SAPIENZA
UNIVERSITÀ DI ROMA

School of Engineering
Department of Computer and System Sciences "Antonio Ruberti"

Master thesis in Computer Science
Academic Year 2010/2011

"Data privacy in Desktop as a Service"

Flavio Bertini

Supervisor: Prof. Roberto Baldoni
First Member: Ing. D. Davide Lamanna

to my parents
ai miei genitori

Contents

Chapter 1

Introduction

1.1 Introduction

Nowadays, privacy has become a touchy issue more then ever, especially if considered in the Cloud Computing[1] environment. People generally think to keep their personal information in a safe place and don't communicate to someone that is not trustworthy. Nevertheless, due to the fact that the Internet offers a lot of possibilities to communicate with other people, we tend to forget this and consequently our privacy is always at risk, especially if we think to the social networks. Furthermore, the necessity to have our personal files available wherever we are, throughout the world, made Cloud Computing something nearly indispensable, and this is the daily bread for systems or, even better, companies like Dropbox[2]. Of course, Dropbox is not the only one supplying for this kind of service, and other famous products have to be mentioned. For instance, Ubuntu One[3] and humyo.com[4] are both online free file storage and backup services, and the privacy issue is present as well. The diffusion of these services caused a lot of people storing documents, images and much more *in the Cloud*. Most of the time no one asks to herself if her data won't be accessed by someone else and this is the common mistake no one should do. We cannot rely on our *online storage service provider* because we will never know if it won't never read our data, even if it declares it won't never do in the service conditions. Cloud Computing is everywhere; just think of Google Mail, where an e-mail service is supplied from Google in conjunction with a complete office suite, in order to be able to open, save or manage all the documents we want. All of these things happen *online*, somewhere in the world, out of our direct control, if compared with our desktop PCs.

The use of Cryptosystems can be adopted in this circumstances to guarantee to the users the

[1] Cloud computing describes computation, software, data access, and storage services that do not require end-user knowledge of the physical location and configuration of the system that delivers the services.

[2] Dropbox is a Web-based file hosting service operated by Dropbox, Inc. that uses cloud computing to enable users to store and share files and folders with others across the Internet using file synchronization.

[3] Ubuntu One is a storage application and service operated by Canonical Ltd. The service enables users to store and sync files online and between computers.

[4] humyo.com is an online file storage service which synchronizes files across multiple computers and a remote data store.

privacy they should have, and this will be one of the main topics of this thesis, taking into account many different attempts for a progressive privacy solution. As it is easy to understand, the main focus is based on *how* to exploit the encryption systems and not simply on the use of cryptography. Despite the topic of this thesis can be considered for any Cloud Computing system, the research is mainly focused on a distributed virtual environment from Trashware[5], called Virtual Distro Dispatcher (VDD) which is described in [3], [4], [13] and introduced in section 1.2. VDD's aim is to project virtual operating systems on terminals (e.g. diskless thin clients) in order to provide fully operating Desktop environments to the users. Since every VDD user has her own virtual operating system, she also has the capability to store personal data in it, and it is possible to identify four data categories that will be stored into the central server: ***User Registration Data (URD)***, ***User Generated Content (UGC)***, ***User Configuration Data (UCD)*** and ***User Log Data (ULD)***[6] as described in [13] and [16].

The main purpose of this research is mainly focused on finding a progressive privacy solution, to protect the four type of data mentioned above[7].

1.2 Virtual Distro Dispatcher

Virtual Distro Dispatcher has evolved since its creation. In order to understand the whole system, and the evolution of VDD, the sections 1.2.1 and 1.2.2 describe what is the basic architecture and the step ahead that has been made in the latest version.

1.2.1 VDD at the beginning

To make its job, VDD needs a fundamental requirement which is the virtualization. The first two versions were very similar, actually the main difference was in the virtualization system used to have a lot of different Linux distributions and Operating Systems in general[8]. In the first version, User Mode Linux[9] has been used, whereas in the last one, Xen[10] was our favorite one. The main reason of this choice is related to the performance issue.

The VDD architecture (see Figure 1.1) is still the same since the first version, and the target to project virtual operating systems on terminals is still there. The main components, if we look at the higher layer as possible are:

[5]The Trashware is a word coming from the contraction of two words: *trash* and *hardware*. It is defined as the practice to reuse obsolete hardware, making new working PCs using pieces coming from diverse computers.

[6]For instance we may think to the Internet browsing footprints protection.

[7]Generally, this type of data are stored in the `/home` directory of every Linux user, like the dotted directories, containing UCD for instance.

[8]It is also important to note that, in the first VDD version we could virtualize only Linux OS whereas in the second one also Microsoft Windows®like Operating Systems are available.

[9]User-mode Linux (UML) enables multiple virtual Linux systems (known as guests) to run as an application within a normal Linux system (known as the host).

[10]Xen is a virtual-machine monitor for IA-32, x86-64, Itanium and ARM architectures. It allows several guest operating systems to execute on the same computer hardware concurrently.

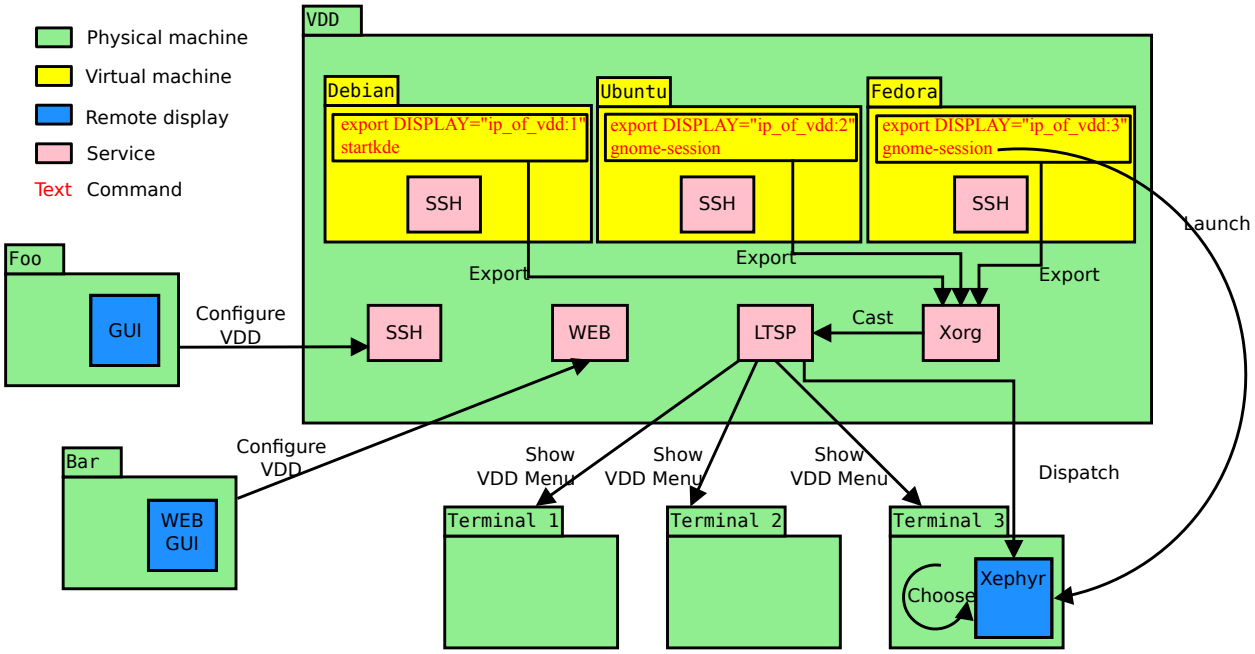

Figure 1.1: Virtual Distro Dispatcher architecture.

- a virtualization system
- a free and open source terminal server for Linux
- a *visualization* system

These three ingredients allow VDD to project the graphical session of a virtualized OS on diskless thin client monitors. In order to make some reference to the Figure 1.1, it is possible to identify in the green area, respectively, Xen which is not explicitly mentioned on purpose, LTSP[11] and Xorg[12]. Other components like SSH, the Web Server Apache and many other services are not part of the VDD core but are certainly useful for many purposes.

1.2.2 The new VDD

The evolution of VDD brought us to concentrate on the Desktop as a Service (DaaS) layer (see Section 2.1), getting rid of the low level virtualization and virtual machine management. Of course, the architecture presented in section 1.2.1 is still intact but the VDD core may vary due to the new

[11]The Linux Terminal Server Project adds thin-client support to Linux servers. LTSP is a flexible, cost effective solution that is empowering schools, businesses, and organizations all over the world to easily install and deploy desktop workstations. A growing number of Linux distributions include LTSP out-of-the-box (http://www.ltsp.org).

[12]The X.Org project provides an open source implementation of the X Window System (http://www.x.org). The research on VDD is also directed in the utilization of other visualization systems like RealVNC (a remote desktop viewer) or the free version TightVNC and also Spice (a complete open source solution for interaction with virtualized desktop devices - http://spice-space.org).

technologies that can be used to reach the same targets in a better way. The change that has been made regards once again the virtualization system: despite Xen is one of the most powerful and efficient virtualization system, it has been chosen to substitute it with Ubuntu Enterprise Cloud (UEC). However, the last sentence is not exactly correct, indeed it is fair to point out that UEC (which is described in details in section 2.5) is not simply a virtualization system but a much more complex (set of) software, which is able to do more than virtualize Operating Systems. Furthermore, Xen has not been definitely abandoned, since UEC supports many virtualization systems including Xen itself, even if it supplies for KVM[13] by default in the standard setup.

Similarly to what has been described in the previous section, VDD is still composed by three main components, and of course LTSP remains along with a remote desktop viewer, with the aim to provide an access to a visualization system. Therefore the three components constituting the core of VDD are now:

- Ubuntu Enterprise Cloud
- Linux Terminal Server Project
- Xorg and/or VNC[14]

The choice of UEC is based on many factors. The most important are: a highly flexible architecture both for the virtual machines management and the Amazon standard API compliance. As a matter of fact, the virtual machines management becomes very easy thanks also to the Mozilla Firefox plug-in called Hybridfox[15] which is one of the targets VDD wants to reach since its birth. Actually, in the previous version of VDD, the action to start a virtual machine was completely *manual* (i.e. the administrator had to open the Linux shell and type a command line instruction) unless an automatic bash script was used, whereas all of those commands are wrapped in the Hybridfox plug-in (in the default case, for KVM) if UEC is used. Another interesting feature is the possibility to make a *cluster* (thanks to Eucalyptus) which is able to support all the workload due to a high number of virtual machines running in UEC. This feature is *enabled* by default, once the private cloud is installed. Furthermore the compatibility with the standard Amazon API allows the development of more sophisticated and personalized software also in order to manage promiscuous environments. The table 1.1 shows the main differences between the VDD previous versions and the last one, but also the progresses that have been made since the first version of VDD. The first column represents the features for VDD to operate at best.

The possibility to have many virtualization systems supported by UEC, gives us also the way to make further performance tests and this is certainly a good point to be considered. Instead,

[13]KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). Further information at http://www.linux-kvm.org.

[14]With the possibility also to use Spice or something similar.

[15]Hybridfox is a Firefox add-on that attempts to get the best of both worlds of popular Cloud Computing environments, Amazon EC2(public) and Eucalyptus(private). In the case in point, we are only dealing with Eucalyptus using UEC.

Features	VDD v1	VDD v2	VDD v3	VDD v4
Virtualization systems support	UML	Xen	Xen	KVM (Xen, vSphere, ESX and ESXi)
HA Clustering		√	√	√
HPC Clustering				√
Web GUI (admin)				√
Web GUI (user)			√	√
GUI (admin)			√	√
GUI (user)			√	√
Open Source	√	√	√	√
Advanced storage management				√
Easy and quick VMs restoring	√	√	√	√
User level kernel execution	√			
Visualization systems	Xorg	Xorg, VNC	Xorg, VNC	Xorg, VNC, NX, Spice
Terminal Server	LTSP	LTSP	LTSP	LTSP
Privacy solution			√	√

Table 1.1: VDD versions comparison.

as regard the clustering, since the research on openMosix prior the VDD's birth[17], we were still looking for a solution to be applied on VDD but unsuccessfully until now[16]. Fortunately, UEC supplies for a complete clustering service, exactly developed for this circumstance. That's just what we were looking for, and it's certainly a great solution to improve the VDD's performances. In VDD version 2 and 3, there wasn't any HPC solution and the only guarantee for the Virtual Machines availability was the utilization of Heartbeat[17] and DRBD[18] to ensure the live migration[19] in case of failure of a host system. The Web Graphical User Interface supplied by Hybridfox is something really useful and also complete, for the UEC management especially as regard the Volumes and Snapshots section. Thanks to this plug-in it is also possible to start and stop any virtual machine as

[16]Unfortunately the openMosix Project has officially closed as of March 1, 2008. The main research target about a clustering solution was to find a HPC clustering system in order to enhance the whole VDD performances.

[17]Heartbeat is a daemon that provides cluster infrastructure (communication and membership) services to its clients. This allows clients to know about the presence (or disappearance!) of peer processes on other machines and to easily exchange messages with them.

[18]DRBD®refers to block devices designed as a building block to form high availability (HA) clusters. This is done by mirroring a whole block device via an assigned network. DRBD can be understood as network based raid-1. Further information at http://www.drbd.org/.

[19]Live migration allows a server administrator to move a running virtual machine or application between different physical machines without disconnecting the client or application.

well as constituting promiscuous environment with Amazon EC2[20] and to restore easily and quickly any virtual machine. As regard the Web GUI at user's side, always thanks to the public API, it is certainly possible to develop a *reduced or modified version* of Hybridfox, with a limited set of features made exactly for the final users. For instance, it would be a good idea to design a graphical interface having few buttons (to start/stop the Virtual Machines), and also the VNC plug-in[21] to use the desktop environment (e.g. Gnome) directly from the web browser. Furthermore, thanks to the Walrus Storage Controller (WS3) and the Elastic Block Storage Controller (EBS) it is possible to store the machine images (MI) that will be instantiated in the cloud (by WS3) and to create persistent block devices that can be mounted on running machines, in order to gain access to virtual hard drives (by EBS). Open Source requirement and the User level kernel execution are very important as well. Actually, as regard these two last features (that are shown in Table 1.1) the Open Source characteristic is a strong property that VDD must have and this is respected from VDD v1 to v4 (actually the VDD project is completely Open Source and GPL[22]), whereas for the User level kernel execution another consideration could be made: in the first version, this feature was a facility because of the fact that a restricted user could be able to start a virtual machine executing a Linux kernel from a command line without any care of the administration right. This limitation has been overcome also thanks to the new infrastructure and software layers introduced by Ubuntu Enterprise Cloud. As a matter of fact, UEC already gives the users the possibility to run a virtual machine through a web GUI, getting rid of the administration right issue (it is sufficient to generate an user account at the UEC side). One firm point of VDD throughout all the versions, is LTSP. The way to dispatch the Linux desktop environments on terminals is always guaranteed by this terminal server and this is used to gain access to the User GUI of VDD central system. Finally, the privacy solution, which is the main topic of this manuscript, is a very important issue that has been considered since the version 3, starting from the disk partitions encryption as described in Section 2.4.

In this manuscript, two approaches are presented and a comparison is made: the first is the utilization of UEC along with the traditional encryption systems (discussed in Chapter 2) and the second one is to use the Homomorphic Encryption (discussed in Chapter 3), highlighting what are the pros and cons for each one. What is important here is that we have to face with a common issue: protecting the privacy of user's data, stored in some place in the cloud. The first method hereby described allows a privacy protection with some limitations with respect to the possibility to deny the root user to read the user's data; in the second method (i.e. using the Homomorphic Encryption) it is possible to overcome this problem but we still have some other limitations.

[20] Amazon Elastic Compute Cloud (Amazon EC2) is a web service that provides resizable compute capacity in the cloud. It is designed to make web-scale computing easier for developers. Further information at http://aws.amazon.com/ec2/.

[21] For instance, gtk-vnc which is a VNC viewer widget for GTK.

[22] The GNU General Public License (GNU GPL or simply GPL) is the most widely used free software license, originally written by Richard Stallman for the GNU project.

1.3 Scope

In this thesis many Homomorphic Cryptosystems are described, in order to evaluate the feasibility of a privacy protection architecture inside VDD, generalizable to Cloud systems at large, in a transparent way. The main target is to build and to study the feasibility of a so called Hedge Proxy[16] to handle encrypted video fluxes and/or homomorphic encrypted data in general, coming from a server.

For those reasons, public key and symmetric Homomorphic Cryptosystems will be considered, for instance Paillier and Modified Rivest's Scheme (MRS)[5] respectively.

The first consideration that will be made is about the utilization of traditional cryptosystems like GnuPG or disk/partition encryption on VDD v.3 showing the pros and cons of this choice. Subsequently, the general idea on how to use Homomorphic Encryption to protect the user's privacy in VDD will be considered; then an evaluation of many different cryptosystems will be done, also considering the performance impact on the central system.

1.4 Thesis organization

The Chapter 2 is an introduction to the privacy issues related to Cloud Computing in general and to the solutions applicable nowadays (like RSA[23], Truecrypt[24], etc.). Furthermore, the issues related to these solutions will be described. Ubuntu Enterprise Cloud will be taken into account in order to show how to protect the privacy using traditional encryption systems.

The Chapter 3 treats the Homomorphic Encryption (HE), describing public and symmetric systems also with respect to the advantages and disadvantages of each one. The main aspect is that we don't need to trust the service provider storing the user data. HE is something really interesting as regard the research for solving this kind of problems at all (also thanks to the fact that it is used to try to create a countermeasure against the root user, with the target to deny the access or the right to see/modify the user data to it too). In other words, not even the system administrator is able to read user data. Normally the user decrypts her data in order to use them remotely and in that very moment the root user is perfectly able to read them. With the HE instead, they can be managed in an encrypted form.

In the Chapter 4 we discuss the future works, and what could be done in order to make all the ideas described in this thesis effective.

[23] In cryptography, RSA (which stands for Rivest, Shamir and Adleman who first publicly described it) is an algorithm for public-key cryptography.

[24] TrueCrypt is a software application used for on-the-fly encryption (OTFE).

Chapter 2

Data privacy in Desktop as a Service

This chapter describes what are the main problems related to what happens or could happen if our data are stored *in the Cloud* and the service provider is not trusted by us. The main countermeasure one should take is, of course, the encryption of personal files, or data in general, but many considerations like performances or usability have to be made.

2.1 Introduction to Cloud Computing

During the last years, a new idea has born: the Cloud Computing. Before the coming of this new idea, the *Grid Computing* was the main infrastructure idea which consists to make a combination of computer resources to reach a common target. The Cloud Computing got the upper hand on the Grid for many reasons. For instance, the Grid Computing is distributed computation oriented and the application running on the Grid should be designed in a specific way to reach a unique target; furthermore, in this circumstance all the resources are going to be used unconditionally, drifting away from the idea to supply for a real service. As regard the Cloud Computing instead, the resources are distributed as well as heterogeneous, and the general idea is to supply for services for the customers. The idea of Cloud Computing may be realized in many ways, so for this reason there is not a precise scheme representing it. As a matter of fact, if one thinks to the Cloud Computing, can imagine the Internet where a lot of technologies communicate between themselves (see Figure 2.1). Hardware and software resources are made available on the Internet, and can be used remotely. This means that if we are dealing with a Cloud Computing application, it doesn't necessary run on our PC but it may be running somewhere in the Cloud, using the system resources given by third parties. In general, a Cloud Computing Service Provider, owns a lot of powerful Servers due to the necessity to instantiate, a huge quantity of virtual machines in order to supply for any service the customer wants. The use of virtualization is very wide in Cloud Computing and for this reason a lot of computational power is needed; actually a customer can have the necessity to rent a system with specific hardware and software requirements which can be very high and this can be accomplished by vendors only having a lot of hardware and software resources. This necessity can be better understood if one thinks to the fact that each Service Provider may have a

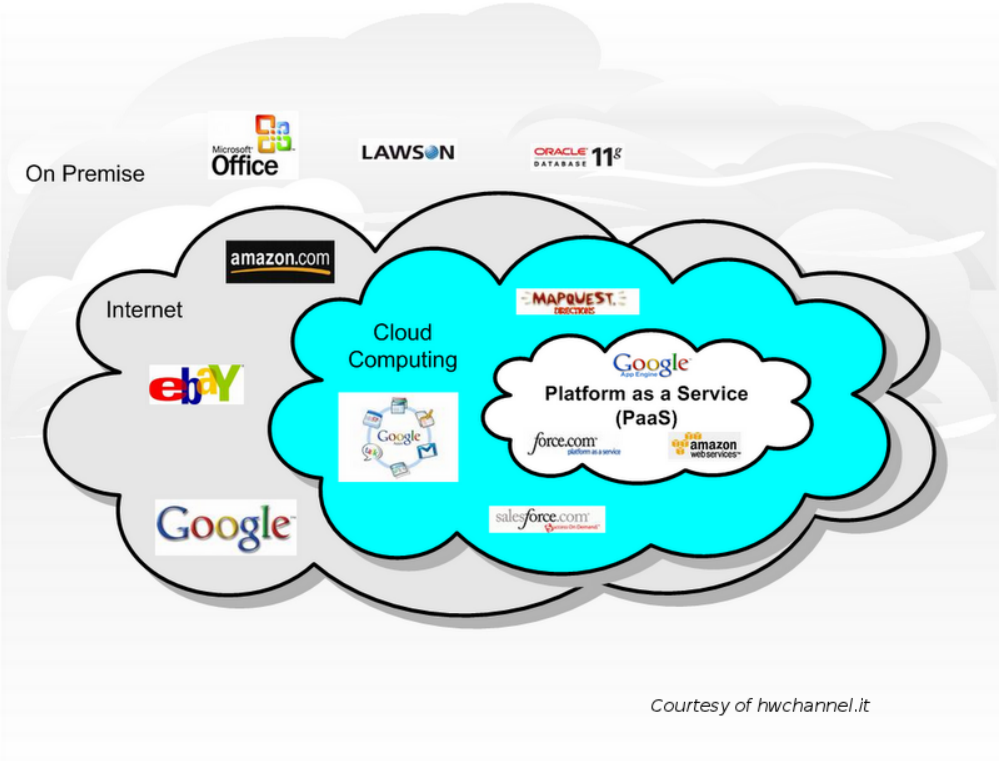

Figure 2.1: The Cloud Computing idea.

massive number of customers.

The name Cloud Computing comes from the fact that, the Cloud represents the Internet and the computation happens somewhere in the Cloud. For this reason one can exploit the resources placed in some place in the network.

The Cloud Computing is represented by three models[13]:

- *Software as a Service* (SaaS): it consists in the remote software utilization, most of the time through a web server.
- *Platform as a Service* (PaaS): the Service Provider supplies for an entire software platform, which can be composed by many applications, services, etc. The customer can also create her own applications and deploy to the cloud.
- *Infrastructure as a Service* (IaaS): this consists in a remote hardware resources usage, where the resources are on demand for the customer. Actually, as the customer needs a certain amount of computational power, the required number of resources are allocated.

In all of those models (see Figure 2.2) we have to face with the privacy issue, just for the main reason that our personal data can potentially be read by the service provider. On the other hand, the advantage to use a Service in the Cloud, allows the customers not to incur into expenses for buying new hardware to execute their own computations and to exploit other computational resources

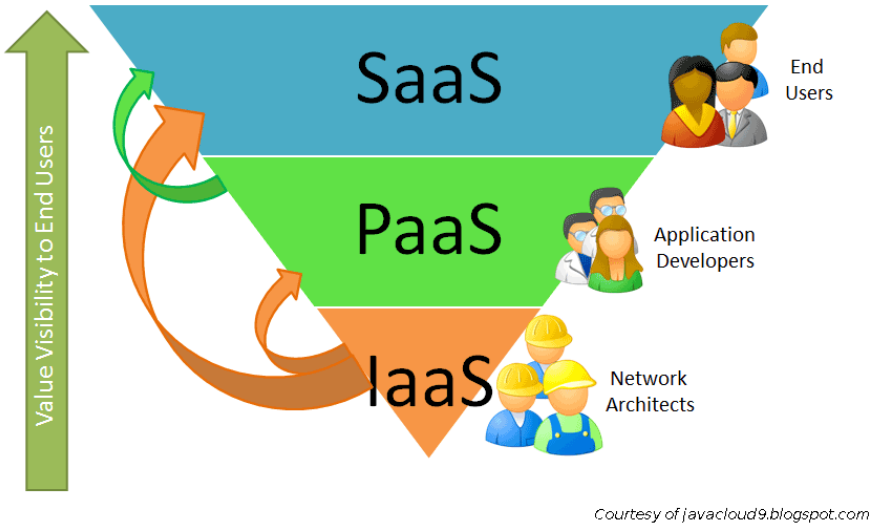

Figure 2.2: The Cloud Computing model stack.

supplied by the Cloud Service Provider. Of course, this allows to save money for the customers but, other issues like the privacy must be considered. Since this manuscript is focused on *data privacy in Desktop as a Service*, we need to place DaaS somewhere in the Cloud Computing model stack represented in Figure 2.2: of course it has to be placed on the top of the stack, but the right position could be at the same level of SaaS (see Figure 2.3). Actually, Desktop as a Service, is a (set of) software, which relies on the underlying layers and could be something more complex than a so called *software* because of the fact that it is composed by many applications constituting a *Desktop* environment for the users. Another reason for the choice to place DaaS and SaaS on the same layer is that a platform like VDD can supply its *resources* to the users in two ways: generic software (e.g. a web application or a shell script) and a complete Desktop environment.

Another advantage of the Cloud Computing utilization, comes from the fact that one can store her personal data in a remote storage. This certainly rises the well known privacy issue, which is widely discussed here, but let's think for a moment to the fact that our data are always available, even in case of a failure of our PC. Taken for granted that every Server exposed to the Internet is always subject to attacks (but this is another aspect related to security which may cause also a privacy violation), and that they should be fought using firewalls and many other techniques, we have to face another issue: the outsourcer (i.e. the Service Provider) owning the data in her local servers, is potentially able to analyze or use them at least inside her system, even she declares our data are encrypted and/or password protected as the service policies states. Another problem could be represented by the risk of lost of customer's data at the Service Provider side. Of course, this is a problem that must be solved by the outsourcer, replicating the servers (potentially) over

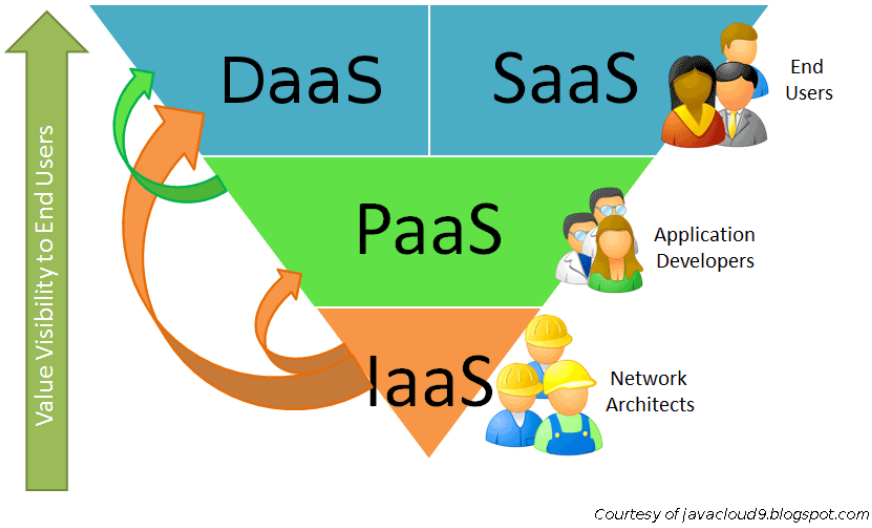

Figure 2.3: The Cloud Computing model stack with DaaS.

the world. This means that our data are spread throughout the world and we never know where they really are; to face this problem, a first and easy answer could be *data encryption*.

2.2 Personal data encryption

The most easy answer to the privacy issue in Cloud Computing is *Encryption*. Unfortunately, it is not so simple as it appears because since we also need to perform some operations onto a file, it is not possible if it remains encrypted, generally speaking. In Chapter 3, we will see that it is possible to perform operations on encrypted data and how this can be used to accomplish our necessities.

Just to focus on what is the problem, think of a remote storage server (see Figure 2.4), which is provided by someone that is untrusted by us. The main problem here, is that when a file is

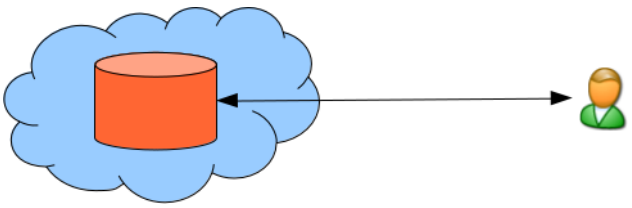

Figure 2.4: Interaction between the user and the remote storage.

decrypted at the Cloud side, it could also be visible to whatever people have the direct access to the storage. Many storage service providers declare they don't read your data, but we don't trust

them anyway. As an example, suppose you own a file containing some personal information on your computer and you want to make it available wherever you are simply staying connected to the Internet. What you can do is to make a subscription to an online storage service provider, encrypt your file on your computer and put it in your *private* online folder. If you need to perform a simple operation, such as open the file, you have to decrypt first. If the decryption is performed in your online *private* folder, the file could be accessed by someone that is untrusted or malicious and can violate your privacy especially if she has physical access to the storage as in the case of the service provider itself. Actually, the solution could be to copy the file into your PC, decrypt and then read it, but this becomes something not performed in the optimal way and requires several operations which is absolutely not efficient and not comfortable too, especially if the file is accessed from a public place, like an Internet point. As a matter of fact, if one accesses to her online storage has to perform the following operations: copy the file (suppose it is a LibreOffice[1] document) on the local computer, decrypt, open it with the word processor *lowriter*, make some modifications, save, close and encrypt again taking care to safely delete the plain copy[2] of the file. Finally, update the encrypted online copy with the most recent one. As it is easy to understand, these operations are unpractical as we will see in Chapter 3 if not executed through a local application, which should then be downloaded and installed. In any case, performance disruption is going to be quite high. A solution could be the possibility to directly manipulate encrypted data.

In the case in point (i.e. VDD), the problem is quite similar but a little bit different. At the moment, VDD's architecture (see Figure 2.5) works in a Local Area Network and the Server is, or could be, present in the same location of the clients. The main difference is between the public and the private Cloud. In order to better understand an example is made: let's consider Dropbox to be our Private Cloud and VDD the Public one. In the Dropbox case, nobody knows the architecture or the subsystem constituting it and for this reason is considered private. In the VDD case instead, everyone can know how it is composed and how it works even going in deep on the software components which make VDD working. What it happens in VDD is something very similar to the situation described for Dropbox with the difference that both the user and the Storage Service Provider are in the same network even if the possibility to dispatch virtual operating systems over the Internet is now more concrete. In the following paragraphs the possible solutions to the privacy issue in VDD are described and what are the problems introduced.

2.3 Traditional cryptosystems

The public key cryptosystems are widely used for private communications by e-mail. These systems may also be used to encrypt files and this is the reason why it is noteworthy in this circumstance. Since every VDD user has her own virtual Operating System and her own filesystem, she can store

[1] LibreOffice is a free software office suite developed by The Document Foundation that is compatible with other major office suites and available on a variety of platforms.

[2] One should use some application which electronically shred the files, overwriting the hard disk many times in the location where the file was saved.

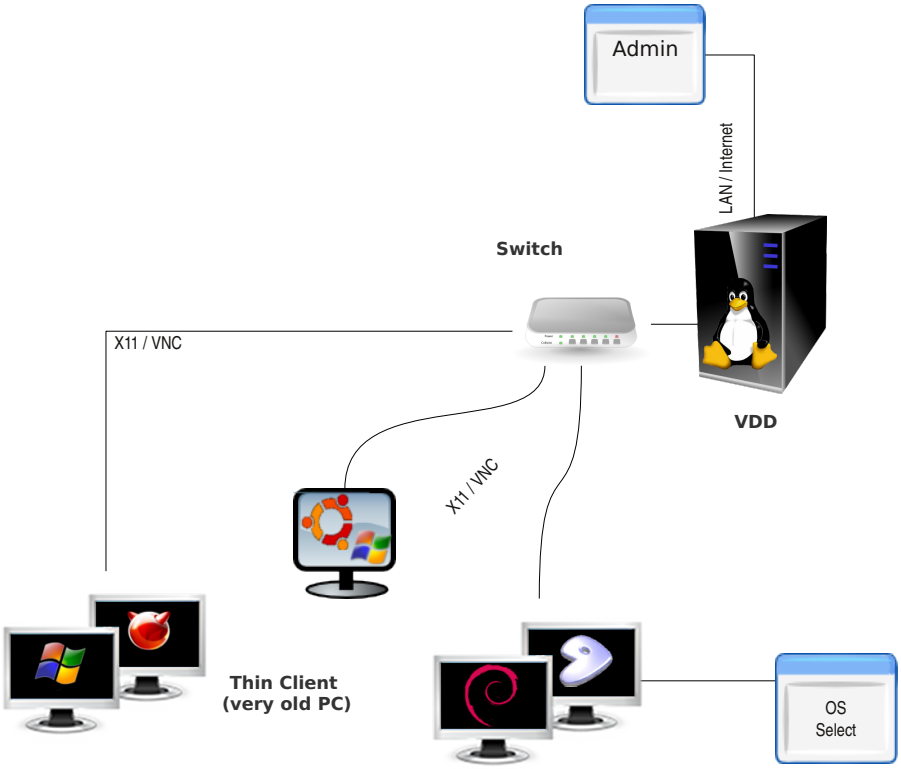

Figure 2.5: VDD General Scheme

files in it once they have been encrypted. This could be a valid solution but the problem is that, once the file is decrypted, it is accessible also from the VDD administrator, unless the user uses her own external USB storage or the thin client has a local hard disk. In any case, this approach is still unusable in VDD, due also to the fact that it is not transparent enough to the users[3] even though GnuPG is installed in every virtual machine, for instance. Furthermore, a very old thin client could not have the USB ports (the reader should keep in mind that most of the thin clients we use come from Trashware).

A similar assertion may also be done as regard a symmetric key cryptosystem. In this case we also have the duty of sharing a key between two parties, both for encryption and the decryption.

2.4 Disk partitions encryption solutions

There are many software solutions available in the Internet as regard the disk partitions encryption. Probably the most common is TrueCrypt[2], whose intent is to encrypt entire disks or partitions.

[3]The users shouldn't care about manually encrypt or decrypt files using software like GnuPG.

The approach hereby adopted is to assign for each user a LVM[4] disk partition in the host system (i.e. the VDD server) and to put a Linux filesystem/distribution in it. In general, the encryption of the entire `root` filesystem is not recommended and most of the time it is usual to make a separated `/home` partition to be encrypted mainly for a pure performance issue. This happens onto normal workstation and it should happen all the more so on VDD as well. As described in [1], each Linux virtual machine doesn't use a file as a root filesystem but it uses a disk partition, that is encrypted using dmcrypt-LUKS[5] system. Once the logical volume has been created and checked with badblocks[6], it is encrypted using cryptsetup, a tool to setup encrypted devices with `dm-crypt`. In order to make this system working, every encrypted volume is mounted on the host and shared via Samba on any VMs, then the users can access their own volume using their credentials from inside the virtual machines. Of course, every time a Linux virtual machine starts, the partition is decrypted and the user can read or write her own files. This solution seemed the most intuitive for VDD, since each user has her own virtual disk for the personal data storage and the partitions can be resized very quickly. This could be a safe approach if once the disk content has been decrypted, it wouldn't be readable by the VDD system administrator too. Actually, as for the RSA example, all the data that before the decryption were unreadable, become readable both from the user and the system administrator[7]. Of course, this happens even in case of the utilization of filesystem images instead of LVM partitions, because of the fact that the *root* user can mount that images as loop files somewhere in the host filesystem and read the content.

In any case, both solutions (shown in paragraphs 2.3 and 2.4) are still not sufficient for our purposes, since the need to guarantee the privacy for URD, UGC, UCD and ULD (mentioned in Section 1.1) is also needed. In Chapter 3, an alternate encryption system, the Homomorphic Encryption, will be described in order to study how it could be possible to use it in VDD to solve the problems described above.

2.5 Ubuntu Enterprise Cloud + Tomb solution

Ubuntu Enterprise Cloud[20] is a Linux distribution based on Ubuntu Server and Eucalyptus, an open source implementation of the EC2 Amazon API, whose aim is to build a completely free and Open Source private cloud. The corresponding enterprise edition is Amazon EC2 and UEC has all

[4]LVM is a logical volume manager for the Linux kernel; it manages disk drives and similar mass-storage devices, in particular large ones.

[5]`dm-crypt` is a transparent disk encryption subsystem in Linux kernel versions 2.6 and later and in DragonFly BSD. It is part of the device mapper infrastructure, and uses cryptographic routines from the kernel's Crypto API. Linux Unified Key Setup or LUKS instead, is a disk-encryption specification created by Clemens Fruhwirth and originally intended for Linux.

[6]Search a device for bad blocks.

[7]This solution has been adopted by BinarioEtico, a cooperative company that provides services related to free software

Type	RAM	DISK	Intel VT-x/ AMD-v	Network Interfaces
Cloud Controller	512 MB	100 GB		1 Gigabit Ethernet
Cluster Controller	512 MB	100 GB		2 Gigabit Ethernet
Node Controller	2 GB	20 GB	√	1 Gigabit Ethernet

Table 2.1: UEC minimal hardware requirements.

the capabilities to interact with it, in order to have a promiscuous cloud infrastructure. UEC gives the possibilities to instantiate our private Cloud as powerful as possible depending on the hardware resources we dispose. Of course it also has the capability to provide *on demand* computing to the users, as well as Amazon Web Services does with its customers. The setup procedure is quite easy: it is sufficient to use the Ubuntu Server setup CD and an automatic installation procedure starts. The minimal hardware requirements to allow UEC working properly are shown in Table 2.1.

In order to setup our private cloud with UEC to be used with VDD, two identical workstations have been used[8]. The Figure 2.6 shows what are the components installed on each workstation.

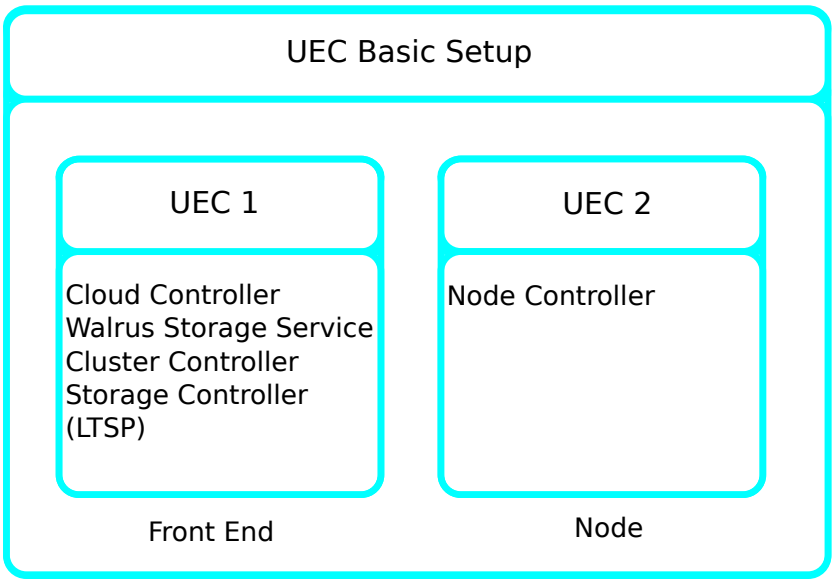

Figure 2.6: The UEC basic setup.

Each workstation has got the following characteristics:

- CPU: Intel core i7 920 @ 2.67GHz 8MB SK1366 RET;
- RAM: 8GB (4 DIMM KINGSTON 2GB PC1333 DDR III CL9);

[8]At least two physical machines are needed to have UEC working.

- 3 Western Digital hard disks, RAPTOR 74GB 10000rpm 16MB SATA;
- BOX COOLER MASTER ELITE 330 BLACK;
- Fan Zalman 7500;
- VD CAPTIVA GEFORCE 9400GT 1GB PCX;
- TECHSOLO 730W;
- Supplementary gigabit network card.

Of course it is also possible to install the Cloud Controller and the Cluster Controller in two separate machines in order to have a better workload distribution and ensure a higher business continuity. With the aim to make a reference on what has been stated in Section 2.1 and to the Figure 2.3, Ubuntu Enterprise Cloud is positioned at the bottom of the stack, having the role of Infrastructure as a Service. Actually, the main UEC's role, is to supply for a complete virtualization and clustering platform for VDD, which is a DaaS and relies on top of the Cloud model stack. Furthermore, it's important to remember that UEC gives the possibility to supply for (virtual) hardware on demand, depending on the user's need. For instance, if a user needs a Desktop Environment with 2 CPUs, 2GB of RAM and a 10GB hard disk, the UEC infrastructure will be able to supply for a virtual machine having all of those requirements, of course always depending on the host server characteristics.

2.5.1 UEC architecture

The Eucalyptus (which is an acronym for "Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems") architecture which UEC is based on, consists in five main and modular components:

- Cloud Controller (CLC)
- Walrus Storage Controller (WS3)
- Elastic Block Storage Controller (EBS)
- Cluster Controller (CC)
- Node Controller (NC)

Every component is an independent web service and it has its own web interface for the system administrator management. The Figure 2.7 shows how these components are deployed in a typical Eucalyptus architecture. In the case of VDD, and referring also to the Figure 2.6, the picture showing the Eucalyptus architecture (Figure 2.7) highlights how the components have been installed in the two servers described in Section 2.5. The three upper services reside on UEC1, whereas the nodes (in the case in point there is only one node) on UEC2.

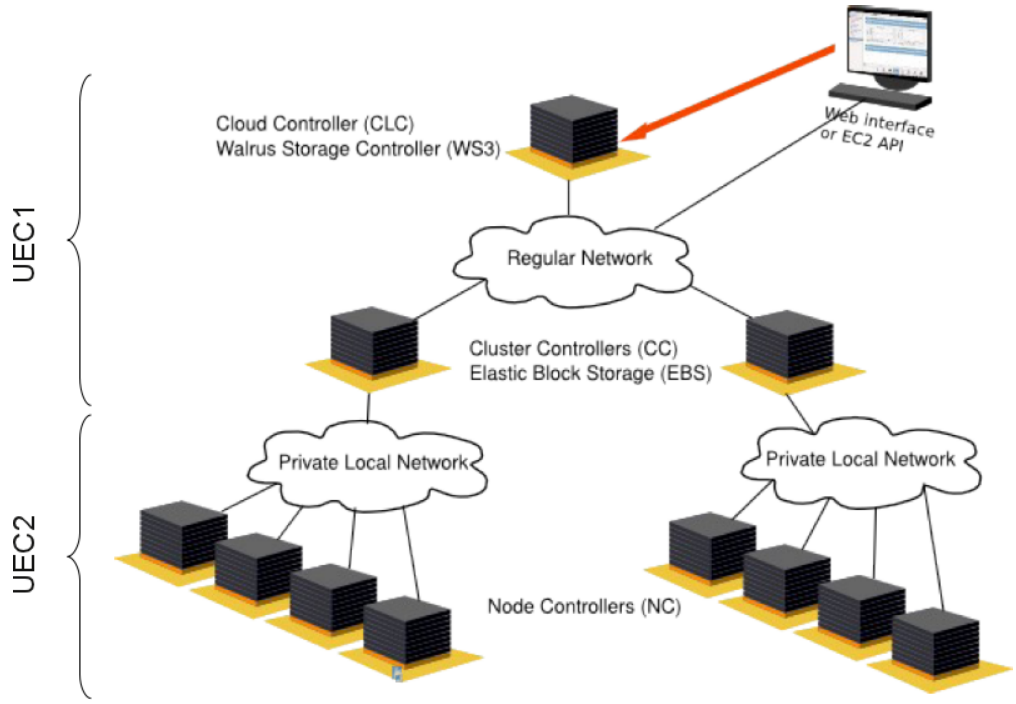

Figure 2.7: The Eucalyptus architecture.

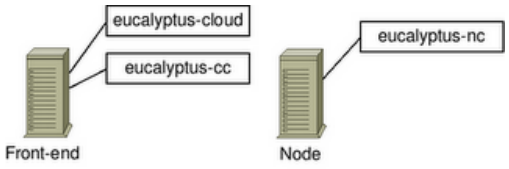

Figure 2.8: Basic UEC setup.

Eucalyptus is an open source implementation of the EC2 Amazon API and it has been realized at the University of Santa Barbara - California and now maintained by Eucalyptus Systems, the company founded by the authors; the Eucalyptus task is to build a completely free and Open Source private cloud using clusters or typical workstations. One of the strong points is the compatibility with EC2 at the API level, since it gives a lot of new possibilities to the service providers but also to normal users. As a matter of fact, the service providers can rely on a professional solution to provide a cloud computing environment which is very extensible due to this characteristic. As regard the other users, they may consider to develop their own applications on UEC in a *private* laboratory in order to put them in Amazon EC2 at the end once they would need more power. Eucalyptus has been developed using standard Linux tools and basic Web Services technologies. The main features can be listed below in few points:

- the Cloud configuration is made through a web interface

- the Web Service and query interface is compatible with EC2
- the internal communication is made safe thanks to SOAP and WS-Security[9]
- it is possible to configure multiple clusters, each one having an IP address, on a single cloud
- the cluster controller may have personalized policies as regard the workload (Round Robin or Greedy scheduling techniques may be used)
- it is possible to quickly build hybrid clouds
- enables the state of the art of the enterprise cloud features: high availability, storage integration, monitoring and auditing
- storage integration (iSCSI, SAN, NAS)

The following sections describe more in deep the components of UEC.

Cloud Controller

The Cloud Controller (CLC)[20] stays at the top level of the UEC architecture and it is the component which is exposed to the Cloud Administrator more than others. Actually, it supplies a coupling point between the whole cloud and who needs to manage the virtual machine instances, the network, the storage and the cloud in general but also the system load monitoring. The cloud can be managed thanks to the Hybridfox Firefox plug-in (or Elasticfox for Amazon EC2) or via SSH. The CLC also provides the standard SOAP[10] based API (compliant with the Amazon EC2 API). There is a direct contact between the Cloud Controller and the Cluster Controller(s), in order to make the allocation of new instances possible.

Walrus Storage Controller

The component which is responsible to provide the storage management, the virtual machine images and user data access control is called Walrus Storage Controller (WS3)[20]. WS3 implements the REST[11] and SOAP API. Of course these API are compatible with Amazon Simple Storage Protocol (S3) but the main role of this component is described in two points. Actually WS3 allows to:

- store the Machine Images (MI) that will be used for the virtual machines
- access and store data

[9] WS-Security (Web Services Security, short WSS) is a flexible and feature-rich extension to SOAP to apply security to web services.

[10] SOAP, originally defined as Simple Object Access Protocol, is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks.

[11] Representational State Transfer (REST) is a style of software architecture for distributed hypermedia systems such as the World Wide Web.

Another important characteristic of WS3 consists in supplying for a high level storage management. The user doesn't care of how and where her files are stored. It's not important, at this level, how the file will be stored at lower levels at the bottom of the whole architecture. The user knows it is reliable and consistent. WS3 is reliable even in case of concurrency, whereas a write operation is performed by many subjects in the cloud.

Elastic Block Storage Controller

The Elastic Block Storage Controller (EBS) [20] allows to dynamically attach (mount) block devices to the Virtual Machines so that the users inside those machines can access to the virtual hard disk and use them to put an entire filesystem inside or other data in general, for instance. EBS is certainly very connected to WS3 and they work in cooperation since it is possible to manage volumes and create filesystem snapshots (as an example, this can be accomplished using Hybridfox at the front end side) which are stored just in the WS3. The snapshots are used to instantiate more volumes from the same starting point or to keep data safe in case of filesystem corruption. The respective of EBS in UEC in the Enterprise version is called Amazon Elastic Block Storage.

Cluster Controller

As already pointed out, between the Node Controller and the Cloud Controller stands the Cluster Controller (CC)[20], because it needs access to both NC and CLC. Actually, the CLC sends the requests to allocate the MIs and the CC has the duty to choice which Node Controller will run the Machine Instance, thanks to the status reports sent by the Node Controllers. CC also manages the virtual network built among the virtual machines and routes the traffic in order to make the communication between the various instances possible and reliable.

Node Controller

The Node Controllers (NC) are at the bottom of the UEC architecture[20]; the more the NCs are available, the more the whole architecture is powerful. The NC controls, as an hypervisor and under the CC requests, the Virtual Machines activities, in particular their execution and termination. One of the most important NC's roles is to decide where to run the VMs on the basis of the available hardware resources (i.e. the disk space, number of CPUs, amount of RAM, etc.). This is a very important issue, also because of the fact it contributes to the workload balancing in the whole system.

2.5.2 Tomb in UEC: an alternative to LUKS

Even adopting the solution described in this section, we have to face with the problem already mentioned in Section 2.4. Actually the concrete is that it is impossible to avoid the administrator reading the user's data: the system administrator *can do whatever she wants* on the system and this is a clear rule on an Operating System. The main problem is that, an encrypted partition, or

a Tomb[12] file (described later on) has to be mounted somewhere in the filesystem in order to be written or read. Once that partition has been mounted, the root user is able to read or modify the data inside it and the privacy is rapidly violated. Another unconvincing solution could be to not influence the filesystem; in other words, if one keeps her data mounted in memory (RAM), the administrator wouldn't find anything in the root filesystem, which is related to that data. The reason why this solution is not so privacy preserving is because an expert system administrator could perform a memory dump meanwhile the user mounts her private data in the RAM, and obtain the (previously) encrypted data as well. One of the possibilities to reduce the risk of a memory dump by the system administrator could be to disable the `/dev/kmem` and the `/dev/mem` devices disabling the `CONFIG_DEVKMEM` kernel module, but this is not definitive, since the root user can always enable this feature again and overcome the limitation. Furthermore, this could be a solution for self installed VDD systems (we trust ourselves), but not for a cloud environment in general, since we don't have any possibility to modify the systems running an Online Storage Provider.

Once we assumed the conditions above, we know what are the limitations of this choice. The reason why we could use Tomb as an alternative to LUKS is because of its easiness to be used and it is specific for Desktop use. Actually, an user can decide to have her own encrypted *folder* or not by herself. Both in the VDD v3 and v4, the user may exploit the encryption feature or not (for example she can decide to encrypt the `/home` partition), making a decision without depending on the administrator and the advantages are described later on. She decides how much disk space has to be encrypted and where to place her file.

Tomb works in a similar way of dm-crypt and LUKS with respect to the user's point of view. It consists in an Open Source system which allows to store and backup personal data into an encrypted storage file (one or more) which can be opened and closed using a keyfile and protected also thanks to the use of a password chosen by the user in the setup phase. The setup is very easy, and the script `tomb-open` may be used to start the whole procedure. Once the script has started, the user is asked to confirm the operation, to choice the name of the tomb file, and the dimension of the file itself. The setup terminates with the choice of the password to protect the tomb file and it is also possible to store the keyfile inside a USB external storage keeping it separated from the tomb file for better security. Every tomb file works like a directory which can be safely moved wherever the user wants in the filesystem and can also be hidden. Opening a tomb file is very easy: it is sufficient to click on the file or using the command tomb-open as said before. Once a tomb is open the application panel shows a skull icon which allows to perform many operations on the tomb itself. One of the interesting features of Tomb is that it makes use of steganography, allowing to store a key inside an image, for instance. This allows the users to keep their keys in a safe place without any serious risk to be victim of a key robbery. The Tomb target community are those users who make use of a desktop environment. Since we are dealing with a Desktop as a Service solution

[12]Tomb is a simple tool to manage encrypted storage on GNU/Linux, from the hashes of the dyne:bolic nesting mechanism. Further information at http://tomb.dyne.org/.

(i.e. VDD), Tomb is something tailor-made for us since it has been developed for Desktop users.

The idea on how to use Tomb in VDD, using UEC is pretty easy. For the sake of simplicity, suppose to have one virtual machine for each VDD user; the intent is to install Tomb on every virtual machine and give the possibility to use it to the users. One of the main advantages of this choice is that the administrator workload decreases due to the fact that she doesn't have to care about the disks partition encryption anymore. The freedom for the users to exploit the availability of a cryptosystem like Tomb is very important as well and this makes things easy to manage more than ever.

2.6 UEC in VDD: what's new?

The decision to use UEC in VDD certainly improved the whole system and the virtual machine management is made easy now, as well as for the virtual disks and many other features. The solution to use Tomb in UEC also brought some improvement with respect to the previous solution (i.e. the disk partition encryption) but the privacy issue is not completely solved even if the VDD's enhancement brought us a lot of facilities.

With the aim to summarize and clarify what are the advantages of the introduction of UEC let's consider the following two points:

- the adoption of DaaS allows us to concentrate on other aspect (like privacy) and all the services are much more easy to be managed
- The EC2 compliance of UEC guarantees a great interoperability between VDD and Amazon EC2 itself

It is clear enough that UEC gives us a lot of facilities in this sense and allows the VDD project to be improved getting rid of the burden of the virtualization configuration, but also of many other aspects. The Chapter 3 introduces a possible solution to a complete privacy protection including preservation against untrusted Service Providers. Pros and cons of that solution will be also considered.

Chapter 3

Homomorphic Encryption

On the basis of what stated so far and in particular in Chapter 2, it is very difficult (nearly impossible) to protect the user's data privacy also against the service provider's privileges unless the data itself remains encrypted. The idea discussed in this chapter is just to never decrypt the user's data and to perform any operation on encrypted data. At first impact, this may seem something impossible or quite odd, but the type of encryption treated in this chapter leaves us the hope that something can be done for the privacy protection. The challenge now is: since the root user can do everything, how can I prevent the administrator to read my private data? One answer is to never give plain documents (or data in general) to the administrator, neither having them plain on the filesystem, nor on the RAM. This is possible using the Homomorphic Encryption but let's see what it takes.

The Homomorphic Encryption is a very powerful encryption scheme, that makes it possible to perform some arithmetical operation on encrypted data. The main reason why the Homomorphic Encryption is considered in this topic is because the solution to the problem already discussed in Chapters 1 and 2, could be to definitely never decrypt the personal information (e.g. files) stored in the VDD server. When we say, *never decrypt*, we intend that the personal data will never be decrypted on the remote storage in VDD or *in the Cloud*, in general. Of course, there should be something between the user and the client terminal allowing the personal data to be viewed only by their respective owners. The Homomorphic Encryption has being studied since over 30 years, and it is certainly something not really new in the literature, but nowadays the interest in this encryption scheme grown a lot, due to the great influence of the Cloud Computing, which is present more than ever in our life, and to the privacy issues related to it.

Just to introduce some mathematical concept, that will be used in this chapter, we can define an Homomorphic Encryption algorithm *E(m)*, where m is the plain-text, such that $E(m) = c$. As a consequence, given $E(m_1)$ and $E(m_2)$, is it possible to obtain $E(m_1 \circ m_2)$ in the encrypted domain, for some operation $\circ$ [19]. Actually, the Homomorphism is defined as it follows:

Definition 3.1 (Homomorphism)**.** Let $\mathbb{A}$ and $\mathbb{B}$ be two sets with a binary operation each, respectively $\circ$ and $\diamond$, we say that $\varphi : \mathbb{A} \to \mathbb{B}$ is a *homomorphism* if the following relation holds:

$$\varphi(a \circ b) = \varphi(a) \diamond \varphi(b) \qquad \forall a, b \in \mathbb{A} \tag{3.1}$$

This property, applied to Cryptography, will be used for the progressive privacy solution hereby presented and discussed with the specific target to manipulate encrypted objects in the cloud.

This chapter presents many different Homomorphic Cryptosystems, giving a particular relevance to the difference between the Partially and the Fully Homomorphic Encryption.

3.1 Partially Homomorphic Cryptosystems

A Partially Homomorphic cryptosystem is a cryptosystem where few arithmetic operations are supported. Typically we are dealing with addition, multiplication or exponentiation. The following sections describe the main partially Homomorphic Cryptosystems and show what are the Homomorphic properties for each one. It must also be said that a partially Homomorphic cryptosystem can be considered as the basic ingredient for the fully Homomorphic Encryption discussed in Section 3.2.

3.1.1 RSA

RSA[1] is the most popular public key algorithm[12]. Given a public key e modulus n and a message m, the encryption is given by $E(m) = m^e \mod n$. The Homomorphic property is defined as it follows:

$$E(m_1) \cdot E(m_2) = m_1^e m_2^e \mod n = (m_1 m_2)^e \mod n = E(m_1 \cdot m_2) \tag{3.2}$$

As the equation 3.2 shows, we have a unique Homomorphic property, which is multiplicative. For the purposes of the research hereby discussed, only one Homomorphic property is not sufficient because of the fact that it is not possible to solve all the operations we would need to perform to preserve the privacy, using only the multiplication. As it will be discussed in Section 3.2, at least two Homomorphic properties are necessary.

3.1.2 ElGamal

The ElGamal[12] scheme is very well known as regard the signature of messages, but it is also known for the encryption mechanism. Given a group G, a public key (G,q,g,h), a secret key x where $h = g^x$, the encryption of a message m is: $E(m) = (g^r, m \cdot h^r)$ for some $r \in \{0, ..., q-1\}$. The Homomorphic property is:

$$E(m_1) \cdot E(m_2) = (g^{r_1}, m_1 \cdot h^{r_1})(g^{r_2}, m_2 \cdot h^{r_2}) = (g^{r_1+r_2}, (m_1 \cdot m_2)h^{r_1+r_2}) = E(m_1 \cdot m_2) \tag{3.3}$$

Also in this case we only have the multiplicative Homomorphic property and, it is not enough for the reasons already mentioned in Section 3.1.1.

[1] Rivest, Shamir and Adleman

3.1.3 Goldwasser-Micali

The Goldwasser-Micali (GM) cryptosystem is a step ahead with respect to the target this research is focused on. Actually, as the equation 3.4 shows, we have two operations. Given the public key x modulus n (x,N), it is possible to encrypt a bit b as it follows: $E(b) = r^2 x^b \mod n$ and the Homomorphic property of GM is:

$$E(b_1) \cdot E(b_2) = r_1^2 x^{b_1} r_2^2 x^{b_2} = (r_1 r_2)^2 x^{b_1+b_2} = E(b_1 \oplus {}^2 b_2) \tag{3.4}$$

The equation 3.4 shows that we are dealing with the multiplication and the exclusive OR. This begins to be very interesting, since it demonstrates that it is possible to perform an operation in the encrypted domain which is different with respect to the plain-text domain as shown in the Definition 3.1. In any case this is not yet enough, because our necessity is to perform operations on quantities that are not simply bits, but something more complex like a file for instance.

3.1.4 Benaloh

The Benaloh cryptosystem is an extension of the Goldwasser-Micali cryptosystem, where the bits are no longer treated. Indeed, in this scheme we can encrypt directly blocks at once[3]. Given:

- a blocksize r
- two large primes p and q such that r divides $(p-1)$ and $\gcd(q-1, r) = 1$
- $n = pq$
- $y \in (\mathbb{Z}/n\mathbb{Z})^*$ [4] such that $y^{(p-1)(q-1)/r} \not\equiv 1 \mod n$

The public key is (y, n) and the two primes p and q constitute the private key. So, to encrypt a message m is necessary to choose a random $u \in (\mathbb{Z}/n\mathbb{Z})^*$ and $E_r(m) = y^m u^r \mod n$. Said that, the Homomorphic property is:

$$E(m_1) \cdot E(m_2) = (y^{m_1} u_1^r)(y^{m_2} u_2^r) = y^{m_1+m_2}(u_1 u_2)^r = E(m_1 + m_2 \mod r) \tag{3.5}$$

As represented in the equation 3.5, the Homomorphic property is stronger respect to the previous we have seen in the equations 3.2, 3.3 and in 3.4. Actually, having the two operations, addition and multiplication available, is it possible to think also to something more complex as regard the operations feasible on entities like a file.

3.1.5 Paillier

The Paillier Cryptosystem[5] is another public key cryptosystem having many different implementations in Java but also in C programming language. Paillier has got these requirements:

[2] Exclusive OR

[3] Further information at `http://en.wikipedia.org/wiki/Benaloh_cryptosystem`

[4] Also represented as Z_n^*.

[5] Further information at `http://en.wikipedia.org/wiki/Paillier_cryptosystem`

- two large random primes p and q such that $\gcd(pq, (p-1)(q-1)) = 1$
- $n = pq$ and $\lambda = \text{lcm}(p-1, q-1)$
- a random integer g where $g \in \mathbb{Z}^*_{n^2}$
- n divides the order of g and the following modular multiplicative inverse must exist:
  $\mu = (L(g^\lambda \mod n^2))^{-1} \mod n$
  The function L is defined as $L(u) = \frac{u-1}{n}$
- the public key (n, g)
- the private key (λ, μ)

This encryption scheme is rather similar to the Benaloh discussed in Section 3.1.4. The public key is modulus n and has base g. To encrypt a message $m \in \mathbb{Z}_n$ we select a random $r \in \mathbb{Z}^*_n$ and compute $E(m) = g^m \cdot r^n \mod n^2$. The Homomorphic property is the following:

$$E(m_1) \cdot E(m_2) = (g^{m_1} r_1^n)(g^{m_2} r_2^n) = g^{m_1+m_2}(r_1 r_2)^n = E(m_1 + m_2 \mod n) \tag{3.6}$$

From the equation 3.6, it is possible to delineate a set of homomorphisms both for addition and multiplication of plaintexts:

- **Addition**

$$D(E(m_1, r_1) \cdot E(m_2, r_2) \mod n^2) = m_1 + m_2 \mod n \tag{3.7}$$

  which means that the decryption of the product of two ciphertexts, gives the sum of the two plaintexts.

$$D(E(m_1, r_1) \cdot g^{m_2} \mod n^2) = m_1 + m_2 \mod n \tag{3.8}$$

  where the decryption of the product between a ciphertext and a plaintext raising g is equivalent to the sum of those plaintexts m_1 and m_2.

- **Multiplication**

$$\begin{aligned} D(E(m_1, r_1)^{m_2} \mod n^2) &= m_1 m_2 \mod n \\ D(E(m_2, r_2)^{m_1} \mod n^2) &= m_1 m_2 \mod n \end{aligned} \tag{3.9}$$

  these two equations are very similar and it is easy to understand that raising an encrypted plaintext m_1 to the power of another plaintext m_2 (first case) will result in the product of those two plaintexts after the decryption.

$$D(E(m_1, r_1)^k \mod n^2) = k m_1 \mod n \tag{3.10}$$

  In general, and similarly to the previous case, raising an encrypted plaintext m_1 to the power of a constant k, will result in the product of the plaintext m_1 and the constant k after the decryption.

The Paillier Cryptosystem is very interesting for the properties mentioned above, and are certainly significant as a starting point for the Fully Homomorphic Encryption discussed in Section 3.2. Actually, we can think that, given the two operations $+$ and $*$ it is possible to realize other more complex operations even using logical circuits, if we intend to build a specific hardware to implement some specific function for our purposes.

3.1.6 Modified Rivest's Scheme (MRS)

In this section, another Homomorphic encryption scheme is going to be described, giving a particular attention to the fact that it is a symmetric key scheme. The main reason why this scheme is considered is because of the performances. Actually, as stated in [5], public key techniques such as RSA or Paillier are computationally inefficient.

Original Rivest's Scheme

The Original Rivest's Scheme[5][18] is hereby described:

- Select two large primes p and q where $n = pq$. Now p and q constitute the secret key and n is publicly known.
- The encryption is defined as it follows: $E_{(p,q)}(m) = (m \mod p, m \mod q)$
- The decryption is performed reducing the two components $\mod p$ and $\mod q$, then the Chinese Remainder Theorem has to be applied

As regard the homomorphism, we have both addition and multiplication. Given $E_{p,q}(m_1) = (x_1, y_1)$ and $E_{p,q}(m_2) = (x_2, y_2)$ the two Homomorphic properties are the following:

- **Addition**:
  $E_{p,q}(m_1 + m_2) = Add(E_{p,q}(m_1), E_{p,q}(m_2)) = (x_1 + x_2 \mod n,\ y_1 + y_2 \mod n)$
- **Multiplication**:
  $E_{p,q}(m_1 m_2) = Multi(E_{p,q}(m_1), E_{p,q}(m_2)) = (x_1 x_2 \mod n,\ y_1 y_2 \mod n)$

As well as for the RSA scheme discussed in Section 3.1.1, to break the algorithm is necessary to factorize the two large primes p and q, but the difference here is to have both addition and multiplication supported. The only problem is that the Original Rivest's Scheme is subject to a ciphertext-only attack[5]. Actually, given $E(m)$, we can use $Multi$ to find $E(m^2)$ and recursively exploit Add until the result is equal right to $E(m^2)$. This cause an attacker to find m, which corresponds to the number of rounds we applied Add. For this reason the next section describes the Modified Rivest's Scheme in detail with the purpose of giving a stronger encryption scheme not leaking this security issue.

Modified Rivest's Scheme Details

The first step of the algorithm is like the original scheme. The secret key is defined as $K = (p, q, r_1, \ldots, r_i, \ldots, r_l, s_1, \ldots, s_i, \ldots, s_l)$ where $1 \leq i \leq l$. The encryption function has got the following form: $E_K(\cdot) : \mathbb{Z}_n \to (\mathbb{Z}_n \times \mathbb{Z}_n)^l$. The MRS works as it follows:

Encryption:

A message $m \in \mathbb{Z}_n$ is encrypted thanks to the following steps:

- The message m is divided into l arbitrary numbers $(m_1, m_2, \ldots, m_l)$ such that $m = \sum_{i=1}^{l} m_i \mod n$
- Two secrets numbers $r_i < p$ and $s_i < q$ are randomly chosen $\forall i \in [1, l]$
- The encryption is performed as it follows:

$$\begin{aligned} E_K(m) = ((m_1 r_1 \mod p, m_1 s_1 \mod q), \\ (m_2 r_2 \mod p, m_2 s_2 \mod q), \ldots \\ \ldots, (m_l r_l \mod p, m_l s_l \mod q)) \\ = ((x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l)) \end{aligned}$$

  So, the encryption consists in the encryption of all the single *pieces*, constituting the original message.

As regard the decryption, given a ciphertext $C = ((x_1, y_1), (x_2, y_2), \ldots, (x_l, y_l))$ it decrypts as it follows:

- First, all components are multiplied with their corresponding $r_i^{-1} \mod p$ and $s_i^{-1} \mod q$:

$$\begin{aligned} ((x_1 r_1^{-1} \mod p, y_1 s_1^{-1} \mod q), \\ (x_2 r_2^{-1} \mod p, y_2 s_2^{-1} \mod q), \\ \ldots, ((x_l r_l^{-1} \mod p, y_l s_l^{-1} \mod q)) \end{aligned}$$

- The Chinese Remainder Theorem is used to find $m_1, m_2, \ldots, m_l(\mod n)$.
- Finally all m_i are summed altogether to find m.

Let's now analyze the Homomorphism in MRS and see that it is slightly different from the previous one. Given $E_K(a) = ((x_1, y_1), \ldots, (x_l, y_l))$ and $E_K(b) = ((u_1, v_1), \ldots, (u_l, v_l))$ and a constant $t \in \mathbb{Z}_n$ we have:

Addition:

$$E_K(a + b) = (((x_1 + u_1), (y_1 + v_1)), \ldots, (x_l + u_l), (y_l + v_l))) \mod n$$

Scalar multiplication:

$$E_K(ta) = ((tx_1, ty_1), \ldots, (tx_l, ty_l)) \mod n$$

It is easy to understand now, that the difference from the original Rivest's scheme, stands in the *scalar* multiplication. Due to this fact, the MRS is not subject to the ciphertext-only attack mentioned in the previous section.

The security aspect is certainly very important, but we are still looking for a compromise between security and feasibility, always as regard our main target: *privacy homomorphism*. In fact, in the next Section the Paillier C library will be taken into account and used for the performance tests. This is because no other C libraries have been found for download at the time of this research.

3.2 Fully Homomorphic Cryptosystems

The Homomorphic Encryption can be a valid solution to perform some operation in the encrypted domain without the need to decrypt the operands. As stated in Section 3.1, we have only the possibility to perform the addition or the multiplication. This limitation implies that it is impossible to build more complex functions in the encrypted domain unless some *logic* architecture is being implemented on top. Indeed, Craig Gentry, as described in [8] and in [10], has given us the definition of *fully Homomorphic Encryption*, describing how to reach the target to perform *any* operation on encrypted data only using the logic AND/OR gates. As it is easy to understand, this can have a great impact in the outsourcing[7][14] and in the Cloud Computing, because of the fact that, hopefully, each operation performed on the data stored in a remote server in the Cloud, are totally protected from a privacy violation attempt. The Homomorphic Encryption has been studied for a long time (about 30 years) and it is currently being studied to solve the privacy issue in the Cloud. During this time, the possibility to build a fully Homomorphic cryptosystem was uncertain until Craig Gentry in 2009, shown the first fully Homomorphic Encryption scheme, using *lattice based cryptography*[9]. The idea of the fully Homomorphic Encryption is based on the fact that it would be possible to perform *any* function, or complex operation, evaluating arbitrary depth circuits; actually, starting from operations like *addition* and *multiplication*, it is possible to derive much more complex operations, combining the operators between them.

As an example let's consider the following simple function y:

$$y = 5x^3 + 4x^2 + 3x + 2 \tag{3.11}$$

using the Horner algorithm, the function 3.11 may be written as it follows:

$$y = 2 + x(3 + x(4 + 5x)) \tag{3.12}$$

It is now easy to represent the function 3.12 as a logical blocks scheme (see Figure 3.1), having at our disposal only the addition and multiplication modules.

Of course, in case of more complicated functions, it is necessary to resort to more complex circuits, but this is just an example to see that it is possible to solve a generic function. A similar example could be represented by an image scaling: let x be the image and y the resulting image after a scaling three times bigger than the original. The function could be easily represented as

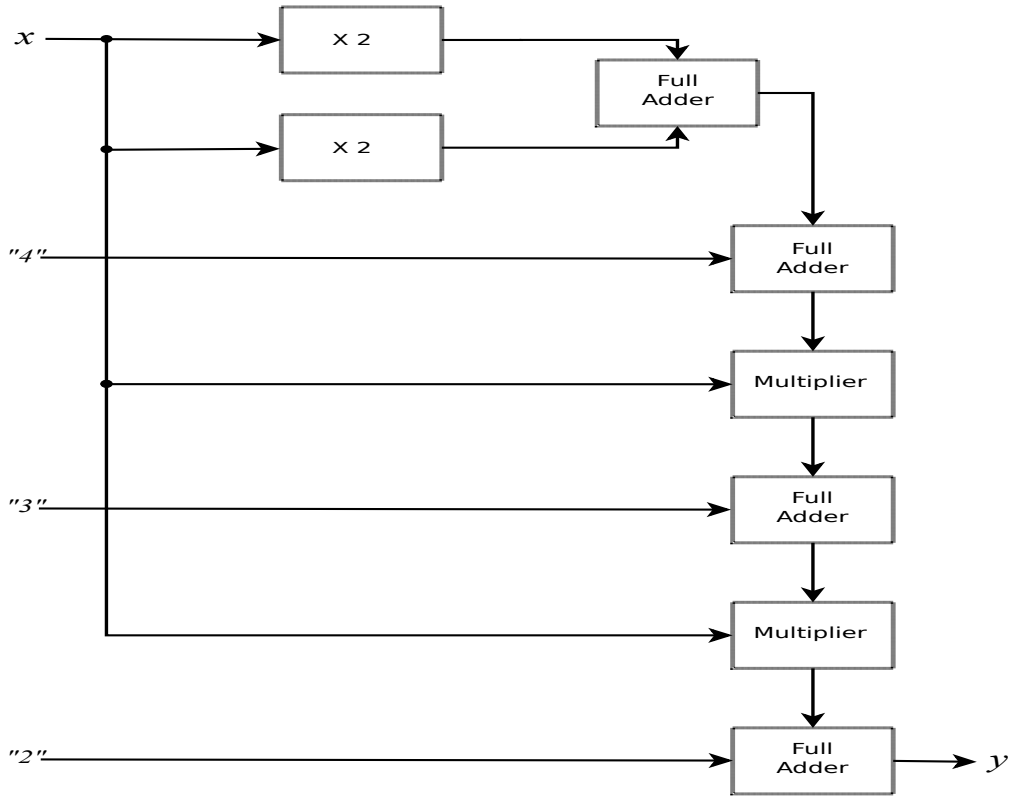

Figure 3.1: The logical blocks scheme which solves the function 3.11.

$y = 3x$. As in the previous example, the function could be written again as $y = x + 2x$ and the logical circuit is pretty easy to realize. The reason why the Horner rule is used is because we already have integrated circuits solving the multiplication by 2. All these modules can certainly be implemented both as software and hardware, depending on the performance requirements and on other factors. What is important here is the possibility to solve any function using the basic mathematical operations.

This is the idea we would like to exploit in VDD, with the purpose to protect user's data and operations, hiding also to the *root* user what she is really doing in the VDD server from the thin client. The Gentry's research started from a *somewhat Homomorphic* Encryption scheme, which proposed to evaluate only low degree polynomials over encrypted data. The reason of this limitation is due to the noise produced by each ciphertext (Gentry says a ciphertext is noisy), which grows every time an operation (a multiplication or an addition) is performed on two ciphertexts. A denoising procedure is also mentioned in [11], where it is possible to understand what is the impact produced on the whole process.

3.2.1 Alice's Jewelry

To give the idea of what the act of performing operations in the encrypted domain means, or even better in a protected environment, Gentry, proposes a Jewelry Store example in [10]. In order to

make a direct comparison between the example hereby discussed and the target we want to reach in VDD, it's sufficient to think that every operation - such as a file manipulation in general - should be performed in such way that *the* (Linux) *box* is not aware of what the user does. This assertion will be more clear as soon as the reader will go ahead. The main concept is that the VDD central server can work on user's data, keeping them always private, without the necessity to know the private key of the encrypted data it is manipulating (i.e. without the necessity to have them decrypted in order to perform whatever function on them); and this is always true even if the function applied on the data is very complex[6].

Gentry, with his Jewelry example, wants to show that data manipulation or processing without having access to it, is not as absurd as it seems. Actually, think of a jewelry store owned by Alice, where a lot of precious material need to be maintained by the salesman. Alice doesn't trust the salesman and she wants that the salesman operates on a diamond, for instance, without giving him the access to it. What could the idea be? Well, the idea is to put the diamond inside a transparent box, giving the possibility to manipulate it only using special gloves. The private key is the physical key to open the box, owned by Alice, and the encrypted data is the diamond. The astute reader could certainly make a remark: *I can see the diamond, it is not really encrypted*! Of course, the analogy is not really perfect for the comparison with the environment considered in VDD, but it makes sense to give the idea of manipulation of a precious object (generally speaking) without giving a direct access to it. Getting back to the point, the plan consists in the following: Alice uses a transparent box which is impenetrable and it has two gloves attached (the only way to touch the box content is to use these gloves). Once the diamond is in the box, Alice closes and give it to the worker. Once the box is in the salesman's hands, he can do whatever he wants with the diamond (i.e. assemble a ring). Once the work has finished, the salesman give back the box to Alice with the manipulated piece. As we can see, the piece comes finished without the need to have direct access to it, by the salesman.

What represents the homomorphism here are just the gloves; actually they allow to perform some operations in the encrypted (secured in the box) domain. Instead, the operation to complete the work represents the function $f(m_1, \ldots, m_t)$, a function we would like to perform Homomorphically using encryption.

The Alice's jewelry is just a metaphor to have an idea of the protection level we would like to have in a system like VDD, but of course, in the Homomorphic encryption circumstance we have to consider a lot of other problems. Furthermore, making a deeper association between this example and the real world in cryptography may induce in some conceptual mistakes as we have warned.

3.2.2 Towards the fully Homomorphic encryption

Given an encryption scheme ε, we have four algorithms: $KeyGen_\varepsilon$, $Encrypt_\varepsilon$, $Decrypt_\varepsilon$ and $Evaluate_\varepsilon$. The first three algorithms are easy to understand and intuitive. Let's focus on the

[6]The reader should keep in mind the noise factor introduced by each chiphertext and the corresponding denoising procedure also discussed in [11]

last one: $Evaluate_{\varepsilon}$. This algorithm has to be intended associated with a set of functions $\mathcal{F}_{\varepsilon}$ which are permitted on the encrypted data. Any function $f \in \mathcal{F}_{\varepsilon}$ performs an operation on the encrypted data C like if it was performed on the plaintext. Actually, after the decryption of the ciphertext C, the result is the same like if f was directly applied to the plaintext[7].

If the scheme ε can handle all functions, then it is *fully Homomorphic*. In order to make an analogy with the Alice's jewelry described in Section 3.2.1, if she had a fully Homomorphic scheme, the salesman could operate on the diamond, performing any manipulation he wants that is ordered by Alice, who will extract the piece from the box at the end once the *function* has finished. Nothing else can be done on the piece out of Alice's control.

One of the most important problem in building a fully Homomorphic cryptosystem is the computational complexity. This is a very important issue also in VDD, since the users cannot wait for a very long time an operation on a file to be terminated. Of course, the complexity, directly depends on the function f. The more f is complex, the more the $Evaluate_{\varepsilon}$ algorithm would be complex. As already mentioned in Section 3.1.5, the $Evaluate_{\varepsilon}$ algorithm is realized using a *Boolean circuit*, using AND, OR and NOT gates, but also NAND instead. The complexity of the circuit is computed considering the size S_f of the circuit itself, which calculates f. Said that, an $Evaluate_{\varepsilon}$ algorithm is efficient if a polynomial function g exists such that for any function f which consists in a logical circuit of size S_f in practice, the algorithm has a complexity of at most $S_f \cdot g(\lambda)$.

As it is easy to understand, the circuit dimension can increase very quickly due to the function complexity. This can have a great impact on a system like VDD where a simple function (with respect to the user point of view) like a picture rotation, could require a very wide logical Boolean circuit. Even worse if this consideration is made for all the functions we would like to supply in a complete desktop environment.

3.3 HE on VDD

Since the Homomorphic Encryption is a very powerful and interesting system, and since we may note that the premises to realize a system preserving the privacy of user's data with HE exist, it is important now to see how this cryptosystem may be incorporated in VDD.

The VDD privacy protection scheme may be summarized in two points:

- user's data protection
- visualization system video stream protection

The solution that implements this protection scheme is called Hedge Proxy[13][16]. The Figure 3.2 shows a scenario where the user is able to perform some operations from a thin client. The main difference with respect to the Tomb solution presented in Section 2.5.2, is that in this case there is not the necessity to decrypt the data to perform an operation with it, in addition to the

[7] We are talking about the *logical* concept of a function f.

fact that it is a completely different approach to the problem. The advantage to never need to decrypt a file is something really innovative in VDD as well as in Cloud Computing.

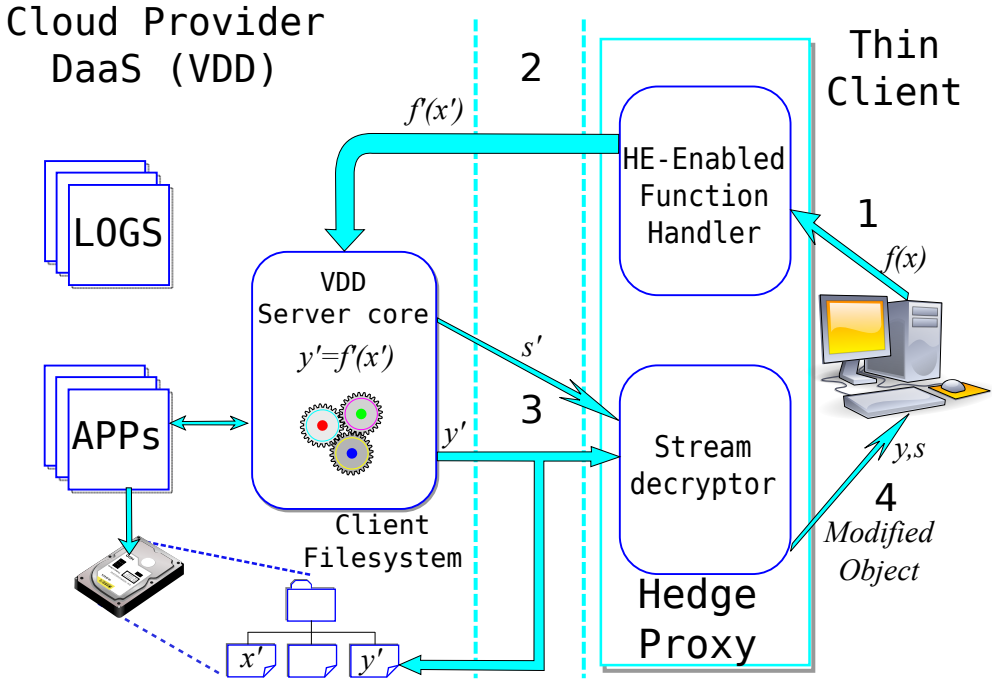

Figure 3.2: The Hedge Proxy solution.

Definition 3.2. Let f be a function in the plaintext domain and x a plaintext object, then f' is the respective of the function f in the encrypted domain and x' is the encrypted object.

For the sake of simplicity, suppose the user is working onto an image (i.e. doing photo editing), and she has the necessity to perform a very banal operation on it. For instance, the operation the user could do on the image, can be to draw a circle somewhere. The operation *draw a circle*, may be represented as a function f to be applied on the image object x, having the target to hide what is the image modification to the administrator. This is reasonably possible because of the fact we start from a fully Homomorphic Encryption scheme (see Section 3.2) where it is possible to perform *any* operation on an encrypted object. Having established this, it would be certainly possible to compute whatever function we want (rotation, scaling, drawing a circle, etc.) on a certain input, in this case the image. It must also be noticed that the image file is always encrypted, therefore the administrator can't know it is an image. Our purpose is to exploit the Homomorphic Encryption to apply the function f on the image x in the encrypted domain. With reference to the Figure 3.2, it is possible to highlight the following steps:

1. the user, with her favorite image manipulation program, modifies the image drawing a circle in a random place. This means performing a certain operation/function on an object as it usually happens.

The action to draw a circle on the image, in other words the function $f(x)$, is sent to the (Homomorphic Encryption Enabled) *Function Handler* which is correctly interpreting and translating in another function $f'(x')$ on the encrypted file thanks to the Homomorphic properties.

2. the function handler, forwards the $f'(x')$ to the VDD Server core, that will execute the $f'(x')$, giving y' as the output and making the modification persistent on the filesystem.

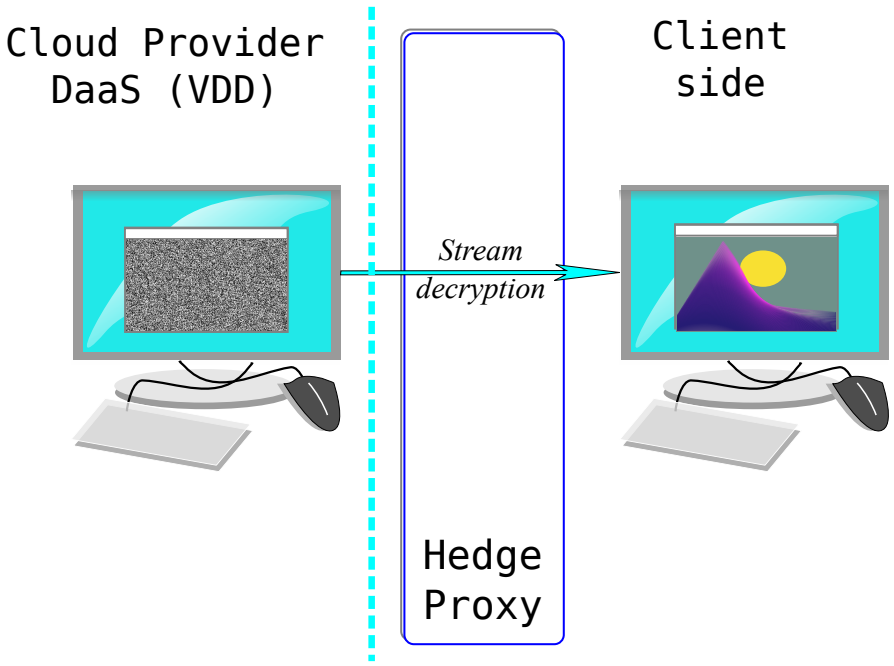

Figure 3.3: The video stream is decrypted frame by frame.

At this point the operating system produces two results: y' and the video stream s' [8], which is encrypted and derived from y' itself. We have to imagine a process which is internal to the operating system in which, after some complex passages, it is possible to obtain the graphical representation of y' (this is what a visualization system actually does). y' will then be decrypted and projected on the thin client screen at the end of the process. To create this encrypted stream, the operating system (and then the CPU) will have to perform many operations (additions, multiplications, shifts, etc.). Thanks to the HE and Fully HE, it is possible to perform those operations in full privacy. In this way it will be generated a video stream s' representing the graphical result of y'. Later, it will be decrypted.

3. the VDD Server core sends the result y' and the video stream s' to the *Stream Decryptor* on the Hedge Proxy; the data y' is also sent to the VDD's filesystem for the persistent storage

[8] As VDD is based on a visualization system, the video stream is what is encrypted on the thin client

where also the old object (image) x' is located (which is the previous version of y'). The Stream Decryptor will decrypt both (see Figure 3.3 and Figure 3.4) producing y and s. The result y can also be sent to a Thin Client USB external memory just in case of explicit request to download and save the result of the operation by the user, while the video stream s will be sent to the Thin Client screen for the final visualization.

The Stream Decryptor receives y' and s', both obtained from y'. The video stream comes encrypted and the Hedge Proxy has the task to decrypt the stream on the fly, before showing the content to the user. The actual purpose of Figure 3.3 is just describing the following scenario: before the application frame (i.e. the window) is delivered to the client, it is like if it was scrambled at the server side. As the figure represents, it is all *plaintext* but the application window, which will be decrypted only on the thin client screen. The main target of this solution is to protect all the data coming from the server and directed to the thin client. In Section 1.2.1, the Xorg windows system has been mentioned, but also Spice must be considered in order to implement the Hedge Proxy. Of course, it is strictly necessary to modify the source code of the visualization system in order to intercept every frame it is transmitted by an endpoint to another. In order to study the feasibility of this solution an analysis of the medium frame size has been made. The Section 3.4.1 describes the whole study in details.

4. Once the Stream Decryptor has received the modified image and converted to plaintext, the Hedge Proxy sends it to the thin client that will show the image itself (or the application window) as plaintext (see Figure 3.3).

The Figure 3.4 shows more in detail what happens between the VDD Server and the Thin Client in terms of data and video fluxes. In this figure two cases are represented: on the top, the case in which no privacy protection is required; on the bottom the Homomorphic Encryption protection is working. On the right part of the figure, two possibilities of final user interaction are depicted:

a) The thin client user watches the stream s on the screen

b) The thin client user downloads a content from the server to her USB memory stick

Starting from the left of the figure for the case one, the function f is performed by the Operations Executor and a result y is produced. The Enhanced Visualization System (EVS) also produces a video stream $\hat{s}$, which consists in the whole stream except the UGC (see Section 1.1) which is represented as scrambled in Figure 3.3 on the left side. The result y passes through the Hedge Proxy unconditionally up to the USB external storage, while the final video stream has to be composed using the $\hat{s}$ and the function result y:

$$s = \hat{s} + y \tag{3.13}$$

As regard the bottom part of Figure 3.4, something similar to the previous case happens. In this case, which is directly connected to what described for the Figure 3.2, the Homomorphic

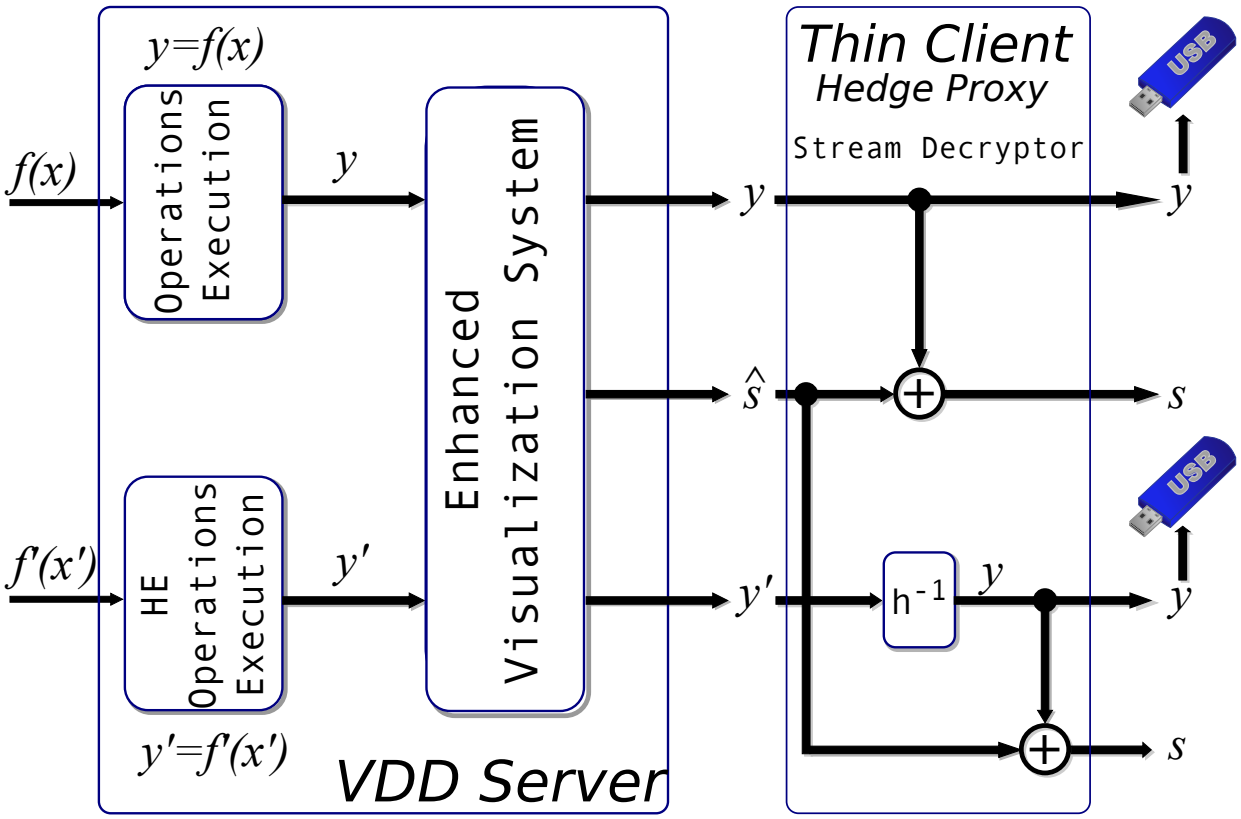

Figure 3.4: User's data and Video stream fluxes.

Encryption Enabled Operations Executor produces an encrypted result y' which is passed to the EVS. A further feature of EVS is the possibility to store a reference to the exact point of the frame where y' has to be projected (painted) on the Thin Client Screen in terms of coordinates. The EVS produces y' and $\hat{s}$ that are sent to the Hedge Proxy. In this case, to produce the stream s a further operation is required before: the Homomorphic decryption of the result y'. Once the result has been decrypted, the situation is the same represented in the relation 3.13.

This experimental solution allows the users to never *stand up and be counted* since their data wont never be read from the root user. Of course, the question that the sharp reader will be doing in this moment would be: *how can a generic desktop software deal with encrypted data?* Actually this is the point of the research, or even better the target that should be reached, as it will be discussed later on in this text. In other words, the challenge now is how to obtain a *HE-Enabled* application (with reference to the example above, the image manipulation program should be HE-Enabled, for instance).

Of course, the image manipulation example is trivial, but this is just to show what are the principles of this scheme. As a matter of fact, if the scheme presented above is applicable to a trivial situation like the one hereby described, it would be certainly possible to project some more

complex architecture to make the final users able to perform more difficult operations.

3.4 Performance issues

The solution discussed in Section 3.3, raised two issues: one is related to the system performance impact when decrypting a video stream at the client side; the other is related to the computational cost due to the operations performed on data. The first issue has to be estimated first computing the dimension of the single frame that is continuously displayed and refreshed at client side and this is taken into account in Section 3.4.1. The successive sections will go in deep in both issues and will show what are the performance test results.

3.4.1 The frame dimension

Since we need to decrypt, at the client side of course, a complete and continuous video stream, how does this workload impact on the system? In order to estimate the workload at the Hedge Proxy we need to know how much time is required to decrypt a single frame. The purpose of this Section is to find a mean value of the frame dimension and to compute how much time is necessary to decrypt it. It must also be considered that the workload depends on the video frequency at which the frames are displayed on the terminals but also on the frame dimension and this is why we need to know a mean value. The idea is to use the Hedge Proxy to continuously decrypt every frame it is able to intercept (we put ourselves in the worst case, i.e. no losses are admitted). We have to put a *performance meter* at the client side, since we are interested on what the final users perceive also in terms of usability. Let's consider to have a source (the VDD server) which is able to send to the clients an encrypted video stream. At client side, the Hedge Proxy needs to decrypt every frame before it is painted on the screen. How much this operation costs?

In order to estimate the cost of a frame decryption, an estimate on what is the medium frame size delivered to every client has to be done. The experiment needs the following requirements:

- two workstations[9] with VNC installed (respectively a VNC server and a VNC client)
- a video stream (it is necessary to ensure that the stream to the clients is roughly the same for each test)
- a packet sniffer at the client side

The idea is to reproduce the same video sequences (see Figure 3.5) at the server side and to visualize it at client side, through VNC. The network protocol analyzer (i.e. the packet sniffer) that has been used is Wireshark[10]. Of course the experiment has been conduced at many different resolutions, sometime even too high on the purpose in a general VDD context. Once the frame

[9]For the purpose two virtual machine have been used.
[10]Further information at http://www.wireshark.org/.

dimension has been found, it is useful to conduct other tests on the decryption of files of that dimension with the aim to verify how much time is required in the decryption of such quantities.

It is also important to say that, all the tests have been performed considering also the VNC developers' specifications as regard the bandwidth to be used in order to guarantee that the video stream passes without any type of delay. The bandwidth specifications may be listed in the following few points:

- 33 kbps for accessing computers having a poor graphic (resolution) and small frame dimension.
- 128 kbps is for those connections where the frame rate and the bandwidth are higher, but having simple imagery with reduced colour resolution. Furthermore, a low colour depth is correctly *supported*.
- 1Mbps is for the full colour access with simple imagery, or complex imagery and reduced colour depth or frame rate.
- 100Mbps for a real time connection like if the client was in front of the server side.

As regard the testbed hereby utilized, a 1Gbps Ethernet connection has been used, in order to ensure a seamless communication between the two endpoint. This is ensured also by the VNC developers who declare a 100Mbps connection would be sufficient

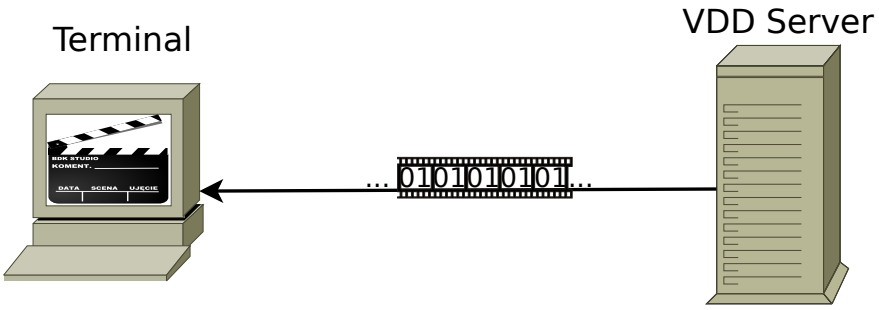

Figure 3.5: The video stream from the Server to the Terminal.

The procedure to compute the frame dimension consists in a *video sequence playback*, which is visualized at the client side for a fixed time of 130 s. In order to see how the frame dimension has been computed, let's consider the following premises for instance:

- Number of colours: 2^{24}
- Resolution: 1024x768
- Screen Frequency: 60Hz
- Sampling time: 130.0 s

Once the video sequence has started to play, Wireshark begins to capture VNC packets on the client and automatically stops after 130 seconds. The statistics produced by Wireshark can be reported in Table 3.1.

Packet lengths	Avg Packet Lengths	# of packets	Percent
5120 - 10239	7679.5	0	0
2560 - 5119	3839.5	2	0.0909
1280 - 2559	1919.5	4	0.1818
640 - 1279	959.5	7	0.3182
320 - 639	479.5	6	0.2727
160 - 319	239.5	3	0.1364
80 - 159	119.5	0	0
40 - 79	59.5	0	0

Table 3.1: Wireshark statistics.

The first column represents the packet lengths (bytes) in a certain interval (i.e. there are 2 packets having a length between 2560 and 5119 bytes in the second row). The second column instead, represents the mean packet length computed as it follows:

$$MPL = \frac{MinLength + MaxLength}{2}$$

In the case in point, (in the first row) $MPL = \frac{5120+10239}{2} = 7679.5$ bytes. The third column is the number of packets of dimension *Packet length* that have been captured and the last column represents the percent of packet of that dimension with respect to the total amount of VNC packets received by the client.

Once the table Table 3.1 is available, it is possible to compute some other value. The overall average packet size is:

$$\begin{aligned} 7679.5 \cdot 0 + 3839.5 \cdot 0.0909 + 1919.5 \cdot 0.1818 \ + \\ 959.5 \cdot 0.3182 + 479.5 \cdot 0.2727 + 239.5 \cdot 0.1364 \ + \\ 119.5 \cdot 0 + 59.5 \cdot 0 = 1166.716 \ \ Bytes \end{aligned}$$

The total amount of packets is the sum of all the values reported in the column three, which is equal to 22 while the total Bytes transmitted are

$$NumPackets \cdot AvgPacketLength = 25667.752 \ \ Bytes$$

so the speed (data rate) at which the frames are delivered to the client is

$$\frac{TransmittedBytes}{SamplingTime} = 197.444246154 \ \ Bytes/sec \rightarrow 0.1928166466 \ \ KBytes/sec$$

Finally the frame dimension is:

$$\frac{DataRate}{ScreenFrequency} = 11.57 \ \ KBytes$$

Summarizing we have:

- Average packet size: 1166.716 Bytes
- Number of packets: 22
- Data transferred: 25667.752 Bytes
- Packet frequency: 0,19 KByte/second
- Frame dimension: 11.57 KByte

The test above (which is the test n. 11 reported in Table 3.2) has been repeated many times, in order to verify many different combinations of resolutions and colour depths. The results are not always as expected but this highlights that the frame dimension depends on several factors other than resolution and colour depth.

Test n.	Resolution	Colour depth	Frame dimension (KByte)
1	1680x1050	8	224.83
2	1024x768	8	15.96
3	640x480	8	15.89
4	1680x1050	64	50.51
5	1024x768	64	129.50
6	640x480	64	15.89
7	1680x1050	256	11.66
8	1024x768	256	62.94
9	640x480	256	343.40
10	1680x1050	16777216	15.89
11	1024x768	16777216	11.57
12	640x480	16777216	24.62

Table 3.2: Final results of the frame test.

Despite that, thanks to the tests reported in Table 3.2 it is possible to have an evaluation on the average frame size in normal conditions (i.e. without performing anything producing a high workload like a movie or 3D graphical software) where the user utilizes a word processor for instance. The average value of the forth column of the Table 3.2 gives us the average frame dimension which is 76.88 KBytes.

Another important issue is related to the VNC stream compression. As a matter of fact, it is possible to define a compression level in the VNC client when connecting to a remote desktop. As regard the case in point, for the preferred encoding, the compression level has been set to *Automatic*; this means VNC automatically detects the network speed and makes its adjustments according to it. If the link between the client and the server is fast, the client will only request full colour updates. VNC also gives the possibility to manually specify other compression levels, respectively from the most effective on slow networks to the most effective on high speed networks: ZRLE[11], Hextile[12] and Raw[13].

In these conditions, we are working with the lowest compression level (due to the 1Gbps Ethernet LAN of course) and the only variation that has been made, apart from the resolution, regards the colour depth. In other words we are in the worst case, and the average frame dimension equal to a value which is lower than 100KByte is a very good result, considering that a further compression level may be introduced and manually specified in the client preferences. In respect of this, it must also be considered that, another remote desktop client similar to VNC, which is called TightVNC[14], gives the possibility to perform a more accurate management of the compression level to the final users. The compression levels are roughly the same of RealVNC with the addition of *Tight* and *Zlib* which give to the users another possibility to define a more precise compression level for each one in a 1-9 range value.

The Table 3.2 can also be graphically analyzed in order to better understand how the frame dimension trend is, in the different cases.

The values represented in Figure 3.6 highlight the difference of the frame dimension that has been obtained at equal resolutions[15] in the various tests at the colour depth variation. As it is possible to see, there is not a regular trend apart for the first group (resolution at 1680x1050). Actually, in the first group we may notice that the results are as expected. In other words, we expect that the more the resolution decreases the less the frame dimension is big if the average group value is considered. In the second group, there is an anomaly for the colour depth equal to 8, and the decrease begins at the colour depth equal to 64.

The chart in Figure 3.7 is another representation of the values in Table 3.2. The purpose of this chart is to highlight the differences between the different frame dimensions where the comparison is made at the same colour depth and at a different resolution. The chart hereby considered shows that the more the resolution decreases, the less the frame dimension is big. This is true for group 1 and 2, but the anomaly in group 3 is very noticeable. It must also be noticed that the values in

[11]ZRLE stands for Zlib (further information for Zlib at http://www.zlib.net/) Run-Length Encoding and combine zlib compression, tiling, palettisation and run-length encoding[15].

[12]Hextile is a variation on the RRE idea (RRE stands for rise-and-run-length encoding and as its name implies, it is essentially a two-dimensional analogue of run-length encoding[15]). Rectangles are split up into 16x16 tiles, allowing the dimensions of the subrectangles to be specified in 4 bits each, 16 bits in total[15].

[13]The simplest encoding type is raw pixel data. In this case the data consists of *width* × *height* pixel values (where *width* and *height* are the width and height of the rectangle)[15].

[14]TightVNC is a free remote control software package. Further information at http://www.tightvnc.com/.

[15]Each column group is at the same resolution.

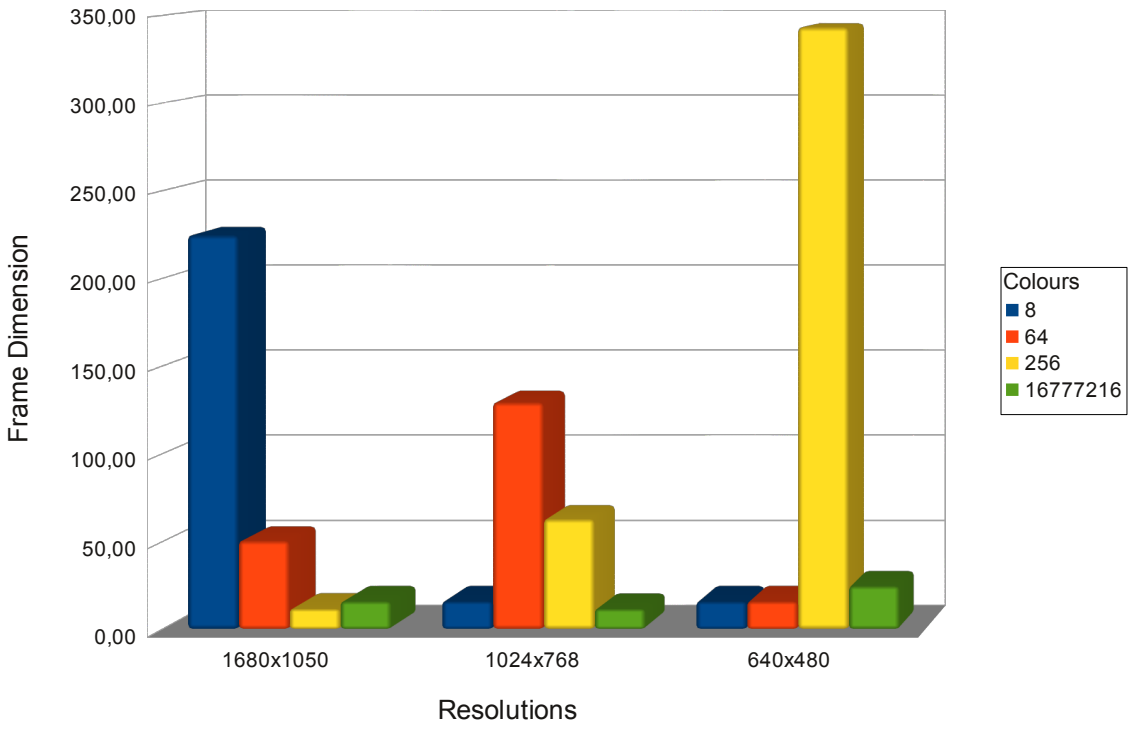

Figure 3.6: The frame tests chart for the comparison between different tests at the same resolution.

group 2 should be higher with respect to the group 1 if considered by colour depth (the blue bar on group 2 should be higher than the blue bar on group 1), because of the higher colour depth. The same should be said for the group 3 with respect to the group 2 and 1 respectively, but this is true only for the colour depth equal to 256 in group 3. This behavior emphasizes the fact that there is not a precise rule to find out the frame dimension, even if, with the tests conduced in this context, it is possible to make some assumption and consideration.

The fact that emerges from the charts in Figure 3.6 and in Figure 3.7, is that we may have some high peak in the frame dimension, up to ∼200 - ∼300 KByte but, fortunately, only in few and isolate cases; nevertheless the average frame size is much more lower than those values. In the light of fact, once the average frame dimension has been derived from the tests described above, it is possible to establish what are the new tests that are needed. How much time is needed to decrypt a frame having an average size equal to 76.88 KBytes? Starting from this value, it is possible to consider an upper bound of 100 KByte (this means we accept occasional frame loss in the stream) frame size for the decryption tests, as described in Section 3.4.2 and to make new considerations in base of the size of the data to be decrypted at the client side.

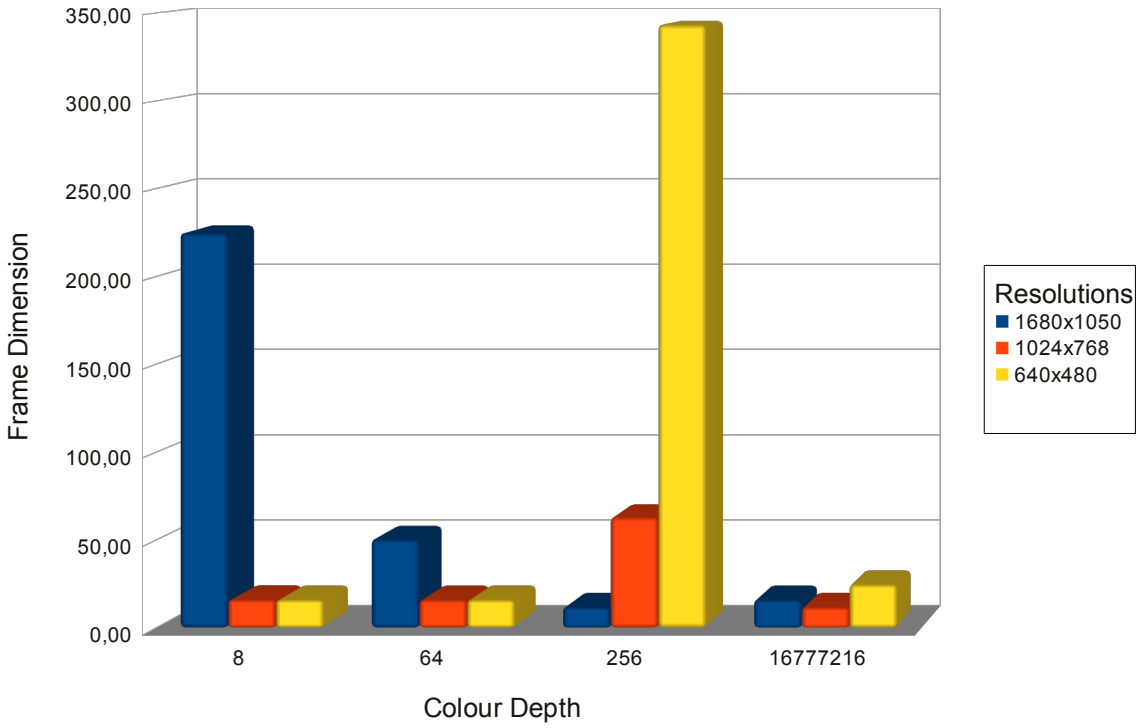

Figure 3.7: The frame tests chart for the comparison between different tests at the same colour depth.

3.4.2 Frame decryption time

The outcome of the tests described in Section 3.4.1 bring us to make new tests with the aim to evaluate the time required to decrypt a frame having an average size of about 80 KByte. In order to measure the time to decrypt a quantity of data equal (more or less) to the frame it is possible to perform a file decryption using the API supplied from the developers of Homomorphic Encryption in Java or even better in C. Unfortunately there are very few implementations in C or Java: as regard the C implementation there is an Open Source library which is called *libpaillier*[16] developed by John Bethencourt, while the Java implementation is called *thep*[17], developed by Michael Clark. Both implementations are absolutely noteworthy, and the experiments have been conduced using both of them, but the C library will be considered in this circumstance, because of the fact that the C language offers much better performances. As already described in Section 3.1

[16]Paillier is a public key cryptosystem which offers an additive homomorphism, making it very useful for privacy preserving applications. This is a simple C library which implements Paillier key generation, encryption, decryption, and also makes it easy to use the homomorphism. Further information at http://acsc.cs.utexas.edu/libpaillier/

[17]The Homomorphic Encryption Project - Computation on Encrypted Data for the Masses

we have many different Homomorphic Cryptosystems, but the few resources in term of available programming API on the Internet, obliged our research to use only the Paillier cryptosystem C implementation (see Appendix A.1 for the test algorithm pseudocode). Nevertheless, the test results are encouraging, as shown later on. As regard the Fully Homomorphic Encryption, no free and Open Source implementation are known at the moment, at least for the tests to be performed for our purposes.

The studies conduced so far, bring us to define a frame (file) size range value for the decryption tests, in order to analyze the trend as the frame dimension increases with respect to the time that is needed to decrypt such quantity. The values to be considered are: 1KByte, 10KByte and 100KByte, being the last one an upper bound for the frame size we found in this research.

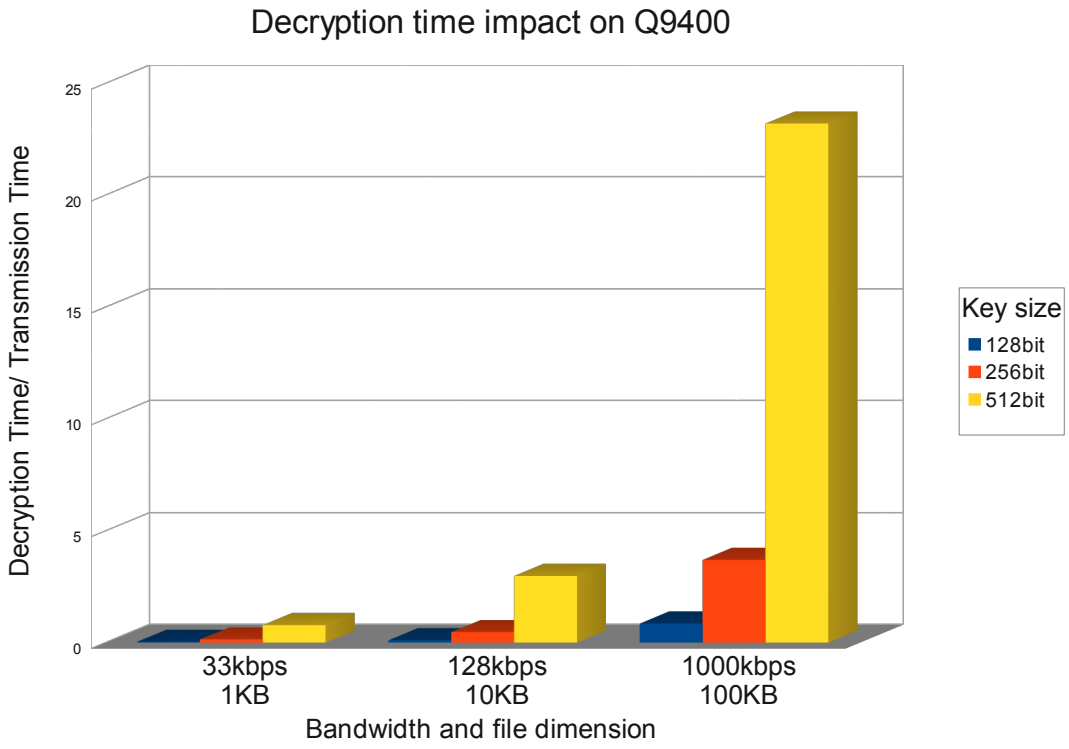

Figure 3.8: The decryption time impact on an Intel Q9400 CPU.

The tests consist in the encryption of text files, of the dimensions declared above, and the consequent decryption, measuring the time this operation requires. Of course, the operations of encryption and decryption have to be performed handling the files in blocks of 8 Bytes each but this has to be implemented separately, since the library doesn't support the file manipulation as it is. Another research regards the choice of the keysize values: even if the more the keysize is high the more the cryptosystem is secure, the compromise between efficiency and security let us

choose these keysize values to be tested: 128 bit, 256 bit and 512 bit. We are aware of the fact that, a bigger key size would certainly be better for privacy, but these three values are sufficient to show how is the trend of the decryption time depending on the file size and the key and without disrupting performance too much.

The chart reported in Figure 3.8 shows the test results for the decryption of files with respect to the transfer time on an Intel Q9400 CPU, which is a Quad Core processor. To be more precise, the chart shows the impact of the time to decrypt a file with respect to the time to transfer the file itself at different bandwidth and key size, showing how much the time to decrypt is worse than the transfer time. The bandwidth values that have been considered are the same described in Section 3.4.1 excluding the 100Mbps case; so the Figure 3.8 shows the ratio between the decryption time values and the transmission time (found considering the bandwidth values above and the file size). The software implementing this test, uses the multithreading technique, in order to exploit the multiprocessor as much as possible. The multithreading allows to assign to each CPU Core an equivalent number of blocks to decrypt. Of course, the higher the number of CPU is, the more the performances are satisfying. The analysis of the chart in Figure 3.8 shows a *knee* in correspondence of the 10 KByte file size @128kbps, where the decryption delay increases fast for the 512 bit key size with respect to the transfer time. After that point, there is a noticeable increase of the decryption time for a key size of 512 bit. It is clear that the decryption times are quite similar if a 1 KB or a 10 KB file is handled at the different key sizes hereby considered, but if the key dimension and the file size increase, there is a big difference between the ratio in case of key sizes less than 512 bit and key sizes equal or greater than 512 bit.

Another consideration can be made: the test results shown in Figure 3.8 and also in Figure 3.9 are satisfying enough. Actually, the only con is for the 512bit key size @1Mbps of bandwidth. Stated that, it is possible to say that if you need high graphic computational power you have to renounce to the privacy protection and vice versa.

In spite of the time to decrypt a file whose size is about 100KByte requires a long period, we also have to consider that the experiments have been conduced in the worst case. Actually, if a compression codec is used, the time to decrypt a 100KByte file would be certainly lower, and the video stream decryption would become feasible without any doubt.

In any case, as also described in [6], the Paillier Cryptosystem is computationally comparable to RSA and the main purpose of this research is mainly to find a Fully Homomorphic Cryptosystem to be applied in VDD.

In order to see how the performances can be improved, the previous test has also been performed on a faster workstation, having an Intel Core i7 920 CPU (8 Cores) and the results are shown in Figure 3.9. As a matter of fact, even if the threads have to be divided among more processors the improvements are not so evident but the same considerations can be made as for the Quad Core case. Actually, as expected, the knee point is still in correspondence of the 10 KByte file @128kbps.

The Figure 3.10, shows in details how much the Core i7 is faster with respect to the Q9400. There are three different cases to be considered, each one having a different key size: respectively

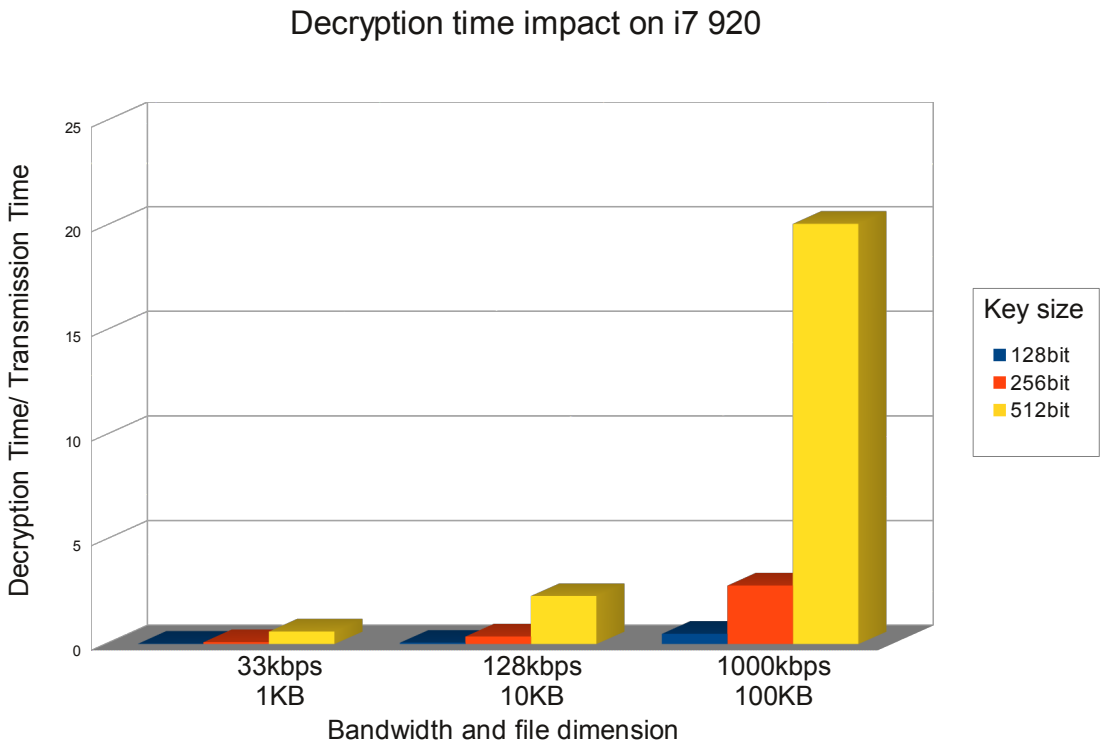

Figure 3.9: The decryption time impact on an Intel Core i7 920 CPU.

128, 256 and 512 bit. Every chart highlights no substantial difference between the decryption of a 1-10 KByte file while for the 100 KByte file a major gain is noticeable. Actually, if a Core i7 is used a 42,7% gain results for the 128 bit key when decrypting a 100 KByte file, while a 24,23% and 13,4% gain results respectively for the 256 bit and 512 bit key size. Despite that the larger the key is, the less the gain is big, when a faster processor is used, this doesn't mean that a more powerful CPU (e.g. a 24 Core CPU), wouldn't improve the performances further on. Another parameter that may also be considered is the average decryption time which is 2736,81 ms for the Quad Core and 2284,94 ms for the i7, corresponding to a gain equal to 16,52% using the fastest CPU.

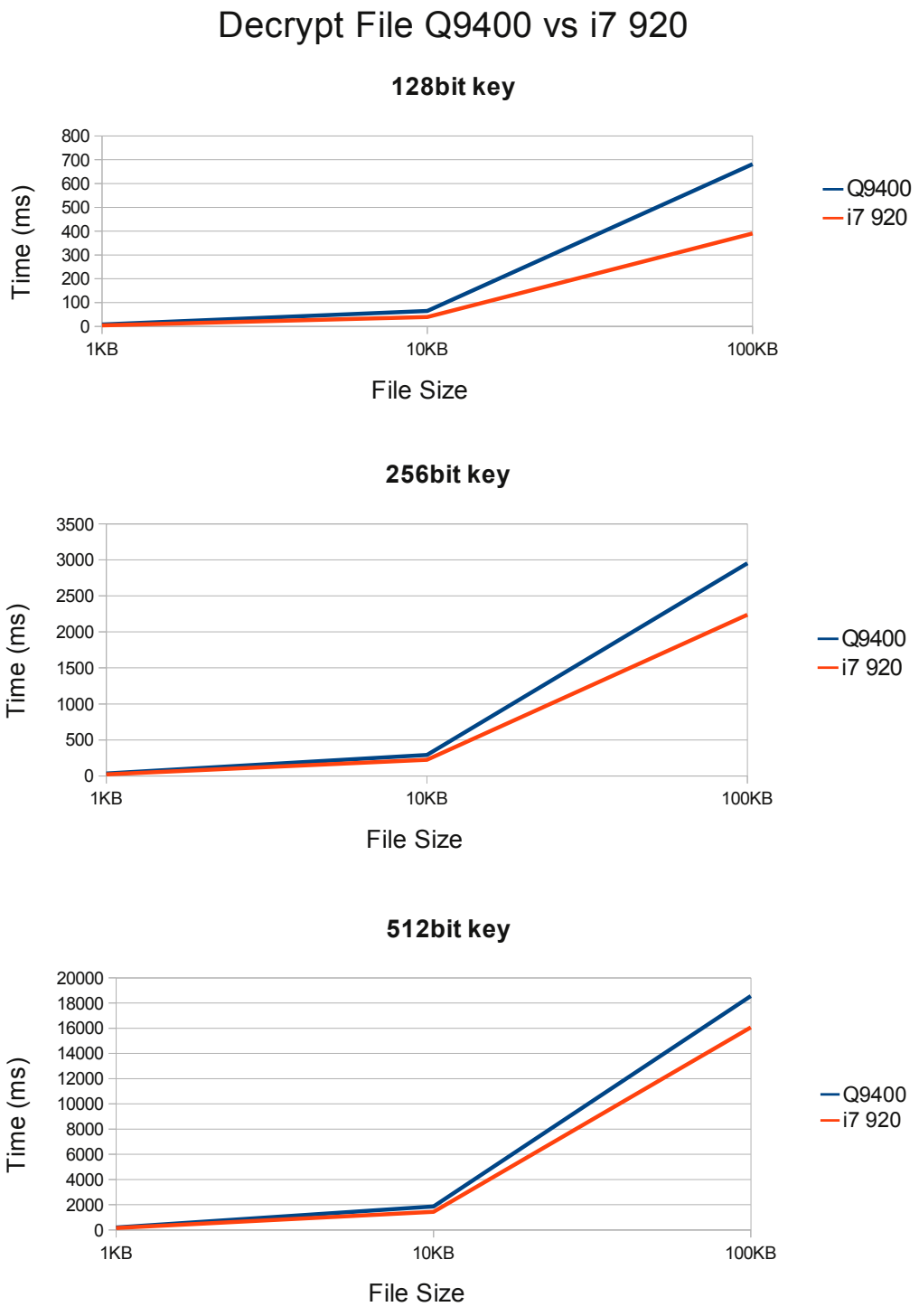

Figure 3.10: Comparison between Intel Q9400 and Core i7 for the frame decryption.

3.4.3 Performance tests on data processing

Another very important aspect for the HE in VDD is about the performances on data processing. The operations on data are performed at server side, as opposed to the frame decryption which is at the client side in the Hedge Proxy. With the purpose to analyze how the data processing impacts onto the VDD server, it has been decided to choose an operation to be Homomorphically performed on it: the multiplication between two integer numbers has been chosen as representative of a generic function/operation on data. This practice could also be extended for any other operation, especially if a Fully Homomorphic Encryption scheme is used. The tests have been conduced both with plaintext and ciphertext integers also with the purpose to highlight the differences between an operation in the plaintext domain and an operation in the ciphertext domain. The Appendix A.2 shows the *serial* and the *parallel* algorithms used for the tests.

All the tests make a comparison between the serial and the parallel multiplication execution time. This means a certain number of multiplications (ten millions for instance) are executed on the CPU: in the first case there is no multithreading execution (a unique *"for" cycle* executes all the multiplications) while in the second one all the multiplications are *divided* by the number of CPU cores with the aim to assign an equal number of operations to each one. The Tables 3.3 and 3.4 show the multiplication test results on an Intel Core i7 CPU, respectively in serial and in multithreading mode. The second column represents the time to perform 10 millions of multiplications in the plaintext domain, while the third column is in the ciphertext domain. The *cipher/plain* ratio indicates how much the time to perform the multiplications in the encrypted domain is greater than the time required in the plaintext domain.

key size	plaintext (ms)	ciphertext (ms)	cipher/plain ratio
128	137	520	3.8
256	137	585	4.27
512	137	742	5.42

Table 3.3: 10 millions of *serial* multiplications on Intel Core i7 920.

In the case in point, we see that the cipher/plain ratio in the serial case is low enough for each key size, while in the multithreading case it is (too much) high. This is caused by the high time required to perform the multiplications in the encrypted domain.

If a comparison between the second columns of the Tables 3.3 and 3.4 is made, it is clear that we are dealing with an operation (i.e. the multiplication) which admits fast parallel algorithms, being the multiplication and the addition of integers in the class NC^1 [18]. Actually, as regard the plaintext case, the multithreading technique allows a gain of 71.54% with respect to the serial computation. This is not true if the ciphertext case is considered; the computation doesn't benefit

[18] In complexity theory, the class NC (for "Nick's Class") is the set of decision problems decidable in polylogarithmic time on a parallel computer with a polynomial number of processors.

key size	plaintext (ms)	ciphertext (ms)	cipher/plain ratio
128	39	769	19.72
256	39	2262	58
512	39	6684	171.38

Table 3.4: 10 millions of *parallel* multiplications on Intel Core i7 920.

of the parallelization at all and the time required to perform the same operations is much higher. As regard the time to execute the operations in the encrypted domain, the Table 3.3 shows a result worsening not so high as for the results in Table 3.4. At this point, this shouldn't seem nothing strange, since one usually expects that an operation in the encrypted domain requires much more time respect to the plaintext domain, if each Table is considered lone, but the time for the multiplication of two ciphertexts is too high and this is considered to be an anomaly.

The reason why this anomaly is present is due to many factors; in the *serial* case the worsening is due to the Paillier cryptosystem, which is inefficient under many points of view, e.g. for the denoising procedure, as already discussed in Section 3.2 and documented by Jibang Liu in [11]. Actually the multiplication may produce results exceeding a 32-bit integer and the execution time increases a lot due to the denoising procedure, but also to the fact that more bits are required and more bit-wise operations are performed. In the *parallel* case instead, the same thing happens but the worsening is higher also because of the fact that a lot of threads have to be created, defeating further on the benefits of parallelization (but just in case of the combination between the denoising factor and the threads creation). As already mentioned, this is not something definitely negative, since the research on Homomorphic Encryption goes ahead and new API will be certainly released as C libraries also for faster algorithms.

The values reported in Tables 3.3 and 3.4 are graphically represented in Figure 3.13, highlighting the differences as the number of multiplications increases. The same tests have also been performed on a Dual Core Intel T7200 CPU and on a Quad Core Q9400 CPU (see Figure 3.11 and Figure 3.12 respectively), in order to see how much the whole system may be improved, if the VDD server has more power, especially if a faster CPU is used. In every case, the time to perform parallel multiplications is higher than the serial case, even if a faster CPU is used.

The Figure 3.14, shows the comparison between the three CPUs as regard the serial multiplications. As it is easy to understand, the more the CPU is fast the less is the time to perform the multiplications. The Figure 3.15 instead, highlights an anomaly for the Q9400 CPU. Actually, for the 256 bit key size case, the time to perform the multiplication is higher with respect to the T7200 CPU. This is true if absolute values are considered, whereas if the average values for each CPU is considered the Q9400 is faster than the T7200.

As a final consideration, it must be said that the Figure 3.14 and the Figure 3.15 are very interesting, not only as regard the absolute values but also because of the fact that they indicate

what happens if a faster CPU is used and how much the technological development is relevant for the Homomorphic Encryption in terms of trends.

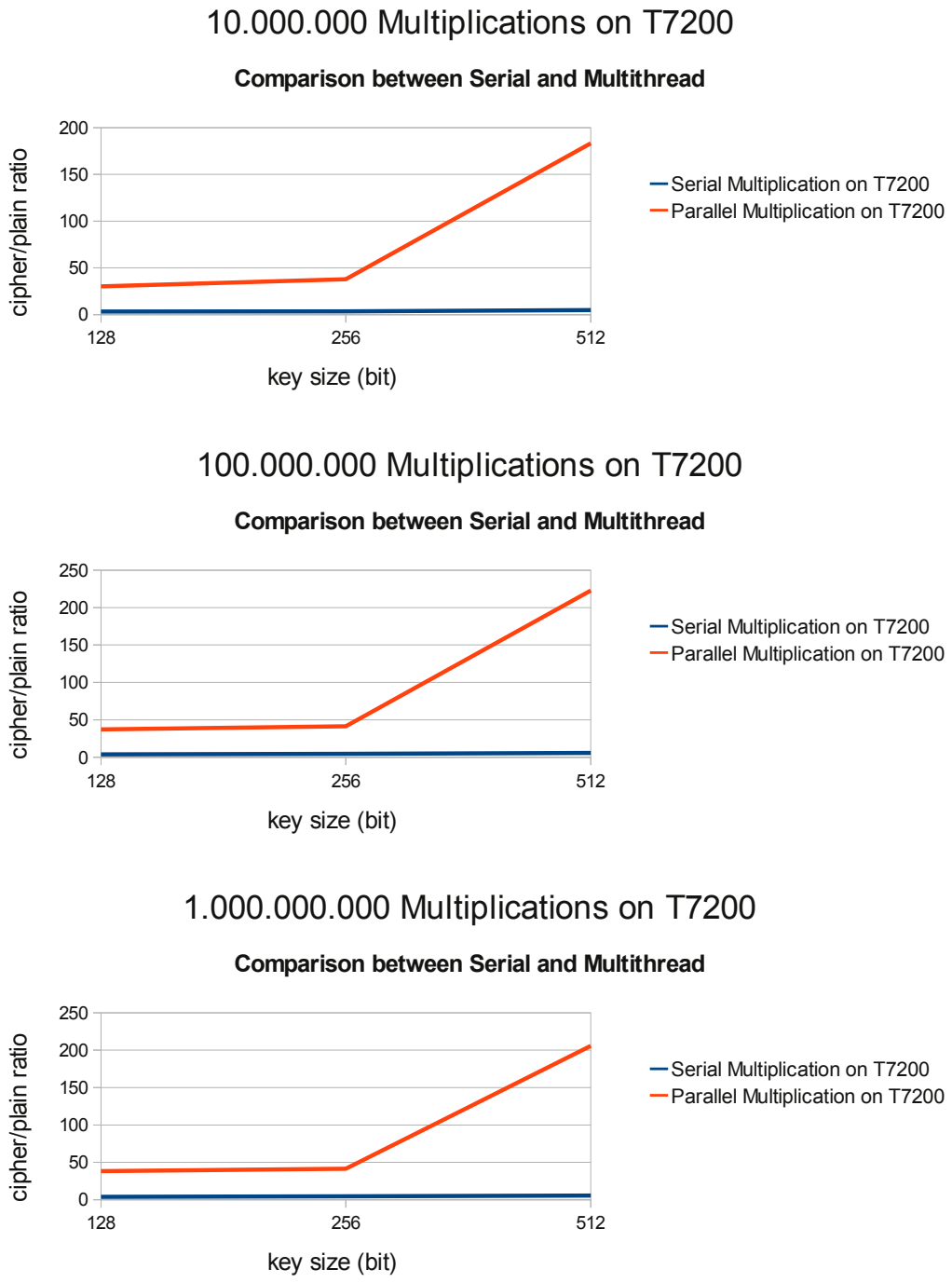

Figure 3.11: Comparison between the serial and parallel multiplication on Intel T7200.

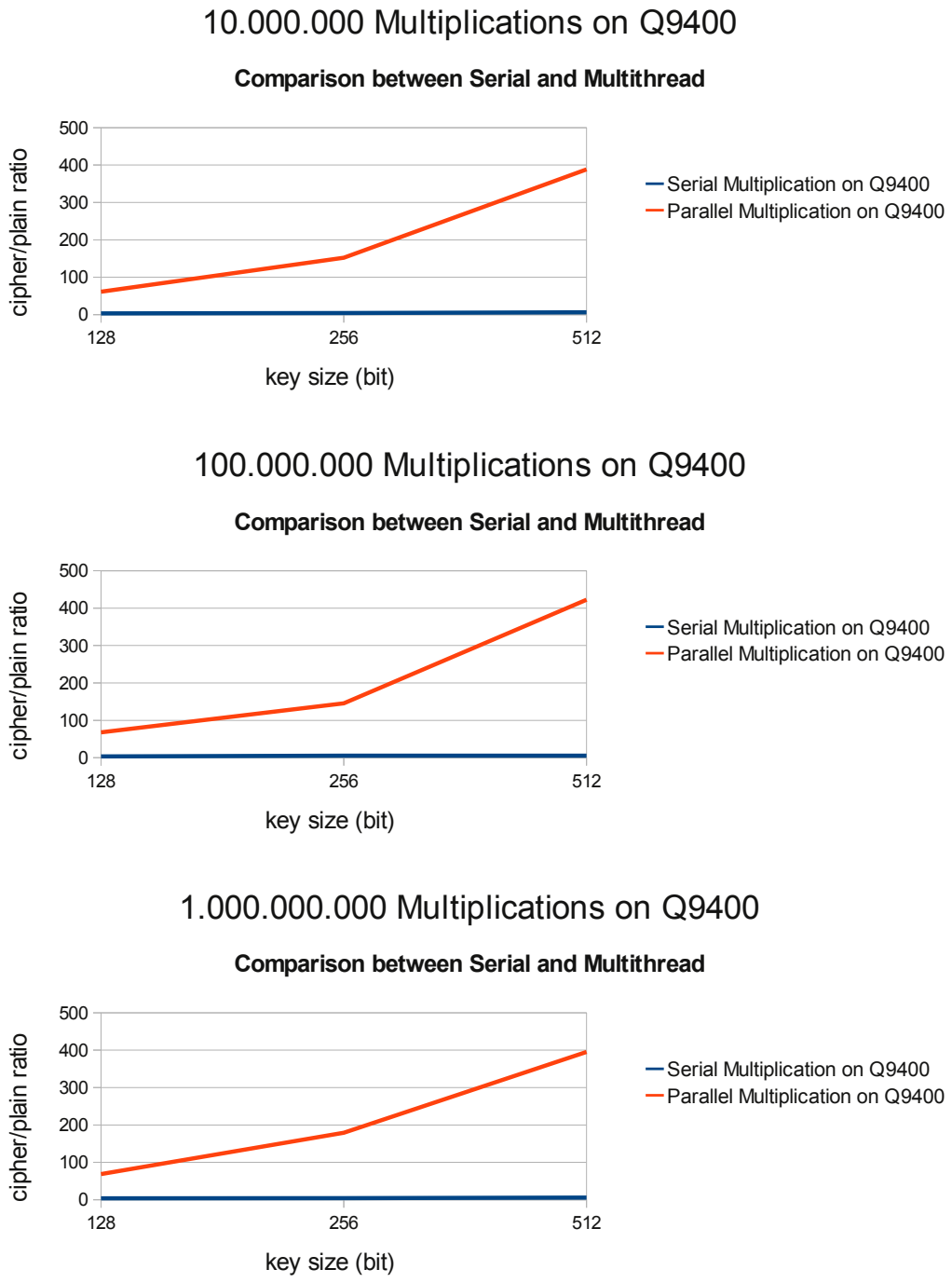

Figure 3.12: Comparison between the serial and parallel multiplication on Intel Q9400.

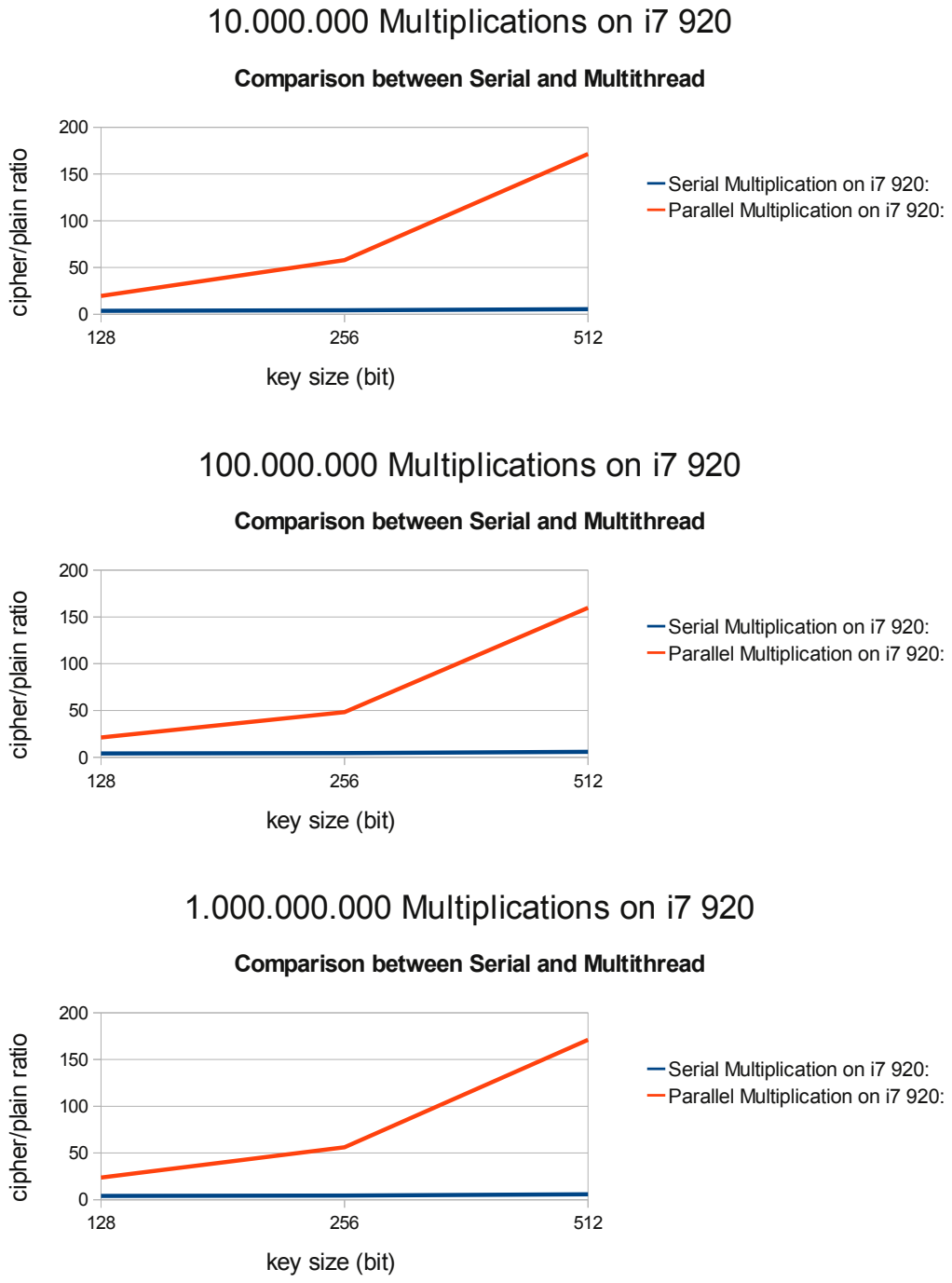

Figure 3.13: Comparison between the serial and parallel multiplication on Intel Core i7 920.

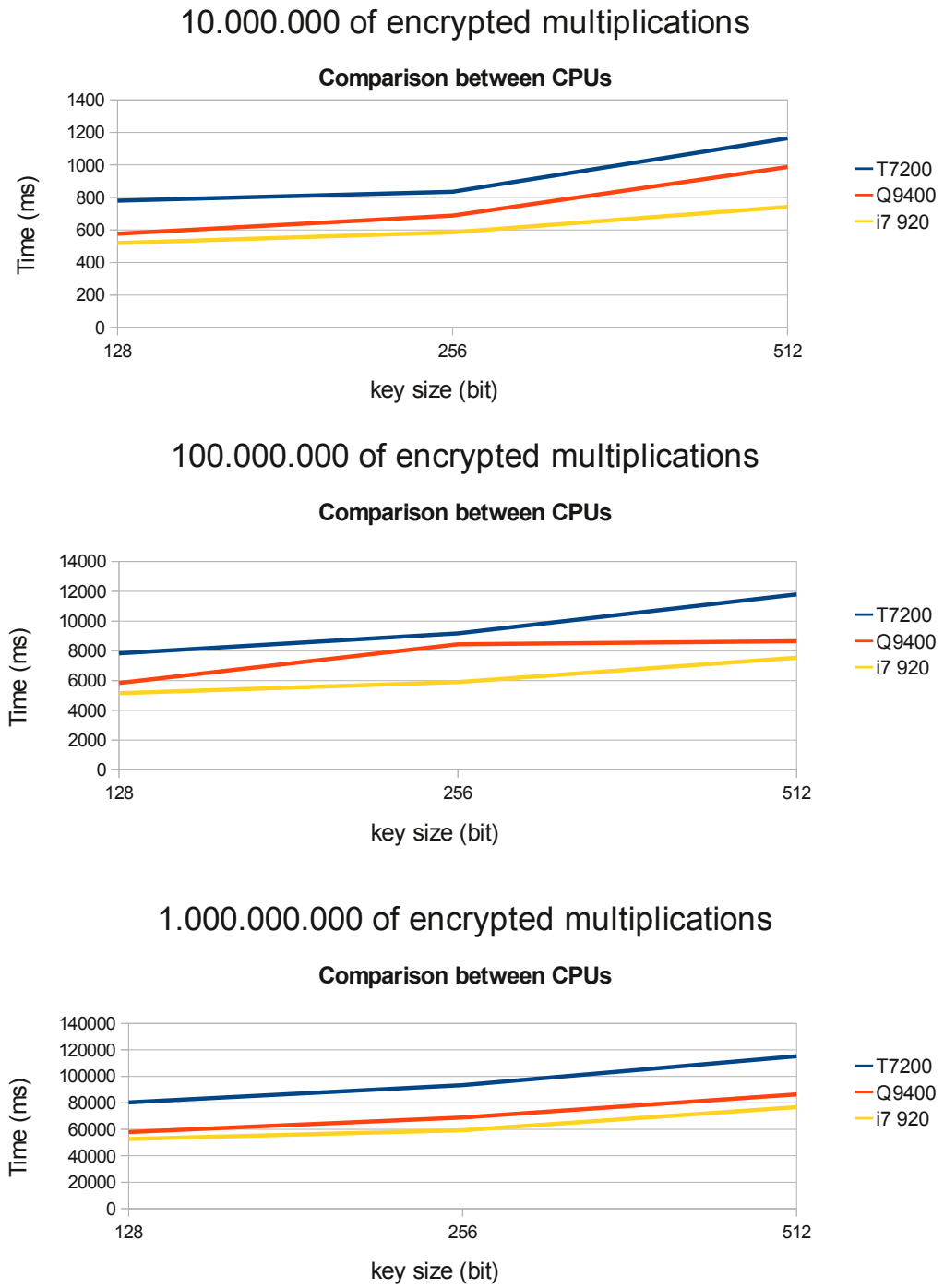

Figure 3.14: Comparison between the serial multiplication time on different CPUs.

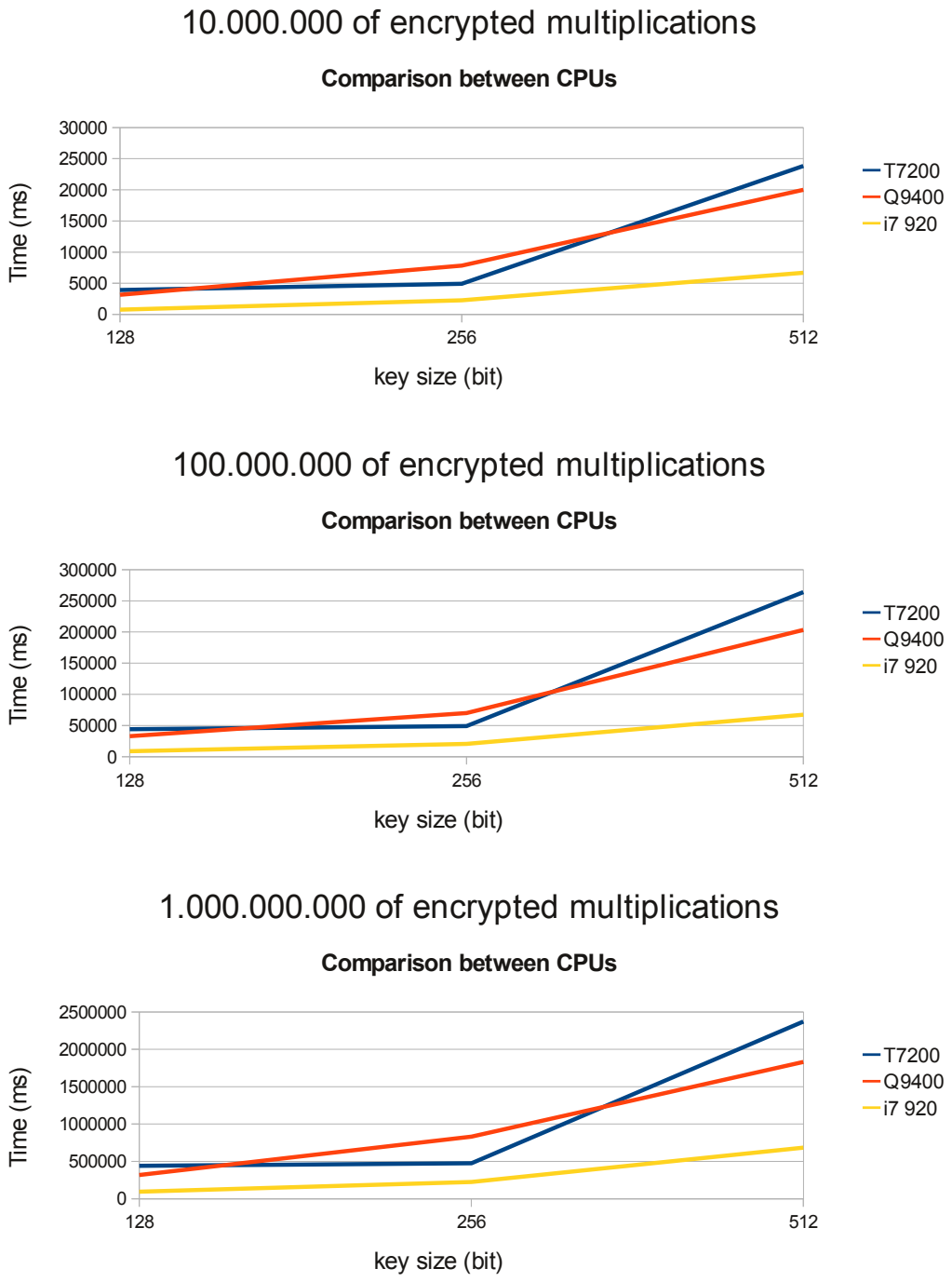

Figure 3.15: Comparison between the parallel multiplication time on different CPUs.

3.5 Final remarks on HE

This Chapter presented a privacy solution which is definitely something different with respect to the classic disk partitions encryption and it's certainly another approach to the privacy protection issue. At the same time the HE solution is rising other problems related to the complexity of the whole architecture and to the performance decay coming from the application of a Hedge Proxy. Nevertheless, the research on HE has got the point on what is the right direction to reach the target having the purpose of the users' privacy protection: the data manipulation in the encrypted domain. Of course this may have bad repercussions due to the whole system performance as the more the required privacy level is high, the more the system overload increases. We are still at the early stages, since the Fully Homomorphic Encryption is a very recent discovery and no usable implementations are available now.

Another point to be considered is about the feasibility of a HE-Enabled VDD; actually, as it will be discussed in Chapter 4, the complexity derives also to the fact that *every software* has to be *informed* that it is dealing with encrypted data and mainly with homomorphic operations. As regard the performance aspects, the experiments conduced in this Chapter show a heavy performance disruption when the level of privacy and/or graphic quality increase. Fortunately, the system is agnostic to hardware innovation or cryptosystem efficiency, hence several degrees of improvements are envisionable, also considering the big investment put in HE, for example by IBM and HP. Paillier Cryptosystem is not the only one around, even if at the time we are writing is the only one providing the code for download. The development of new API for the Fully Homomorphic Encryption implementation, suitable for VDD and certainly more efficient than the one we used for the tests of the previous sections, is another milestone to be reached by research in the field.

The Homomorphic Encryption is a very interesting solution, especially if compared to the alternative discussed in Section 2.5.2 or to the disk encryption in general, but some pros and cons have to be considered: HE allows to operate on encrypted data stored at Server side, keeping the user's privacy always safe and it also allows to be used as a traditional Cryptosystem as well as it gives the possibility to encrypt also the video stream delivered at the terminals, but it is hard to be implemented on systems and the performances are not so satisfying at the moment. On the other hand, the Tomb solution doesn't allow to perform operations on encrypted data and hence is not able to ensure the user's privacy at the moment in which the data are decrypted server side in order to be accessed client side; as we have seen, this solution is easy to implement on VDD and comfortable for the user but doesn't fit all our needs, as discussed above.

For the reasons discussed so far, it is worthwhile to continue in the research of the Homomorphic Encryption field, also with the hope that the Fully HE will be suitable for VDD but also for the Cloud Computing in general.

Chapter 4

Conclusion and future work

This thesis presented two solutions for the privacy protection in Desktop as a Service and the entire work is mainly focused on the Homomorphic Encryption research. The evaluations and the considerations made so far, encouraged our research to go ahead with the awareness that this is the right direction for a solution to the privacy protection in DaaS. The work on Homomorphic Encryption is very important for the VDD enhancement and for the definitive privacy solution in it. Another important purpose of this research is to apply the privacy solution proposed in this thesis also to a Cloud Computing environment in general.

4.1 Related work

Nowadays, there are many Desktop as a Service solutions on the market, especially with the boom of Cloud Computing. The convenience to store online our files having them always at our disposal, the need to have a virtual Desktop with our applications wherever we are in the world, made Cloud Computing something nearly indispensable also for these reasons. As an example, Cendio ThinLinc[1] proposes a solution which is similar to VDD, and it makes applications available from decentralized and remote clients, in a Desktop environment. As regard the remote access to the Desktop environment, currently guaranteed by the gtk-vnc plug-in for Mozilla Firefox in VDD, another market solution is represented by LogMeIn Pro[2] giving the users the access to their computers remotely.

The diffusion of software like ThinLinc and LogMeIn, convinced us that VDD is something really alternative especially if we consider that it is fully open source and free software, released under the GPL license. For this reason we trust in the development of all the components of VDD (i.e. LTSP, X.Org, the virtualization systems, etc.) but in particular in the Homomorphic Encryption research.

[1] ThinLinc is a Linux Terminal Server solution. ThinLinc is used to virtualize desktops and applications by the use of Server Based Computing. Further information at http://www.cendio.com.

[2] LogMeIn allows the access your LogMeIn computers and to manage your LogMeIn account. Further information at https://secure.logmein.com/.

Another important contribute to VDD has been certainly brought by Ubuntu Enterprise Cloud (see Section 2.5) giving us the possibility to abstract from the virtualization level and concentrate on many other aspects including the privacy as stated here. At the moment UEC already offers a great virtualization platform providing a comfortable and compliant IaaS layer and for this reason it will be part of VDD for sure in the next years.

All of these genres of platforms have to face with the privacy issue and this is why the parallel research on Homomorphic Encryption is important. A great step ahead in this field would mean the high probability to defeat the privacy violation issue in Cloud Computing. At the moment the directory or disk encryption seems to be the most practical and easy solution (see Section 2.5.2); actually it has been the first step to bring some (incomplete) privacy solution to VDD, but the problems described in this thesis convinced us to make something more.

The most important work for the issues discussed in this thesis is certainly represented by the Craig Gentry's research on Homomorphic Encryption. He has been the first who demonstrated the possibility to make a Fully Homomorphic Cryptosystem, giving the go to the VDD research team to investigate on the possibility to solve the privacy issue thanks to this innovation.

4.2 Contribute

The Chapter 2 describes the state of the art especially as regard the VDD development status. Furthermore it introduces to the Cloud Computing and to the issues related to it with the purpose to understand what are the problems and how it is possible to solve them, giving particular attention to the privacy issue. The first contribute is given by the introduction of UEC, which allowed to increase both the performance and the efficiency of VDD; then the Tomb encryption solution has been adopted in place of the entire disk partition encryption, giving the users more flexibility and more possibility of choice on what to do with their data (if they want protection or not).

Figure 4.1: The private visualization of user's data on the thin client screen.

The Chapter 3 is the real core of the research. The real innovations actually is in the remote encrypted data manipulation, giving the users the complete certainty to protect their data without never decrypt them at the server side. The Hedge Proxy solution represents something really innovative. It allows only the final user to see her data (see the Figure 4.1 as a picturesque example of a private channel built between the final user and the terminal). A particular emphasis is also given by the difference between the privacy solution presented in the Chapter 2 and the one discussed in Chapter 3, discussing also the pros and the cons coming from each one.

Of course a theoretical approach has been presented but a logical design of the whole architecture has been described too, in order to evaluate the feasibility of the Homomorphic Encryption solution on VDD and in the Cloud Computing in general. Of course this may have a great impact on all the software constituting the VDD core system and also on the Desktop applications running in it, but the bet (i.e. the use of Homomorphic Encryption for a privacy solution) is too high to not be considered at all. At the moment the current question which is still remaining is: *where we can go without the Homomorphic Encryption if the privacy solution has to be solved?* As Jibang Liu says in [11], *even if the computations on encrypted data are very slow we can remain optimistic for the future of Homomorphic Encryption*; so this is one of the main reason why we don't want to give up in this direction. Actually, as this thesis proposes, it is possible that many vendors would be able to supply for services based on encrypted data manipulation in the next future. This could be certainly boosted by the utilization of hardware accelerators, co-processors, graphic cards or even new instructions in future CPUs. This would be the right way to face with the denoising problem discussed in Section 3.2. It must also be considered that the contribute given by HE is already high even if it is still at an early stage at the moment.

4.3 Future work

The solutions presented in this thesis, giving a major emphasis to the one which uses the Homomorphic Encryption, are certainly not definitive and they require to be considered more in depth and implemented. The Hedge Proxy solution is something really useful and it is required to be implemented at client side, as already stated in the Chapter 3. Another very important issue consists in how to have a HE-Enabled Desktop application: every software at the user's disposal must be modified in order to allow the data manipulation in the encrypted domain, and this can be accomplished only making a modification to the source code of each one or maybe introducing a plug-in. This would clearly introduce a revolution in application development. Another hypothesis, to be considered as a solution to the problem risen by the complexity of the modification of every cloud application, could be represented by the possibility to modify the compilers. Actually, the idea is to compile every software running on VDD from sources with a particular modified version of GCC, the GNU Compiler Collection, in order to automatically enable the software for the homomorphic encryption. A detailed study of the compiler could make sense, in order to see if a modification is really needed with the purpose to ensure that the Homomorphic property doesn't

get lost in the operations sequence for the compilation result. In any case, the compilation of the source code must respect the Homomorphic property of the encrypted operands. As a matter of fact, another piece of research could be related to this fundamental aspect which is very important for the feasibility of the solution presented in this thesis. As regard the Hedge Proxy solution, instead, the visualization system (as an example X.Org) currently being used has to be considered for a modification, where a sort of wrapper or listener has to be placed in it with the purpose to intercept every frame it receives, keeping always in mind that it has to run at the client side, for the reasons widely discussed in this manuscript. As regard a performance improvement instead, we saw in Chapter 3, the test results of the *libpaillier* C library. Since they are not really satisfying for us, it must be considered an alternative for the test software development: the use of the CUDA[3] libraries. Actually, if a Nvidia graphic card is used (and if CUDA is supported by it) it is possible to exploit the GPUs' power of the card itself for the computations which are normally going only on the CPU. This could certainly be a good chance to see better performances on our tests and it should be definitely tried.

VDD has never been released as a package to be installed on Linux boxes, and this can be frustrating for those Linux users who want to try to use it for many other purposes, other than research and development. Actually VDD can be rather demanding to be installed, just for the reason that it is composed by several software components, joined together and separately configured. For these reasons, another future work on VDD could be the release of a complete Linux distribution where VDD already comes bundled, in order to dismiss the system administrators from the burden of installation and configuration.

Finally, in order to come back to Cryptography and especially to the Homomorphic Encryption for VDD, the research should go ahead to find out more efficient Encryption Systems with respect to the one tested and proposed in this thesis. Of course, a Fully Homomorphic Encryption which is implemented and usable for VDD is still the main target.

[3]Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by Nvidia. CUDA is the computing engine in Nvidia graphics processing units (GPUs) that is accessible to software developers through variants of industry standard programming languages. Further information at http://www.nvidia.com

Appendix A

Test algorithms

This appendix shows the algorithms used to perform the tests described in Chapter 3.

A.1 Decrypt files

Decrypt files - Main Module

Implements: Decrypt Files

Requires: libpaillier

procedure INIT
    String filename
    int KeySize ← size ▷ Size (bit) for the private key
    PubKey pubk ← paillier_keygen()
    PrivKey privk ← paillier_keygen()
    trigger get_file_in_blocks(filename) → blocks
    trigger parallel_encrypt(pubk,blocks)
    trigger parallel_decrypt(privk,encrypted_blocks) → decryption_time
    Output decryption_time
end procedure

procedure PARALLEL_ENCRYPT(*PubKey k*, *Blocks blocks*) ▷ The blocks are divided over the CPUs
    int intervals_matrix[N][2] ← create_intervals_matrix()
    for $i = 1$ to N **do**
        trigger parallel_encrypt_thread(intervals_matrix[i][0],intervals_matrix[i][1])
    end for
    for $i = 1$ to N **do**
        trigger thread_join(i) ▷ Wait for threads termination
    end for
end procedure

```
procedure PARALLEL_DECRYPT(PrivKey k, EncBlocks enc_blocks)
    int time ← 0
    int intervals_matrix[N][2] ← create_intervals_matrix()
    startTimer(time)
    for i = 1 to N do
        trigger parallel_decrypt_thread(intervals_matrix[i][0],intervals_matrix[i][1])
    end for
    for i = 1 to N do
        trigger thread_join(i)                          ▷ Wait for threads termination
    end for
    stopTimer(time)
    return time
end procedure

procedure PARALLEL_ENCRYPT_THREAD(int start, int stop)
    for i = start to stop do
        paillier_enc(blocks)
    end for
end procedure

procedure PARALLEL_DECRYPT_THREAD(int start, int stop)
    for i = start to stop do
        paillier_dec(blocks)
    end for
end procedure
```

A.2 Multiplication

The multiplication algorithm has got two versions: serial and multithread.

A.2.1 Serial Multiplication

Serial Multiplication - Main Module

Implements: Serial Multiplication

Requires: libpaillier

```
procedure INIT
    int CYCLES ← #Multiplications                ▷ Number of multiplications
    int KeySize ← size                           ▷ Size (bit) for the private key
```

```
    mpz_t integer ← 10^19                                   ▷ A big integer value
    EncInt cipher ← paillier_initialize_cipher()
    PubKey pubk ← paillier_keygen()
    PrivKey privk ← paillier_keygen()
    trigger multiply_mpz(integer,integer) → time_plain
    trigger multiply_enc(pubk,cipher,cipher) → time_cipher
    trigger free_resources()
    Output time_plain
    Output time_cipher
end procedure

procedure MULTIPLY_MPZ(PlainInt a, PlainInt b)              ▷ Plaintext multiplication
    int time ← 0
    startTimer(time)
    for i = 1 to CYCLES do
        mpz_mul(a,b)
    end for
    stopTimer(time)
    return time
end procedure

procedure MULTIPLY_ENC(PubKey k, EncInt a, EncInt b)        ▷ Ciphertext multiplication
    int time ← 0
    startTimer(time)
    for i = 1 to CYCLES do
        paillier_mul(k,a,b)
    end for
    stopTimer(time)
    return time
end procedure
```

A.2.2 Multithread Multiplication

Parallel Multiplication - Main Module

Implements: Parallel Multiplication

Requires: libpaillier

```
procedure INIT
    int KeySize ← size                                      ▷ Size (bit) for the private key
    mpz_t integer ← 10^19                                   ▷ A big integer value
    EncInt cipher ← paillier_initialize_cipher()
```

```
    PubKey pubk ← paillier_keygen()
    PrivKey privk ← paillier_keygen()
    int N ← get_nr_of_cpus()
    trigger parallel_multiply(integer,integer) → time_plain
    trigger parallel_enc_multiply(pubk,cipher,cipher) → time_cipher
    trigger free_resources()
    Output time_plain
    Output time_cipher
end procedure

procedure PARALLEL_MULTIPLY(PlainInt a, PlainInt b)
    int time ← 0
    int intervals_matrix[N][2] ← create_intervals_matrix()
    startTimer(time)
    for i = 1 to N do
        trigger parallel_multiply_thread(intervals_matrix[i][0],intervals_matrix[i][1])
    end for
    for i = 1 to N do
        trigger thread_join(i)                          ▷ Wait for threads termination
    end for
    stopTimer(time)
    return time
end procedure

procedure PARALLEL_MULTIPLY_THREAD(int start, int stop)
    for i = start to stop do
        mpz_mul(integer,integer)
    end for
end procedure

procedure PARALLEL_ENC_MULTIPLY(PlainInt a, PlainInt b)
    int time ← 0
    int intervals_matrix[N][2] ← create_intervals_matrix()
    startTimer(time)
    for i = 1 to N do
        trigger parallel_enc_multiply_thread(intervals_matrix[i][0],intervals_matrix[i][1])
    end for
    for i = 1 to N do
        trigger thread_join(i)                          ▷ Wait for threads termination
    end for
    stopTimer(time)
    return time
end procedure
```

procedure PARALLEL_ENC_MULTIPLY_THREAD($int\ start, int\ stop$)
  for $i = start$ to $stop$ **do**
    paillier_mul($integer$,$integer$)
  end for
end procedure

List of Figures

List of Tables

Bibliography

[1] Documentation on LVM. `http://www.vdd-project.org/`.

[2] TrueCrypt - Free open-source disk encryption software for Windows 7/Vista/XP, Mac OS X, and Linux. `http://www.truecrypt.org/`.

[3] R. Baldoni, F. Bertini, S. Cristofaro, and D. Lamanna. Virtual Distro Dispatcher: a light-weight Desktop-as-a-Service solution. In *First International Conference on Cloud Computing October 19 - 21, 2009, Munich, Germany, CLOUDCOMP 2009, ICST*, pages 247–260, 10 2009.

[4] R. Baldoni, F. Bertini, and D. Lamanna. Virtual Distro Dispatcher: A Costless Distributed Virtual Environment from Trashware. In *5th International Symposium on Parallel and Distributed Processing and Applications, ISPA 2007, LNCS, 223-234*, pages 223–234, 7 2007.

[5] Aldar C-F. Chan. Symmetric-key homomorphic encryption for encrypted data processing. In *Proceedings of the 2009 IEEE international conference on Communications*, ICC'09, pages 774–778, Piscataway, NJ, USA, 2009. IEEE Press.

[6] Soo-Chang Pei Chao-Yung Hsu, Chun-Shien Lu. Homomorphic encryption-based secure sift for privacy-preserving feature extraction. Technical report, 2010.

[7] Sabrina De Capitani di Vimercati, Sara Foresti, Sushil Jajodia, Stefano Paraboschi, Gerardo Pelosi, and Pierangela Samarati. Preserving confidentiality of security policies in data outsourcing. In *Proceedings of the 7th ACM workshop on Privacy in the electronic society*, WPES '08, pages 75–84, New York, NY, USA, 2008. ACM.

[8] Craig Gentry. *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. `crypto.stanford.edu/craig`.

[9] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of computing*, STOC '09, pages 169–178, New York, NY, USA, 2009. ACM.

[10] Craig Gentry. Computing arbitrary functions of encrypted data. *Commun. ACM*, 53:97–105, March 2010.

[11] Jibang Liu, Yung-Hsiang Lu and Cheng-Kok Koh. Performance Analysis of Arithmetic Operations in Homomorphic Encryption. 2010.

[12] Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network security: private communication in a public world, second edition*. Prentice Hall Press, Upper Saddle River, NJ, USA, second edition, 2002.

[13] D. D. Lamanna. *Performance monitoring and Progressive privacy in Virtual Distro Dispatcher, a Desktop-as-a-Service solution.* PhD thesis, Sapienza Università di Roma, 2011.

[14] Jibang Liu and Yung-Hsiang Lu. Energy savings in privacy-preserving computation offloading with protection by homomorphic encryption. In *Proceedings of the 2010 international conference on Power aware computing and systems*, HotPower'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[15] Tristan Richardson RealVNC Ltd. The RFB Protocol. Technical report, Olivetti Research Ltd / AT&T Labs Cambridge, 2010.

[16] D. Davide Lamanna R. Baldoni and Giorgia Lodi. How not to be seen in the cloud: a progressive privacy solution for desktop-as-a-service. 2011.

[17] D. Lamanna R. Baldoni and R. Russo. Distributed software platforms for rehabilitating obsolete hardware. In *First international conference on Open Source Systems (OSS2005) - Genova, Italy*, 7 2005.

[18] R. L. Rivest, L. Adleman and M. L. Dertouzos. On data banks and privacy homomorphisms. In *Foundations of Secure Computation, Academic Press*, pages 169–179, 1978.

[19] Ron Rivest. Voting, Homomorphic Encryption. Technical report, MIT Computer Science And Artificial Intelligence Laboratory, 2002.

[20] Simon Wardley, Etienne Goyer and Nick Barcet. *Ubuntu Enterprise Cloud Architecture*, 8 2009.