Cloud Provider
Email address: cloud@example.org
Comment: Key for signing metadata

You selected this USER-ID:
    "NCI Thesis OpenStack Cloud Provider (Key for signing metadata) <cloud@example. ↩
        org>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit?
```

In the next screen choose the password, and confirm.
Once key is generated, verify that the key pair exist in our GPG keyring:

```
gpg --list-keys

pub 2048D/0B59790D 2015-08-29 [expires: 2015-09-28]
uid  NCI Thesis OpenStack Cloud Provider (Key for signing metadata) <cloud@example. ↩
    org>
```

Extract the public, and private keys into ascii text files, specifying the key using the email address:

```
gpg --armor --export cloud@example.org > /opt/ff/gpg/gpg_key.public
gpg --armor --export-secret-key cloud@example.org> /opt/ff/gpg_key.private
```

Edit the extract.sh and change the GPGPASS variable with the password used for creating the GPG key pair. Also update the TEMP_FOLDER variable, to specify the temporary folder where the output files will be manipulated before uploading to the Swift container, the folder should be big enough to store the raw disk and memory images.

By default API server listen on port 9111. Port can be changed in the bottom of the api_server.py:

```
# listen on http://0.0.0.0:9111
if __name__ == '__main__':
    app.debug = True
    app.run(
        host='0.0.0.0',
        port=9111
    )
```

Now we are ready to start the API server.

Starting the server

API server should be started as root user, as it need access to the hypervisor, and the images stored by libvirt (it may be possible to add user into groups that have read access to the required files, but we haven't tested it):

```
cd /opt/ff
sudo ./api_server.py
```

API Calls

API Server exposes three API endpoints:

```
/gpg    : Public GPG key for signature verification
/disk   : Extract disk Image
/memory : Extract Memory Image
```

The “gpg” endpoint prints the cloud provider’s GPG key for signature verification. The “disk” and “memory” endpoints are used to extract disk, and memory images. Additionally, accessing the ROOT of the API server “” prints the available endpoints.

Images can be extracted without API Client, using any HTTP client capable of posting JSON formatted data. In this example we use cURL application, which is very common on most of the Linux distributions.

First we need to obtain the authentication token from Keystone Identity Service running on server with IP 192.168.122.10:

```
curl -s -X POST http://192.168.122.10:5000/v2.0/tokens -H "Content-Type: application ↩
    /json" -d '{"auth": {"tenantName": "'"demo"'", "passwordCredentials": {"username ↩
    ": "'"admin"'", "password": "'"Secret123"'"}}}'| python -m json.tool
```

Keystone response will contain token block similar to this:

```
       "token": {
           "audit_ids": [
               "BmU6dQNMT520b9lJvzMJMg"
           ],
           "expires": "2015-08-23T16:14:46Z",
           "id": "f19283dd8a254b43af0aacdb431a8147",
           "issued_at": "2015-08-23T15:14:46.293570",
           "tenant": {
               "description": "default tenant",
               "enabled": true,
               "id": "a196f9b39bc44c06aaae9e4949e19ecb",
               "name": "demo"
           }
       },
```

Token itself is a string in the “id” section after the “expires” entry, which indicates the expiry of the token (typically 1 hour).

For extraction of the disk, and memory images we need an instance ID, which is an identifier of an instance in Openstack. Using the Horizon dashboard or OpenStack API tools we can list the instances, obtain the instance ID of the instance we are interested in extracting. From command line execute following command:

```
1 nova list
```

We also need to create Swift container where the output files will be stored, either in Horizon or from command line using the OpenStack API. Example: following command will create container "forensics" if it doesn't exist yet:

```
1 swift post forensics
```

At this stage we are ready to start the image extraction. Command to extract memory should look something like this:

```
1 curl -H "Content-type: application/json" -X POST http://192.168.122.10:9111/memory - ↩
    d '{"tenant": "demo", "username": "admin", "password": "Secret123", "container": ↩
     "forensics", "instanceid": "f7d6273b-fe0e-42ae-9dd8-c8a539cc1d95"}'
```

or the disk image:

```
1 curl -H "Content-type: application/json" -X POST http://192.168.122.10:9111/disk -d  ↩
    '{"tenant": "demo", "username": "admin", "password": "Secret123", "container": " ↩
    forensics", "instanceid": "f7d6273b-fe0e-42ae-9dd8-c8a539cc1d95"}'
```

Command will either return message "Extracting memory/disk image", or return an error message.

After extraction is finished, image, metadata file, and signature will be available in the "forensics" container. Following command display the content of the swift container:

```
1 swift list forensics
```

Output of the command should be similar to this:

```
1 admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378/memory-2b9036db-15e9-488 ↩
    b-b0fd-76ffa1c34378.img.gpg
2 admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378/memory-2b9036db-15e9-488 ↩
    b-b0fd-76ffa1c34378.metadata
3 admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378/memory-2b9036db-15e9-488 ↩
    b-b0fd-76ffa1c34378.sig
```

Files can be downloaded also via Horizon dashboard or using the following command:

```
swift download forensics admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378 ↩
    /memory-2b9036db-15e9-488b-b0fd-76ffa1c34378.img.gpg
swift download forensics admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378 ↩
    /memory-2b9036db-15e9-488b-b0fd-76ffa1c34378.metadata
swift download forensics admin/20150604-1646-19/2b9036db-15e9-488b-b0fd-76ffa1c34378 ↩
    /memory-2b9036db-15e9-488b-b0fd-76ffa1c34378.sig
```

API Client

API Client was designed to simplify the process described in previous section. It eliminates the need for crafting the JSON message, or obtaining the authentication token from Keystone. Installation is not required, simply copy the api_client.py file into desired location and make it executable, or execute it with python.

```
chmod +x api_client.py
./api_client.py
```

Or

```
python ./api_client.py
```

Command will return an error message and usage instructions:

```
Error: missing required options: host action container instnceID

Usage: api_client.py [options]

Options:
  -H HOST,      --host=HOST            hostname or IP of forensic API host (required ↩
      )
  -A AUTH_HOST, --auth-host=AUTH_HOST  hostname or IP of Keystone API host (required ↩
      )
  -a ACTION,    --action=ACTION        disk or memory (required)
  -i INSTANCEID, --instance=INSTANCEID Instance ID (required)
  -t TENANTNAME, --tenant=TENANT       Tenant name (required)
  -c CONTAINER, --container=CONTAINER  Name of existing Swift container (required)
  -u USERNAME,  --username=USERNAME    Openstack username (required)
  -p PASSWORD,  --password=PASSWORD    Openstack password (required)
  -h,           -help                  Print this help

Examples:
  api_client.py -H http://192.168.1.1:9111 -a disk -t demo -i 12345678-9abc-def1 ↩
      -2345-67890abcdef1 -c output -u demo -p demo
  api_client.py -H http://192.168.1.1:9111 -a memory -t demo -i 12345678-9abc-def1 ↩
      -2345-67890abcdef1 -c output -u demo -p demo
```

Examples in the output of the command are self-explanatory. The API Client also supports standard OpenStack environment variables in example bellow. If those variables exist in the system, parameters such username, password, tenant name, and Keystone API Url can be omitted.

```
export OS_USERNAME=admin
export OS_TENANT_NAME=demo
export OS_PASSWORD=Secret123
export OS_AUTH_URL=http://192.168.122.10:5000/v2.0/
```

In addition, we support own environment variable for "OS_FORENSICS_URL", that provides the url of the API server.

```
export OS_FORENSICS_URL=http://192.168.122.10:9111
```

If all environment variables are used, API client command for extracting memory will look like this:

```
./api_client -a disk -t demo -i 12345678-9abc-def1-2345-67890abcdef1 -c forensics
```

A.1 Source code

A.1.1 api_server.py

```
#!/usr/bin/env python
#
# API server for Openstack Forensic framework
# extracts disk and memory images of OpenStack instances
#

from flask import Flask, request, jsonify, json
import requests, subprocess, sys, os

# check if user is root
if os.geteuid() != 0:
   exit("You need to have root privileges to run this script.\nPlease try again, ↩
       this time using 'sudo'. Exiting.")

app = Flask(__name__)

# check if container exists
def check_container( token, accid, container ):
    # hardcoded url, will work only on single node openstack
```

```
19      url = 'http://127.0.0.1:8080/v1/AUTH_'+ str(accid) +'/'+ str(container) +'? ↩
            format=json'
20      headers = {'X-Auth-Token': str(token)}
21      r = requests.get(url, headers=headers)
22      return r.status_code
23
24  # check if instance exists
25  def check_instance( token, accid, instanceId ):
26      # hardcoded url, will work only on single node openstack
27      url = 'http://127.0.0.1:8774/v2/'+ str(accid) +'/servers/'+ str(instanceId)
28      headers = {'X-Auth-Token': str(token)}
29      r = requests.get(url, headers=headers)
30      return r.status_code
31
32  # extract disk image - call bash script with parameters:
33  # create metadata file: instance name, md5 hash, creation date, username, ip  ↩
        addresses
34  # store it in swift container
35  def extract_img( instanceId, tenant, username, token, accid, container, mediatype,  ↩
        password ):
36      ret_check_instance = check_instance( token, accid, instanceId )
37      ret_check_container = check_container( token, accid, container )
38      if ret_check_instance == 200:
39          if ret_check_container == 200:
40              # extract image
41              subprocess.Popen(["./extract.sh", mediatype, instanceId, container,  ↩
                    tenant, username, password])
42              return True
43          else:
44              return 'Container Error, check container Name'
45      else:
46          return 'Instance Error, check instance ID'
47
48  def extract_media( instanceId, tenant, username, token, accid, container, mediatype, ↩
        password ):
49  # check if variables are empty
50      if tenant == '':
51          return 'tennant not defined \n'
52      if username == '':
53          return 'username not defined \n'
54      if token == '':
55          return 'token is missing \n'
56      if accid == '':
57          return 'account (accid) id is missing \n'
58      if instanceId == '':
59          return 'instanceid not defined \n'
60      if container == '':
```

```
61          return 'container not defined \n'
62      if password == '':
63          return 'password not defined \n'
64
65  # extract disk immages
66      ret_extract_img = extract_img( instanceId, tenant, username, token, accid,  ↩
            container, mediatype, password )
67
68      if ret_extract_img == True:
69          msg = 'Extracting '+mediatype+' image'
70      else:
71          msg = 'Extracting '+mediatype+' image failed. '+ret_extract_img
72      return msg
73
74  # print message on root path
75  @app.route('/')
76  def api_root():
77      return 'Openstack Forensic Framework API\n\nAvailable API:\n/gpg : Public GPG  ↩
            key for signature verification\n/disk : Extract disk Image\n/memory :  ↩
            Extract Memory Image\n'
78
79  # return public key used for verifying signed metadata file
80  @app.route('/gpg')
81  def api_gpg():
82      gpg_key = open('gpg/gpg_key.public', 'r')
83      return gpg_key.read()
84
85  # api for extracting disk image
86  @app.route('/disk', methods = ['POST'])
87  def api_extract_disk():
88      if request.headers['Content-Type'] == 'application/json':
89      # set media type for extraction
90          mediatype = 'disk'
91          # Get the parsed contents of the form data
92          username = request.get_json().get('username', '')
93          tenant = request.get_json().get('tenant', '')
94          token = request.get_json().get('token', '')
95          accid = request.get_json().get('accountid', '')
96          instanceId = request.get_json().get('instanceid', '')
97          container = request.get_json().get('container', '')
98          # for this implementation I'm getting openstack password, to encript the  ↩
                image,
99          # but it could be additional variable for custom gpg password
100         password = request.get_json().get('password', '')
101
102         # extract media
```

```
103         ret_extract = extract_media( instanceId, tenant, username, token, accid,  ↩
                container, mediatype, password )
104
105         return ret_extract+'\n'
106     else:
107         return '415 Unsupported Media Type, expecting JSON\n'
108
109 # api for extracting memory image
110 @app.route('/memory', methods = ['POST'])
111 def api_extract_mem():
112     if request.headers['Content-Type'] == 'application/json':
113         # set media type for extraction
114         mediatype = 'memory'
115         # Get the parsed contents of the form data
116         username = request.get_json().get('username', '')
117         tenant = request.get_json().get('tenant', '')
118         token = request.get_json().get('token', '')
119         accid = request.get_json().get('accountid', '')
120         instanceId = request.get_json().get('instanceid', '')
121         container = request.get_json().get('container', '')
122         # for this implementation I'm getting openstack password, to encript the  ↩
                image,
123         # but it could be additional variable for custom gpg password
124         password = request.get_json().get('password', '')
125         # extract media
126         ret_extract = extract_media( instanceId, tenant, username, token, accid,  ↩
                container, mediatype, password )
127         return ret_extract+'\n'
128     else:
129         return '415 Unsupported Media Type, expecting JSON\n'
130
131 # listen on http://0.0.0.0:9111
132 if __name__ == '__main__':
133     app.debug = True
134     app.run(
135         host='0.0.0.0',
136         port=9111
137     )
```

A.1.2 extract.sh

```
1 #!/bin/bash
2
3 # script for extracting disk and memory images from openstack/libvirt
4 # api_server.py execute it like this:
5 # ./extract_disk.sh disk|memory instanceId container tenant username password
```

```
6
7  # change temp locations here
8  TEMP_FOLDER="/tmp/forensics"
9
10 # using current date to create output folder and avoid conflicts when extracting ↩
       same image multiple times
11 CURRENTDATE="$(date +%Y%m%d-%H%M-%S)"
12
13 # parameters are supposed to be passed by api_server.py in particular order
14 export PARM1=$1
15 export INSTANCEID=$2
16 export CONTAINER=$3
17 export OS_TENANT_NAME=$4
18 export OS_USERNAME=$5
19 export OS_PASSWORD=$6
20 export GPGPASS="Passw0rd"
21 export AUTH_URL="http://127.0.0.1:5000/v2.0"
22
23 # get location of instance path from nova.conf
24 NOVA_STORE=$(grep instances_path /etc/nova/nova.conf |grep -v ^# |sed 's/ ↩
       instances_path//g;s/\=//g;s/\ //g')
25 # extract libvirt name of the instance from xml definition og the VM
26 INSTANCE_NAME=$(grep \<name\> $NOVA_STORE/$INSTANCEID/libvirt.xml |sed 's/<name>//g; ↩
       s/<\/name>//g;s/\ //g')
27
28 # get location of the tools
29 VIRSH=$(which virsh)
30 QEMUIMG=$(which qemu-img)
31
32 # hash can be specified here
33 HASHCMD=$(which md5sum)
34 HASHTYPE="md5"
35
36 # set initial location
37 GOHOME=$(pwd)
38
39 # set umask
40 umask 0044
41
42 # check if temp locations exist
43 if [ ! -d "$TEMP_FOLDER" ]; then
44     mkdir $TEMP_FOLDER
45 fi
46
47 # check first parameter, expecting disk|memory
48 if [ $PARM1 == "disk" ]; then
49     MEDIUM="Disk"
```

```
50      EXTRACT_CMD="extract_disk"
51  elif [ $PARM1 == "memory" ]; then
52      MEDIUM="Memory"
53      EXTRACT_CMD="extract_mem"
54  else
55      echo "First parameter should be disk or memory"
56      exit 1
57  fi
58
59  create_folders()
60  {
61      if [ ! -d "$TEMP_FOLDER/$OS_USERNAME/$CURRENTDATE/$INSTANCEID" ]; then
62          mkdir -p $TEMP_FOLDER/$OS_USERNAME/$CURRENTDATE/$INSTANCEID
63      fi
64      OUTPUT=$TEMP_FOLDER/$OS_USERNAME/$CURRENTDATE/$INSTANCEID
65  }
66
67  extract_disk()
68  {
69      # NOTE: this will work only with single qcow2 disk
70
71      # get disk location from virsh xml
72      DISK_LOCATION=$(virsh dumpxml $INSTANCE_NAME |xmllint --xpath 'string(///devices ↩
            /disk[@device="disk"]/source/@file)' -)
73      # get disk type
74      DISK_TYPE=$(virsh dumpxml $INSTANCE_NAME |xmllint --xpath 'string(///devices/ ↩
            disk[@device="disk"]/driver/@type)' -)
75
76      # if variables are empty, create error log instead of image in Swift
77      if [ -z $DISK_LOCATION ]; then
78          echo "create file with error message and store it in swift"
79          exit 1
80      fi
81      if [ -z $DISK_TYPE ]; then
82          echo "create file with error message and store it in swift"
83          exit 1
84      fi
85
86      # if disk image is qcow2 extract it, otherwise create error log
87      if [ $DISK_TYPE == "qcow2" ]; then
88          cd $OUTPUT
89          $QEMUIMG create -f qcow2 $PARM1-$INSTANCEID.qcow2 -o backing_file= ↩
                $DISK_LOCATION &> /dev/null
90          $QEMUIMG convert $PARM1-$INSTANCEID.qcow2 -O raw $PARM1-$INSTANCEID.img &> / ↩
                dev/null
91          rm -f $PARM1-$INSTANCEID.qcow2
92      else
```

```
93          # here wee can decide what to do with different types of images
94          echo "create file with error message and store it in swift"
95          exit 1
96      fi
97  }
98
99  extract_mem()
100 {
101     cd $OUTPUT
102     $VIRSH dump $INSTANCE_NAME $OUTPUT/$PARM1-$INSTANCEID.img --memory-only &> /dev/ ↩
            null
103 }
104
105 create_metadata_file()
106 {
107     cd $OUTPUT
108     echo "Extraction Date: $(date)" > $OUTPUT/$PARM1-$INSTANCEID.metadata
109     echo "Extracted By: $OS_USERNAME" >> $OUTPUT/$PARM1-$INSTANCEID.metadata
110     echo "Image type: $MEDIUM" >> $OUTPUT/$PARM1-$INSTANCEID.metadata
111     #hardcoded cloud provider for now, should be parsed from OpenStack configuration
112     echo "Cloud Provider: Openstack NCI Demo" >> $OUTPUT/$PARM1-$INSTANCEID.metadata
113 }
114
115 create_hash()
116 {
117     cd $OUTPUT
118     HASHRAW=$($HASHCMD $PARM1-$INSTANCEID.img)
119     HASH=$($HASHCMD $PARM1-$INSTANCEID.img|awk '{print $2}')
120     FILENAME=$($HASHCMD $PARM1-$INSTANCEID.img|awk '{print $1}')
121     echo "File Name: $HASH" >> $OUTPUT/$PARM1-$INSTANCEID.metadata
122     echo "File Hash ($HASHTYPE): $FILENAME" >> $OUTPUT/$PARM1-$INSTANCEID.metadata
123 }
124
125 encrypt_image()
126 {
127     # encrypt image with user's password (using openstack password, however it  ↩
            should be command line parameter)
128     cd $OUTPUT
129     gpg --batch --yes --passphrase "$OS_PASSWORD" --cipher-algo AES256 --symmetric  ↩
            $PARM1-$INSTANCEID.img &> /dev/null
130     rm -f $PARM1-$INSTANCEID.img
131 }
132
133 sign_metadata()
134 {
135     cd $OUTPUT
136     TMPGPG=
```

```
137     trap 'rm -rf "$TMPGPG"' EXIT INT TERM HUP
138     TMPGPG=$(mktemp -d)
139     export GNUPGHOME="$TMPGPG"
140     gpg --allow-secret-key-import --import $GOHOME/gpg/gpg_key.private &> /dev/null
141     gpg --yes --batch --passphrase="$GPGPASS" --output $PARM1-$INSTANCEID.sig -- ↩
            detach-sig $PARM1-$INSTANCEID.metadata &> /dev/null
142 }
143
144 upload_container()
145 {
146     cd $TEMP_FOLDER
147     swift \
148         --os-auth-url "$AUTH_URL" \
149         --os-tenant-name "$OS_TENANT_NAME" \
150         --os-username "$OS_USERNAME" \
151         --os-password "$OS_PASSWORD" \
152         upload $CONTAINER $OS_USERNAME/$CURRENTDATE/$INSTANCEID &> /dev/null
153 }
154
155 clean_temp()
156 {
157 # this should be more sophisticated, however it will work fine for my POC
158     cd $GOHOME
159     rm -rf $TEMP_FOLDER/$OS_USERNAME/$CURRENTDATE
160 }
161
162 execute()
163 {
164     create_folders
165     create_metadata_file
166     $EXTRACT_CMD
167     create_hash
168     encrypt_image
169     sign_metadata
170     upload_container
171     clean_temp
172 }
173
174 execute
```

A.1.3 api_client.py

```
1 #!/usr/bin/env python
2
3 # api client for openstack forensic framework
4 #
```

```
5
6  import requests, json
7  import sys, getopt, os
8
9  # simple check if user has access to the tenant using provided credentials
10 # return code must be 200, but there must be better way of doing this:
11 def verify_permission( instanceid, tenant, username, password ):
12     # hardcoded url, will work only on single node openstack
13     url = 'http://127.0.0.1:5000/v2.0/tokens'
14     payload = {'auth': {'tenantName': str(tenant), 'passwordCredentials': {'username ↩
           ': str(username), 'password': str(password)}}}
15     headers = {'content-type': 'application/json'}
16     r = requests.post(url, data=json.dumps(payload), headers=headers)
17     return r.status_code
18
19 # print parameters for testing
20 def test_vars(action, ahost, host, tenant, username, password, container, instanceid ↩
       ):
21     print 'Action   : ', action
22     print 'Host     : ', host
23     print 'Auth-Host : ', ahost
24     print 'Tenant   : ', tenant
25     print 'UserName : ', username
26     print 'Password : ', password
27     print 'Container : ', container
28     print 'InstanceID : ', instanceid
29
30 def usage():
31     script = os.path.basename(__file__)
32     print ''
33     print 'Usage: '+ script+' [options]'
34     print ''
35     print 'Options:'
36     print ' -H HOST,      --host=HOST            hostname or IP of forensic API host ↩
           (required)'
37     print ' -A AUTH_HOST, --auth-host=AUTH_HOST  hostname or IP of Keystone API host ↩
           (required)'
38     print ' -a ACTION,    --action=ACTION        disk or memory (required)'
39     print ' -i INSTANCEID, --instance=INSTANCEID Instance ID (required)'
40     print ' -t TENANTNAME, --tenant=TENANT       Tenant name (required)'
41     print ' -c CONTAINER, --container=CONTAINER  Name of existing Swift container ( ↩
          required)'
42     print ' -u USERNAME,  --username=USERNAME    Openstack username (required)'
43     print ' -p PASSWORD,  --password=PASSWORD    Openstack password (required)'
44     print ' -h,           -help                  Print this help'
45     print ''
46     print 'Examples:'
```

```
47      print ' '+script+' -H http://192.168.1.1:9111 -a disk -t demo -i 12345678-9abc- ↩
            def1-2345-67890abcdef1 -c output -u demo -p demo'
48      print ' '+script+' -H http://192.168.1.1:9111 -a memory -t demo -i 12345678-9abc ↩
            -def1-2345-67890abcdef1 -c output -u demo -p demo'
49      print ''
50
51  # get token and accountid
52  def get_token_accountid( ahost,instanceid, tenant, username, password ):
53      url = ahost+'/tokens'
54      payload = {'auth': {'tenantName': str(tenant), 'passwordCredentials': {'username ↩
            ': str(username), 'password': str(password)}}}
55      headers = {'content-type': 'application/json'}
56      r = requests.post(url, data=json.dumps(payload), headers=headers)
57      if r.status_code == 200:
58          #get json response
59          json_resp = r.json()
60          #get token
61          token = json_resp['access']['token']['id']
62          #get accountID
63          accid = json_resp['access']['token']['tenant']['id']
64          return (token, accid)
65      else:
66          return False, False
67
68  # extract media
69  def extract( ahost, host, action, tenant, username, password, container, instanceid  ↩
        ):
70      token, accid = get_token_accountid( ahost, instanceid, tenant, username,  ↩
            password );
71
72      if token == False:
73          return 'Authentication failed\n'
74      else:
75          url = host+'/'+action
76          payload = {'tenant': str(tenant),\
77                 'username': str(username),\
78                 'password': str(password),\
79                 'container': str(container),\
80                 'accountid': str(accid),\
81                 'token': str(token),\
82                 'instanceid': str(instanceid)}
83
84          headers = {'content-type': 'application/json'}
85          r = requests.post(url, data=json.dumps(payload), headers=headers)
86          return r.text
87
88  def main(argv):
```

```
89      host = os.environ.get('OS_FORENSICS_URL','')
90      ahost = os.environ.get('OS_AUTH_URL','')
91      action = ''
92      tenant = os.environ.get('OS_TENANT_NAME','')
93      username = os.environ.get('OS_USERNAME','')
94      password = os.environ.get('OS_PASSWORD','')
95      container = ''
96      instanceid = ''
97
98      try:
99          opts, args = getopt.getopt(argv,'hH:A:a:t:u:p:c:i:',['help','host=','auth- ↩
                host=','action=','tenant=','username=','password=','container=',' ↩
                instanceid=']) 
100     except getopt.GetoptError:
101         usage()
102         sys.exit(2)
103     for opt, arg in opts:
104         if opt in ('-h', '--help'):
105             usage()
106             sys.exit()
107         elif opt in ('-a', '--action'):
108             action = arg
109         elif opt in ('-A', '--auth-host'):
110             ahost = arg
111         elif opt in ('-H', '--host'):
112             host = arg
113         elif opt in ('-t', '--tenant'):
114             tenant = arg
115         elif opt in ('-u', '--username'):
116             username = arg
117         elif opt in ('-p', '--password'):
118             password = arg
119         elif opt in ('-c', '--container'):
120             container = arg
121         elif opt in ('-i', '--instanceid'):
122             instanceid = arg
123
124
125
126 # check if all parameters were passed
127     errmsg = ''
128     stop = False
129
130     if host == '':
131         errmsg+=' host'
132         stop = True
133
```

```
134     if ahost == '':
135         errmsg+=' ahost'
136         stop = True
137 
138     if action == '':
139         errmsg+=' action'
140         stop = True
141 
142     if tenant == '':
143         errmsg+=' tenant'
144         stop = True
145 
146     if username == '':
147         errmsg+=' username'
148         stop = True
149 
150     if password == '':
151         errmsg+=' password'
152         stop = True
153 
154     if container == '':
155         errmsg+=' container'
156         stop = True
157 
158     if instanceid == '':
159         errmsg+=' instnceID'
160         stop = True
161 
162     if stop == True:
163         print 'Error: missing required options:'+ errmsg
164         usage()
165         sys.exit(2)
166 
167 # execute API calls based on action:
168     if action == 'disk':
169         msg = extract(ahost, host, action, tenant, username, password, container, ↩
                instanceid)
170         print msg
171         sys.exit()
172 
173     elif action == 'memory':
174         msg= extract(ahost, host, action, tenant, username, password, container, ↩
                instanceid)
175         print msg
176         sys.exit()
177 
178     elif action == 'test':
```

```
179         test_vars(action, ahost, host, tenant, username, password, container,  ↩
                instanceid)
180         sys.exit()
181 
182     else:
183         print 'Error: -a, --action must be "disk" or "memory"'
184         usage()
185         sys.exit(2)
186 
187 if __name__ == '__main__':
188     main(sys.argv[1:])
```