Insider Threat: Memory Confidentiality and Integrity in the Cloud

Francisco Rocha

School of Computing Science

Newcastle University

This dissertation is submitted for the degree of

Doctor of Philosophy

Newcastle upon Tyne, UK June 2015

I would like to dedicate this thesis to my parents, my sister, and my closest family and friends.

Acknowledgements

I would like to acknowledge my supervisor, Professor Aad van Moorsel, for giving me the opportunity to pursue my PhD studies at Newcastle University and for guiding me throughout this journey.

Abstract

The advantages of always available services, such as remote device backup or data storage, have helped the widespread adoption of cloud computing. However, cloud computing services challenge the traditional boundary between trusted inside and untrusted outside. A consumer's data and applications are no longer in premises, fundamentally changing the scope of an insider threat.

This thesis looks at the security risks associated with an insider threat. Specifically, we look into the critical challenge of assuring data confidentiality and integrity for the execution of arbitrary software in a consumer's virtual machine. The problem arises from having multiple virtual machines sharing hardware resources in the same physical host, while an administrator is granted elevated privileges over such host.

We used an empirical approach to collect evidence of the existence of this security problem and implemented a prototype of a novel prevention mechanism for such a problem. Finally, we propose a trustworthy cloud architecture which uses the security properties our prevention mechanism guarantees as a building block.

To collect the evidence required to demonstrate how an insider threat can become a security problem to a cloud computing infrastructure, we performed a set of attacks targeting the three most commonly used virtualization software solutions. These attacks attempt to compromise data confidentiality and integrity of cloud consumers' data. The prototype to evaluate our novel prevention mechanism was implemented in the Xen hypervisor and tested against known attacks.

The prototype we implemented focuses on applying restrictions to the permissive memory access model currently in use in the most relevant virtualization software solutions. We envision the use of a mandatory memory access control model in the virtualization software. This model enforces the principle of least privilege to memory access, which means cloud administrators are assigned with only enough privileges to successfully perform their administrative tasks.

Although the changes we suggest to the virtualization layer make it more restrictive, our solution is versatile enough to port all the functionality available in current virtualization solutions. Therefore, our trustworthy cloud architecture guarantees data confidentiality and integrity and achieves a more transparent trustworthy cloud ecosystem while preserving functionality.

Our results show that a malicious insider can compromise security sensitive data in the three most important commercial virtualization software solutions. These virtualization solutions are publicly available and the number of cloud servers using these solutions accounts for the majority of the virtualization market. The prevention mechanism prototype we designed and implemented guarantees data confidentiality and integrity against such attacks and reduces the trusted computing base of the virtualization layer. These results indicate how current virtualization solutions need to reconsider their view on insider threats.

Contents

List of Figures

Chapter 1

Introduction

Cloud computing offers multiple advantages such as a pay-per-use cost model and easy access to an apparently unlimited pool of computational resources. These features are appealing for small and midsize companies looking for a versatile solution to expand their businesses without a significant financial commitment. However, cloud computing is also susceptible to new security threats such as abuse and nefarious use (e.g., criminal activities) of cloud resources and malicious insiders. The latter is the focus of the work introduced in this thesis [1].

The two major characteristics of cloud computing making it vulnerable to the type of attacks discussed in this document are the use of virtualization technology, and the offloading of consumer data to an off-premises cloud provider's owned and managed remote infrastructure. The latter changes the traditional boundary between trusted inside and untrusted outside.

In a cloud computing environment virtual machines from different consumers can co-exist in the same physical server, while cloud administrators are granted elevated access privileges over virtualization administration software, e.g., a management virtual machine. These characteristics combined with a security design flaw and the offloading of consumer data to the cloud provider's infrastructure make it possible for a malicious insider to compromise security sensitive data that belongs to cloud consumers [73, 74].

An ideal cloud computing environment should permit cloud administrators to perform the necessary tasks to keep the whole cloud ecosystem operational, while at the same time guarantee that those administrators cannot compromise consumers' data. Unfortunately, current solutions do not enforce the principle of least privilege giving a malicious administrator the ability to compromise security sensitive data [73, 74]. The security problem identified in this thesis affects the latest versions of the three major virtualization software solutions used in the cloud industry. Therefore, it is of paramount importance to perform this research so virtualization software developers become aware of such risks.

Considering this information we define the problem statement for this thesis to be designing a secure cloud computing architecture that enforces the principle of least privilege and guarantees that cloud administrators can perform the necessary tasks to keep the system operational.

1.1 Aim and Objectives

The aim of our work, as established in our problem statement, is to design a cloud architecture that prevents insider threats. Although this type of threat has been overlooked in the research community, its impact is quite severe since malicious insiders, denial of services, and malicious code account for more than 55% of the cost of cyber crime in the United States [35].

Our strategy to achieve such goal consists in first exposing the insider threat as a relevant security problem which is transversal to the most commonly used virtualization software solutions. After exposing the seriousness of such security problem we introduce our prevention mechanism and the role such mechanism plays in a more secure cloud architecture.

This thesis contains the context and details of a set of objectives we defined as necessary to present our final solution. These objectives provide a clear definition of the scope of our work, demonstrate the security problem we studied, and present our proposed solution, results, and its evaluation. This document should help the reader understand our motivation for preventing insider threats, the reasons supporting the design decisions for our solution, and how the suggested solution improves on current state of the art. Therefore, this document focuses on background and literature review, elaborating on how the security flaw impacts memory confidentiality and integrity, and discussing our solution and its applicability.

The background and literature review content supplies information on key subjects such as cloud computing and security, and an analyses of current state of the art in cloud security. This analysis focuses on how effectively cloud security prevents insider threats. These chapters should provide a clear view of the scope of our research. The concepts and ideas explained in these sections are paramount for understanding the chapters that follow.

After the background and literature review chapters, we introduce and explain the relevant security design flaw in detail and demonstrate how a malicious insider can exploit such flaw to compromise cloud consumers' security sensitive data. Another important point in this chapter is showing how this security problem is not just an implementation issue affecting a particular virtual machine monitor. The content of this chapter also introduces advanced real time attacks a malicious insider can perform to compromise the confidentiality and integrity of consumers' data.

The remainder of the thesis centres on describing our solution in detail, discussing how we implemented and evaluated its effectiveness, and how taking advantage of such solution can improve the security of current approaches.

Our solution introduces a mandatory memory access control mechanism into the virtualization software, or virtual machine monitor. This mechanism performs access control in the virtualization layer guaranteeing that a malicious insider with administrative privileges over a virtualization server cannot compromise the confidentiality and integrity of consumers' security sensitive data.

This approach restricts the access of an insider to the amount of privileges required to conclude administrative tasks, i.e., it enforces the principle of least privilege. Using the virtualization software as the access control enforcer also assures that the trusted computing base of our approach is smaller when compared to previous solutions. A simple definition for trusted computing base is the set of system components we implicitly trust to behave correctly.

1.2 Thesis Contributions

This thesis contains a few contributions related to insider threats in cloud computing.

- A thorough literature review of the current state of the art in cloud computing security. Our analysis centres on how effective current solutions are at preventing a malicious insider from compromising memory confidentiality and integrity and consequentially consumers data. The insights from this review led us to identifying the research gap we could target for this work.

- Successfully compromised security sensitive data in three major virtualization software solutions demonstrating their vulnerability to malicious insiders. The success of these attacks proves the existence of a specific security design flaw. Performing these attacks improved our perception of the problem which ultimately helped us with devising our final solution.

- Our main contribution is the design, implementation, and testing of a novel prevention mechanism that enforces the principle of least privilege to protect against insider threats. This approach enforces the principle of least privilege, and guarantees memory confidentiality and integrity for the execution of arbitrary software in a consumer virtual machine hosted in a cloud computing environment.

- Proposal of a trustworthy cloud computing architecture. This architecture takes advantage of the security properties guaranteed through the use of our prevention mechanism. Although a few changes are required, this architecture does not prevent a cloud administrator from performing his administrative tasks. The novelty of our approach is designing the architecture so its security sensitive operations become more transparent to cloud consumers.

1.3 Publication History

The contributions in this thesis were first introduced in published peer-reviewed work written or co-written by the author. Those publications are properly referenced throughout this document. A list of the relevant publications follows.

- Rocha, F. (2015). Insider Threat: Memory Confidentiality and Integrity in the Cloud, Lambert Academic Publishing (LAP). (invited publication)
- Rocha, F. and van Moorsel, A. (2015). Protecting against malicious insiders in cloud computing: Survey and research challenges. (to be submitted)
- Rocha, F., Abreu, S., and Correia, M. (2013a). *The Next Frontier: Managing Data Confidentiality and Integrity in the Cloud.* IEEE Computer Society Press.
- Rocha, F., Gross, T., and van Moorsel, A. (2013). Defense-in-depth against malicious insiders in the cloud. In *IEEE International Conference on Cloud Engineering 2013*, San Francisco, USA.
- Rocha, F., Abreu, S., and Correia, M. (2011). The final frontier: Confidentiality and privacy in the cloud. *Computer*, 44:44–50.
- Rocha, F. and Correia, M. (2011). Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, DSNW '11, Hong Kong.

The remainder of this thesis is organized as follows. Chapter 2 provides the background context with the key concepts required to understand later discussions in the document. A literature review of related research work is included in Chapter 3. This chapter identifies the research gap for our work and how it relates to similar research work. Chapter 4 introduces successful attacks performed against major virtualization software. In Chapter 5 our approach to preventing the malicious insider threat is explained and its advantages and disadvantages are also discussed. Chapter 6 proposes a trustworthy cloud architecture that builds on our prevention mechanism to make the cloud ecosystem more transparent and secure. The final chapter presents our conclusions and suggestions for future work.

Chapter 2

Background

This chapter introduces the required concepts from the key main subjects used in the remainder of the thesis. The definitions needed to understand our explanations are included in this chapter. More specific concepts and technologies are introduced in the relevant sections.

Security related publications typically use Alice and Bob as communicating parties as a way of making the explanation of a complex subject easier to follow. These two individuals were introduced in the publication that defined the RSA algorithm for public-key cryptography [70]. Alice and Bob are the legitimate participants. For malicious actions we chose to use Mallory, an active adversary that tries to subvert the security mechanisms Alice and Bob use to keep their data secure [82].

The remainder of this chapter is organised as follows. It starts with an introduction to the necessary security principles such as basic security requirements and approaches to fulfil them. Next, it provides a short overview of relevant cryptographic primitives such as secure hash functions. The two sections that follow define the key concepts of virtualization and cloud computing. The final section contains a short introduction to key concepts of trustworthy, or trusted, computing technology.

2.1 Security Principles

This first section is used to introduce the basic requirements and design principles a system should respect in order to the protect the information resident within it.

2.1.1 Basic Requirements

When considering the security of a system the basic requirements to look for are *confidentiality*, *integrity*, and *availability*. There are several definitions for these requirements but we decided to use the definitions established by the National Institute of Standards and Technology (NIST) [60].

Confidentiality is the requirement that private or confidential information not be disclosed to unauthorized individuals. There are several scenarios where it is necessary to keep information secure. For example, the motivation behind the first formal work in computer security was the military need to implement the "need-to-know" principle [11].

The requirement of *integrity* can be subdivided in *data* and *system integrity*. Some authors also consider *origin integrity*, which is typically known as *authentication* [11]. For the purposes of this document we are going to focus on the first two.

Data integrity states that modification of information and programs must only happen in a specified and authorized manner. For example, consider a scenario where Alice requests a funds transfer to Bob with a value of one hundred pounds sterling. If Mallory is capable of compromising the data integrity of such transaction, she could manipulate the bank to process the transfer to her account instead of Bob's account.

Guaranteeing *system integrity* means that the system must perform the expected function in an unaltered fashion, free from unintended or deliberate unauthorized manipulation. A famous example of inadvertent system failures was when an Ariane 5 rocket launch system exploded due to a software error just forty seconds after it had initiated its flight. The project had a $7 billion development cost and the destroyed rocket and its cargo were valued at $500 millon [22].

The final requirement is *availability*. A system satisfies the requirement of availability when its legitimate users are able to access the desired data or resource. If for some reason the system is unresponsive to its legitimate users, its availability is considered compromised. A well-known malicious attempt against a system's availability is a denial of service (DoS) attack. This type of attack consists in an adversary flooding a system with illegitimate requests in order to consume resources. Hence, preventing the system from handling legitimate requests.

2.1.2 Design

An accepted set of security design principles was established in the early days of computer security. Since researchers could not find a complete method to eliminate all security flaws from large scale general purpose systems, those principles were devised based on lessons learnt from designing systems with security requirements. The list of principles includes, *economy of mechanism*, *fail-safe defaults*, *complete mediation*, *open design*, *separation of privilege*, *least privilege*, *least common mechanism*, and *psychological acceptability* [78]. For our work the most relevant principles are:

- Economy of Mechanism
- Complete Mediation
- Least Privilege

In this chapter we include definitions for these principles. In later chapters, when the context is appropriate, we explain in detail why and how these principles are the most relevant to our work.

The principle of *economy of mechanism* states that the design should be kept as simple and small as possible. This principle is relevant when considering the security of a system because a more complex and larger system means a wider attack surface for those wanting to break the security of the system [78]. The importance of this principle is also noticeable for mechanisms such as software source code inspection, whose success depends on a system being small and simple.

Another principle to consider is the principle of *least privilege*. This principle states that every program and user of the system should only be given the necessary privileges to complete a task. The primary objective of this principle is to limit the damage that can occur from unintentional, unwanted, or improper use of privilege [78]. In Chapter 4, we demonstrate how current virtualization solutions not respecting the principle of least privilege puts consumers' data at risk.

The requirement that any access to a system object be checked for authorization is the core of the principle of *complete mediation* [78]. The correct application of this principle implies a system-wide view of access control, which means that it should be applied to initialization, normal operation, recovery, shutdown, and maintenance. This wide influence makes this principle a foundation to any protection mechanism of a system. Therefore, any protection mechanism should deploy methods of identifying the source of all requests.

Throughout this document the notion of economy of mechanism is going to be associated with the concept of *trusted computing base*, which is why we decided to introduce the latter at this point. We chose to use the definition of trusted computing base provided by the Department of Defence (DoD). The DoD defines trusted computing base (TCB) to be the totality of protection mechanisms within a computer system, including hardware, software and firmware, the combination of which is responsible for enforcing a security policy [19].

Considering the principle of economy of mechanism and the definition of trusted computing base, we conclude that a system with a smaller trusted computing base is more secure, i.e., respects the principle of economy of mechanism. This idea is very important for later when discussing the advantages of our solution when compared with similar research.

These three principles are fundamental when you want to develop a trustworthy system. The *economy of mechanism* principle guarantees a reduced *trusted computing base* by design which is paramount for implementing trustworthy software modules. Assuring a software module is not accessing unnecessary functionality/data through *the principle of least privilege* creates an ecosystem where trustworthy software modules can interact with each other whilst guaranteeing strong security properties. Finally, enforcing *complete mediation* in a secure system guarantees unwanted accesses can be prevented and logged when applicable.

2.1.3 Approaches

Researchers have defined four classes of security approaches to enforce the basic security requirements in a system. Those approaches are security through *prevention*, *detection and recovery*, *resilience*, or *deterrence* [66].

Prevention solutions require detailed knowledge of a security problem at design time. This knowledge makes it possible to devise mechanisms to harden a system's security against a known threat. For example, a prevention mechanism meant to guarantee the integrity of data is going to operate in a way that prevents unauthorized users from changing that data.

A *detection and recovery* method centres on monitoring the behaviour of every program and user of a system. This monitoring permits the detection of unauthorized behaviour, which then allows the elimination of the source of malicious behaviour and consequent restoration of normal system functionality. If we consider the integrity example once more. When a detection and recovery mechanism is deployed, it is not going to try to prevent violations of data integrity, but instead report when data integrity is no longer trustworthy.

Security mechanisms to assure *resilience* aim to guarantee graceful performance degradation in an already compromised system. The assumption is that systems can contain compromised nodes and hosts. For example, if a compromised node performs unauthorized changes to data, violating its integrity, a resilience mechanism should guarantee that such violation has the minimum impact possible in the whole system.

The last of the four approaches in our list is non-technical. *Deterrence* mechanisms consist in providing legal disincentives, which should reduce the percentage of attacks against a system. A simple example would be computer crime law that applies heavy penalties to the exploitation of vulnerabilities in computer systems.

These definitions are relevant to our research so we can isolate the area our security mechanism falls into. Without disregarding the importance of other security research areas. Our work focuses on technological prevention mechanisms. The structure of this thesis highlights it perfectly. We find and understand in detail a security problem which allows us to design and prototype a prevention mechanism for such problem.

2.2 Cryptography

In this section we introduce the cryptographic primitives that are important for later discussions. The content of this section does not intend to provide an extensive introduction to cryptography.

2.2.1 Hash Function

The aim of a one-way hash function is to generate a *fingerprint* of a block of data, e.g., a file or a message. When communicating parties wish to verify if the content of a message has not changed and that it comes from the alleged source, they use a mechanism for message authentication. Hash functions are widely used in message authentication. Let us consider a hash function H, and list the properties it needs to satisfy in order to be practical in the process of achieving message authentication [87].

1. H needs to handle blocks of data of any length.
2. H's output must have a fixed-size.
3. Hardware and software implementations must be practical. Therefore, for any given x, it is relatively easy to compute $H(x)$.
4. A hash function is referred to as *one-way* if, for any given hash code h, it is computationally infeasible to find the x in $H(x) = h$.

5. For a hash function to be considered *weak collision resistant*, it must satisfy the following requirement: For any given block of data x, it needs to be computationally infeasible to find a block of data y not equal to x such that $H(y) = H(x)$.

6. A *strong collision resistant* hash function fulfils the requirement of being computationally infeasible to find any pair (x, y) such that $H(x) = H(y)$.

Properties from one to three are basic requirements for a hash function to be used in message authentication. The fourth property adds the one-way attribute to a hash function, which guarantees that computing a hash code is easy but, given a hash code, it is virtually impossible to reach the original message. This attribute makes a hash code useful for secret-based user authentication because the secret does not need to be send. If this property is not valid, an attacker can easily invert a hash function, and from the hash code obtain the original message.

Satisfying the fifth property turns a hash function weak collision resistant assuring that for such function it is impossible to find an alternative message with the same hash code as a known message. This property prevents forgery when using an enciphered hash code. Assuming this property is not true and the message is not encrypted. An attacker could intercept a message plus its enciphered hash code, use the message to generate a deciphered hash code, and finally compute an alternative message for the same hash code.

The strong collision resistant property serves to protect against the sophisticated birthday attack. This attack reduces the strength of a n-bit hash function from 2^n to $2^{n/2}$ [86].

A hash code is as useful for message authentication as it is to verify data integrity. Since it is computationally infeasible to find a $y \neq x$ such that $H(x) = H(y)$, any changes to the bits of the block of data x, result in a different hash code. This difference can be used to verify if a message's data integrity was compromised. This property is paramount for later discussions.

A hash function can be as simple as performing a bit-by-bit exclusive-or of every data block. However, to satisfy the properties previously enumerated we need a secure hash function. The Secure Hash Algorithm (SHA) was developed by NIST and is the most widely used secure hash function. Its hash codes can have 160, 256, 384, or 512 bits of length [87].

2.2.2 Public-Key Cryptography

The origins of public-key cryptography date back to 1976 when Diffie and Hellman introduced their revolutionary idea [21]. When compared with traditional symmetric encryption algorithms, a public-key algorithm uses mathematical operations rather than relying on operations over bit patterns. Moreover, public-key cryptography is asymmetric, which means it uses two distinct keys whereas symmetric encryption uses the same secret key for encryption and decryption.

The assumption about an adversary trying to break the security of an encryption scheme is that she can obtain access to all encrypted data that gets transmitted, and has every detail about the encryption/decryption algorithm [45]. Therefore, the security of any cryptosystem depends on the size of the key, and the quality of the algorithm or how much computational work is required to break a cipher.

The two distinct keys part of a public-key cryptosystem are referred to as public key (PU) and private key (PR). These keys are usually generated by their owner. The owner of a private key should always keep it secret and secure. The public key must be published in a key directory or public file containing a set of such keys. This key distribution centre is where individuals can collect public keys for the entities they wish to contact.

Diffie and Hellman defined the set of requirements a public-key cryptography algorithm needs to satisfy [21, 87]. The list that follows enumerates those requirements. From this point forward, the encryption and decryption procedures are denoted as E and D, respectively.

1. Generating a private-public key (PR_{Bob} and PU_{Bob}, respectively) pair must be computationally easy for Bob.

2. It is computationally easy for Alice to obtain the ciphertext, C, for message M when she knows Bob's public key: $C = E\left(PU_{Bob}, M\right)$.

3. Using his private key, it is computationally easy for Bob to retrieve the original message M: $M = D\left(PR_{Bob}, C\right) = D\left[PR_{Bob}, E\left(PU_{Bob}, M\right)\right]$

4. It is computationally infeasible for Mallory to derive the private key from the public key.

5. Assuming Mallory has Bob's public key and a ciphertext Alice encrypted for Bob. It is computationally infeasible for Mallory to recover the original message.

6. Any key of the private-public key pair can be used for encryption, with its pair used for decryption. $M = D\left[PU_{Bob}, E\left(PR_{Bob}, M\right)\right] = D\left[PR_{Bob}, E\left(PU_{Bob}, M\right)\right]$

The three applications of public-key cryptography are encryption and decryption of data, digital signatures, and key exchange. An algorithm that supports these three applications is the well-known RSA [70].

The first application is encryption and decryption of data. This application is similar to what two communicating parties can achieve when using symmetric encryption algorithms without the need for a shared secret key. An example of this application follows.

Let us consider a scenario where Alice needs to send a private message to Bob. Assuming Alice and Bob have generated their public and private keys, and registered their public keys in a public directory. The set of steps involved in Bob receiving an enciphered message from Alice and deciphering it are described below.

1. Alice communicates with the public repository where Bob's public key (PU_{Bob}) was registered and obtains her own copy. This copy remains in Alice's key ring for future communications with Bob.

2. Alice wishes to send message M to Bob. To obtain a ciphertext, C, of message M, Alice uses Bob's public key and performs the operation $C = E(PU_{Bob}, M)$. Finally, she forwards cipher C to Bob.

3. Bob receives the ciphertext, C, and uses his private key (PR_{Bob}) to retrieve M through the operation $M = D(PR_{Bob}, C)$. Bob is the only one that can perform this decryption step because only he has access to his private key.

This scenario illustrates how public-key cryptography can be used for exchanging confidential data. The sender only needs the recipient's public key to encrypt a message. The security of this approach can be compromised if Mallory is capable of obtaining Bob's private key. A new public-private key pair can be generated at any point as long as the old public key is revoked and replaced with the new one. This creates key management challenges in public-key cryptography systems.

Digital signatures is another application. For Bob to generate a digital signature he needs to supply two arguments to the algorithm. The first argument is Bob's private key and the second is the whole message or a small block of data that is a function (e.g., a hash code) of the message. The algorithm then *signs* (i.e., encrypts) the block of data with Bob's private key. Alice can then use Bob's public key to decrypt the signed message which guarantees her that the message came from Bob. This is valid as long as Bob is the only one with access to his private key.

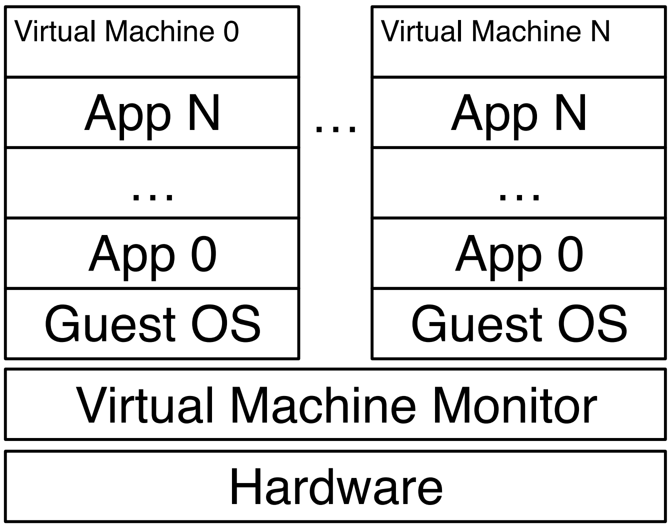


Fig. 2.1 Native Virtualization.

The final application is key exchange. The Diffie-Hellman approach is probably the most famous key exchange method [21]. Key exchange algorithms are a complex area on its own which is not in the scope of this thesis. Since Diffie-Hellman is a complex algorithm, and not necessary for understanding the content of this document, we only provide a simple way of using public-key cryptography for exchanging a secret key. Let us imagine that Alice needs to send a secret key to Bob. Alice can sign a secret key with her private key, encrypt it with Bob's public key, and send it to Bob. This method is enough to guarantee a secure key exchange method between Alice and Bob.

2.3 Virtualization

Virtualization is a main technological foundation for cloud computing. This section introduces the key concepts of virtualization, the expected security properties in a virtualization environment, and some security applications of virtualization.

The concept of virtual machine systems was introduced as a solution to the limitations of the extended machine concept [28]. This concept was commonplace in 70s third generation architectures and multi-programming operating systems such as OS/360 [18]. An extended machine includes a set of non-privileged instructions combined with a set of system/supervisory calls, with no access to the privileged instructions set. Which means that, without the right privileges, the previous two sets are not enough to replicate a bare metal machine. Therefore, one of the main limitations in this approach was that it did not support multiple operating systems on the same physical machine [28].

In virtual machine systems there are two main components, which are *virtual machines (VMs)* and the *virtual machine monitor (VMM)*. The term *virtual machine* refers to the simulated machine whereas the *virtual machine monitor* is the simulator software. The latter is the innovation in virtual machine systems which overcomes the limitations in the extended machine approach. The VMM layer transforms the single machine interface into many [28]. This means that each virtual machine is a replica of a computer system with every instructions set and the required system resources.

The architecture in Figure 2.1 is an illustration of the classic virtualization approach IBM introduced in the 70s [28]. The figure also shows that every virtual machine can execute its own guest operating system (OS), which can then execute its own applications. In current literature the term *full virtualization* is used to refer to an environment like the one shown in Figure 2.1. The virtual machine monitor can also be referred to as hypervisor.

This particular full virtualization architecture is known as *native*, or *bare metal*, virtualization because the virtual machine monitor interfaces directly with the bare metal machine. The VMM controls the flow of instructions between virtual machines and the physical hardware (e.g., CPU and memory) [81].

For purposes of the implementation discussed in this thesis, we are interested in the virtualization of the x86 platform. The virtualization of this platform started with VMWare back in the 90s [102]. The three virtualization approaches for the x86 platform are full virtualization through *binary translation*, *paravirtualization*, and *hardware assisted virtualization*. A brief introduction to each of these techniques follows.

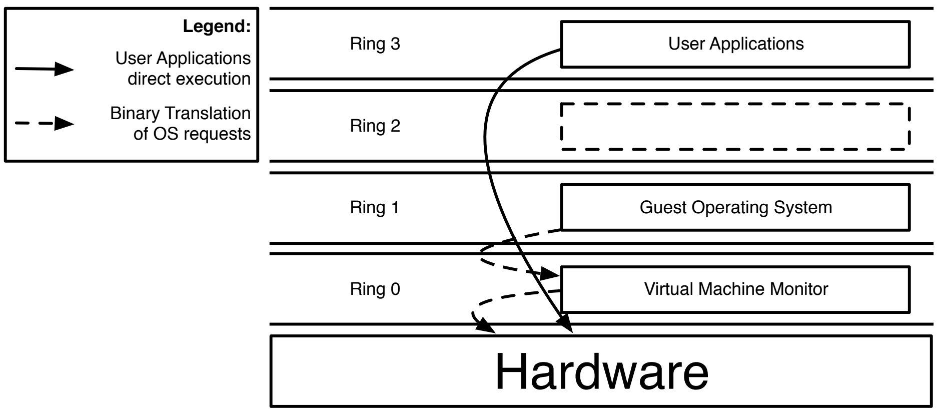


Fig. 2.2 Binary Translation.

2.3.1 Full Virtualization - Binary Translation

Full virtualization through binary translation was the first virtualization technique for the x86 platform introduced by VMWare in 1998 [102]. VMWare solved the problem of virtualizing sensitive x86 instructions that, at the time, were believed impossible to virtualize. Their approach is a combination of binary translation and direct execution.

Figure 2.2 depicts the binary translation of sensitive x86 instructions and the direct execution of *user applications*, i.e., non-privileged user-level instructions. The kernel code of the *guest operating system* contains nonvirtualizable x86 instructions. The binary translation procedure consists in translating that kernel code to replace the nonvirtualizable x86 instructions with new instructions code that the *virtual machine monitor* understands, and from which it can generate the desired requests to the virtual hardware. User applications have permission to execute their non-privileged user-level instructions directly on the physical hardware to assure high throughput performance.

The combination of binary translation and direct execution guarantees full virtualization for the guest operating systems executing in virtual machines. This means that the operating system does not need any modifications and it is not aware that it is executing on top of a virtualization layer. There is a dedicated virtual machine monitor for each virtual machine. The VMM is responsible for providing the guest operating system of its VM with the physical machine resources it expects, which include a virtual Basic Input/Output System (BIOS), virtual devices, and virtual memory management.

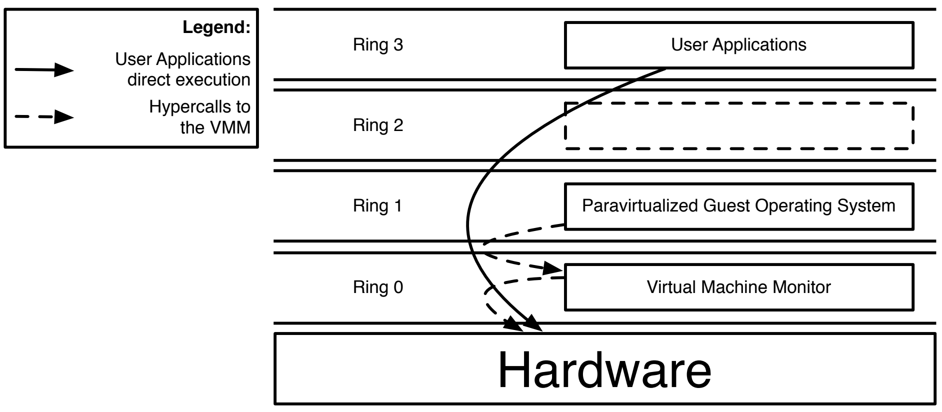


Fig. 2.3 Paravirtualization.

2.3.2 Paravirtualization

Paravirtualization introduces a new technique to handle the problem of the nonvirtualizable privileged x86 instructions. This method relies on co-design of virtual machine monitor and operating system [104]. The former offers an interface slightly different from the physical machine. The latter is modified to interact with the virtual machine monitor so the set of privileged x86 instructions can be virtualized.

Figure 2.3 illustrates the handling of instructions in a paravirtualizion environment. *User applications* continue to execute their non-privileged instructions set natively for high performance while the privileged OS-level instructions from the *paravirtualized guest operating system* generate an *hypercall*. Hypercalls are similar to the traditional system calls user applications use in nonvirtualized environments to request privileged operations from the kernel. In an hypercall, however, it is the operating system requesting a privileged operation to the *virtual machine monitor*.

The paravirtualization approach enhances performance, scalability, and simplicity [104]. Another advantage is that it does not require hardware-assisted virtualization technology (discussed below), but it has its own disadvantages. For example, a guest operating system can only execute in a paravirtualized environment if its kernel is modified so it uses hypercalls to interact with the virtual machine monitor. These changes increase maintenance costs and prevent off the shelf operating systems from executing in paravirtualized platforms.

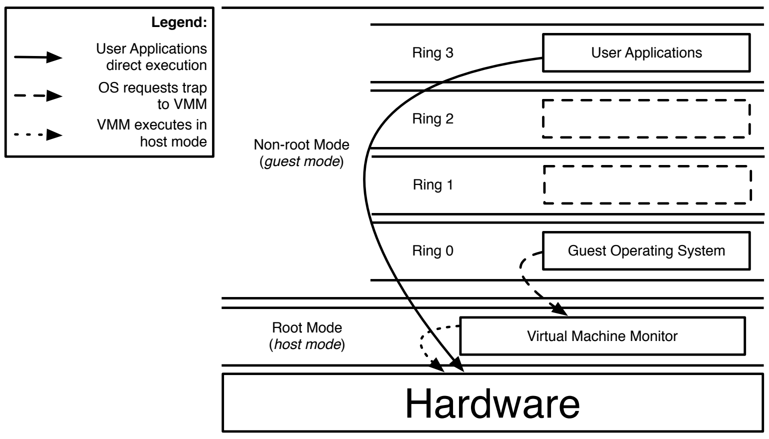


Fig. 2.4 Hardware-Assisted Virtualization.

2.3.3 Hardware-Assisted Virtualization

Binary translation and paravirtualization were envisioned to solve the problem of virtualizing privileged x86 instructions due to rigid architecture requirements. The x86 architecture provides four levels of privilege through rings 0, 1, 2, and 3. The problem arises from moving the OS away from ring 0 so the VMM can take its place as the software with highest privilege execution level. Since the architecture was designed for a single operating system when this change happens the privilege OS instructions do not behave as designed because they are not executed in ring 0 any more. Therefore, an architecture redesign considering virtualization requirements would improve its support.

Hardware-assisted virtualization is the redesign of the x86 architecture to better accommodate the requirements of virtual machine systems. Figure 2.4 shows the existence of hardware support for a new root privilege level (*root mode*) where the *virtual machine monitor* can execute at a higher privilege level than ring 0. In this architecture, when a virtual machine monitor is present, privileged x86 instructions from the *guest operating system* are set to automatically transfer platform control to the virtual machine monitor. This feature removes the need for binary translation or paravirtualization.

Examples of hardware-based virtualization are Intel's virtualization technology (Intel VT) and AMD's virtualization technology (AMD-V) [2, 36]. The set of features includes fast transfer of platform control from guest operating systems to virtual machine monitor, and the ability to uniquely assign physical Input/Output devices to a particular guest operating system.

2.3.4 Guest OS Isolation

The virtual machine monitor is responsible for partitioning the platform's physical and logical resources among the virtual machines [81]. Disk drives and network interface cards are examples of physical resources available for partitioning. When a hypervisor is performing *physical partitioning*, it assigns a separate physical resource to a particular virtual machine. In *logical partitioning*, a set of resources available in one or more physical hosts is divided among the virtual machines. This type of partitioning allows the VMs to share physical resources such as processor and random access memory.

The responsibility of partitioning the resources implies that the hypervisor is also in charge of keeping those resources properly isolated. Guaranteeing isolation between guest operating systems, restricting which resources they can access, and managing their privilege level is widely known as *sandboxing* [81]. The sandboxing of guest operating systems has security and reliability benefits. The security advantages are obviously assuring that a malicious guest OS cannot perform unauthorized accesses to the resources of other guest operating systems. The isolation between guest OSs guarantees that a crash or denial of service in a guest OS is confined to its resources. Hence, such events do not affect the whole system improving overall reliability.

Although sandboxing of guest operating systems improves security and reliability, a virtualization environment might still be vulnerable to *side-channel* or *escape* attacks [81]. An example of a side-channel attack is having a guest OS exploiting information patterns for CPU usage to extract cryptographic keys [47]. A guest OS trying to break out of sandboxing to escalate its privileges to the hypervisor level is an example of an escape attack. Succeeding in such attack guarantees an adversary complete control over all the guest OSs executing in the compromised physical host.

2.3.5 Virtual Machine Introspection

Since the virtual machine monitor executes at the highest level of privilege in a virtualization architecture, it has complete access to the resources assigned to each virtual machine executing on top of it. Therefore, the virtual machine monitor has the right permissions to monitor the guest operating systems executing in virtual machines. Externally inspecting the runtime state of a virtual machine is known as *virtual machine instrospection* (VMI) [92].

The monitoring software can be located within a virtual machine or the virtual machine monitor. Processor registers and memory are examples of collectible data through virtual machine introspection techniques. This data can then be forwarded to external security controls to be used in intrusion detection or other security operations [81].

2.4 Cloud Computing

Cloud computing is the platform under scrutiny in this thesis. We decided to adopt NIST's definition of cloud computing, which reads as follows: "Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." [57]. The concept of *utility computing* introduced with the cloud demarcates cloud computing from previous distributed systems models [3].

The remainder of this section provides key definitions for cloud entities, essential characteristics, and service and deployment models.

2.4.1 Entities

The two main entities present in a cloud computing environment are the *cloud provider* and the *cloud consumer*. Throughout this document these entities are also referred to as provider or consumer, omitting the cloud prefix. The cloud provider is the owner of the physical cloud infrastructure, which offers the shared pool of computing resources (e.g., storage and services) to cloud consumers. The cloud consumer can be an individual or a company that takes advantage of the resources supplied by cloud providers.

2.4.2 Essential Characteristics

A set of essential characteristics distinguish cloud computing from traditional distributed systems. NIST has identified *on-demand self-service*, *broad network access*, *resource pooling*, *rapid elasticity*, and *measured service* as the most relevant features of cloud computing.

- **On-Demand Self-Service:** The automatic provisioning of computing resources, such as server time and network storage, do not require any human interaction with each of the service providers and can be performed as needed.

- **Broad Network Access:** The standardized access mechanisms and networked access to computing resources promote the use of variable thin or thick client platforms (e.g., tablet, laptop, or smartphones).

- **Resource Pooling:** The computing resources available to consumers are setup in a pool used to distribute them among multiple different consumers. The assignment or reassignment of either physical or virtual resources is done in a dynamic manner in order to satisfy consumer demand. Although the sense of resource location is lost in cloud computing because consumers are not aware of the physical location of where their share was allocated, it is possible to specify the desired location at a higher level (e.g., country). The term computing resources refers to, for example, network bandwidth or data storage.

- **Rapid Elasticity:** A consumer's capabilities demand is satisfied through elastic resource provision and release. Using this approach makes it look like an unlimited pool of resources is available to the consumer. From which, capabilities can be acquired and released at any time.

- **Measured Service:** A cloud system must be capable of automatically measuring resource usage at the appropriate level of abstraction for each type of service (e.g., storage and bandwidth). This capability allows for resource usage monitoring, control, and reporting, offering both the provider and consumer a good level of transparency.

2.4.3 Deployment Models

A cloud system can be deployed according to four different models as defined by NIST. The models include *private cloud*, *community cloud*, *public cloud*, and *hybrid cloud*. The list that follows provides a description of each model, offering an extra analysis of key advantages and drawbacks relevant to our work. The different models are sorted from the one with the highest to the one with the least number of security concerns. The hybrid model is left for last because its security depends on the weakest model involved.

- **Public Cloud:** This is the deployment model that comes to mind when the word cloud is mentioned. This model is provisioned for use by the general public. The owning entity responsible for its management, and operation, can be a business, academic, or government organization, or some combination of the previous. The location of the cloud infrastructure is on the premises of the cloud provider. The major benefit of this model is the low investment required by consumers to kick start a cloud-based service offering. However, the open use philosophy of a public cloud brings additional security risks for both consumers and providers.

- **Community Cloud:** An infrastructure created for a set of consumers from organizations with shared objectives (e.g., mission or security requirements). The owner, manager, and operator of the infrastructure can be any of the participating organizations, a third party, or some combination of the previous. The location of such infrastructure can be on or off premises. The advantage of this model is reduced costs when compared with private clouds but sharing the infrastructure introduces some security concerns for the consumers.

- **Private Cloud:** In this model the cloud infrastructure is deployed for use by a single organization with several consumers (e.g., different business units). The owner of the infrastructure can be the organization using it, a third party, or a combination of both, and its location can be on or off premises. The disadvantage of this approach is the higher cost when compared to other approaches, e.g., maintenance and initial investment costs. However, not sharing the infrastructure with other organizations is beneficial to data security.

- **Hybrid Cloud:** This deployment model consists in a combination of two or more distinct cloud infrastructures (public, community, or private) that remain separate entities, but are connected through standardized or proprietary technologies that enable data and application portability (e.g., cloud bursting for load balancing between clouds). In this deployment model the clear advantage is the standardization of processes between multiple cloud infrastructures. Although cost and security vary from cloud to cloud, the security is always dependent on the low hanging fruit which in this case is the presence of a public cloud among the set of cloud infrastructures.

2.4.4 Service Models

According to NIST there are three service models a cloud infrastructure can implement, *Infrastructure as a service (IaaS)*, *Platform as a Service (PaaS)*, and *Software as a Service (SaaS)*. A detailed description of each model follows. The service models are ordered with respect to the level of control (highest to lowest) the consumer has over the resources offered to him on the cloud infrastructure.

- **Infrastructure as a Service (IaaS):** The cloud provider supplies the consumer with fundamental computing resources such as processing, storage, and networks. The consumer has complete control over the provisioned resources where he can deploy and run arbitrary software, including operating systems and applications. The control of the underlying physical infrastructure and virtualization layer are on the provider's realm. In this model, the consumer has the highest level of control over the resources the remote cloud infrastructure provisions to him (e.g., control over a whole operating system).

- **Platform as a Service (PaaS):** This type of service model gives the consumer permissions to deploy his own applications onto the cloud infrastructure. The consumer can create or acquire applications, which are typically implemented using programming languages, libraries, services, and tools offered by the provider. Like in IaaS the consumer has no control over the underlying physical infrastructure or virtualization layer, and in this model the control over the remote operating system is also removed.

The consumer has control over the deployed applications and in some cases can also manipulate specific environment configurations of hosting applications.

- **Software as a Service (SaaS):** The consumer sees further reduction in his permissions over the remote cloud infrastructure. In a SaaS scenario the consumer only has control over a set of user-specific application configuration settings. This means the consumer is only allowed to use applications the cloud provider runs and offers in the cloud infrastructure.

2.5 Trustworthy Computing

The not-for-profit Trusted Computing Group (TCG) is responsible for the definition of standards for trustworthy, or trusted, computing. This standards body is an initiative of major information technology companies such as Intel and IBM [97].

An important standard TCG created is the Trusted Platform Module (TPM), which defines hardware or software based security extensions for computing platforms. The implementation of this standard protects a system from unauthorized modifications and attacks such as root kits. TCG has expanded its hardware-based security extensions to multiple platforms ranging from hard disk drives to mobile phones. Intel's Trusted Execution Technology (TXT) and AMD's Secure Virtual Machine (SVM) are examples of implementations of hardware-based security extensions defined in the Trusted Platform Module standard [30, 91].

The subsections that follow introduce the key concepts and elements of a Trusted Platform Module. The information provided in this section does not intend to be exhaustive.

2.5.1 Trusted Platform Module

The Trusted Computing Group defines the Trusted Platform Module (TPM) as a microcontroller with capacity for secure storage of keys, passwords, and digital certificates [98]. The secure storage of sensitive data is assured through tamper-protection measures such as tamper resistance and evidence. This protection prevents physical tampering of the TPM module [95].

A *root of trust* is a hardware or software mechanism a system user implicitly trusts [29]. TPM's tamper-protection features make it a logic choice for a *root of trust*. A root of trust typically provides three roots of trust, a root of trust for measurement (RTM), root of trust for storage (RTS), and root of trust for reporting (RTR) [95].

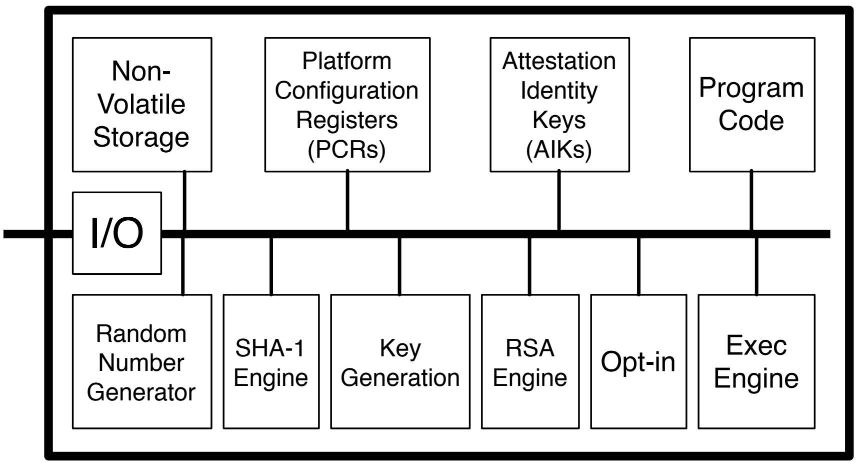


Fig. 2.5 Trusted Platform Module (TPM) [95].

- **Root of Trust for Measurement (RTM).** The RTM consists of an inherently reliable implementation of a secure hash function, which may or may not reside within the TPM. This hash algorithm is trusted to provide system integrity measurements used in establishing the trustworthiness of a platform. An integrity measurement is the process of collecting a hash code of software or data.

- **Root of Trust for Storage (RTS).** The group of TPM components and features responsible for securely storing information such as hash codes obtained in integrity measurements.

- **Root of Trust for Reporting (RTR).** The set of TPM components and mechanisms involved in reliably reporting the state of the platform in a verifiable manner.

These roots of trust are building blocks in mechanisms used to establish the trustworthiness or integrity of a TPM-enabled platform.

Figure 2.5 illustrates the components of a Trusted Platform Module (TPM) according to Trusted Computing Group's specifications. The most relevant components for this document are introduced in the subsections that follow. These components are introduced together with the functionalities they enable.

Cryptographic Keys and Data Storage

The Trusted Platform Module provides both non-volatile and volatile data storage. The non-volatile storage is important to hold the *endorsement key* (EK), *storage root key* (SRK), and other data used to manage the TPM state. Due to its security sensitive nature, these keys are stored in shielded-locations only accessible through a set of commands with the required permissions. The EK and SRK are public-key cryptography key pairs. The TPM stores the private key of each of these key pairs [29, 95].

The endorsement key is the foundation of the root of trust for reporting. The endorsement key pair is generated in the manufacturing process. An endorsement key uniquely identifies a platform, and ideally, should remain unchanged throughout the life of the associated TPM. However, TPM vendors can expose functionality that allows an endorsement key to be changed [29]. Taking ownership of a TPM allows the platform owner to create a storage root key pair. In the event of a change of owner, the new TPM owner can generate a different storage root key pair. The SRK is the base of the root of trust for storage.

Since using the endorsement key for signing platform state reports can violate the privacy of the platform. The endorsement key can be used to generate Attestation Identity Key (AIK) pairs, which permit a remote entity to verify it is communicating with a TPM without revealing which TPM. The TCG recommends attestation identity keys to be stored as blobs in persistent external storage, instead of using TPM's non-volatile storage.

Platform Configuration Register

Platform configuration registers (PCRs) are a set of storage locations whose principal objective is to provide secure storage for the hash codes collected during integrity measurement processes. The minimum number of platform configuration registers is sixteen, with registers 0-7 reserved for TPM use and the remainder free for an operating system and/or applications to use. The values stored in platform configuration registers are not lost with a system reboot but are reset whenever the platform loses power. The operations of read and write for a platform configuration register (PCR) require special TPM commands and a write operation can never be directly performed on a PCR [29, 95].

The write operation is actually an extend operation, typically referred to as *extending the PCR*. There are two types of data involved in an integrity measurement process. The first type is known as *measured values*, which are a representation of embedded data or program code. The second type are hash codes of the measured values, referred to as measurement digests.

A representation of the extend operation can be seen in equation 2.1. An extend operation consists in using the current PCR value, concatenating it with the hash code of a measured value (<*new_hash*>), and computing the hash code of the concatenated value. The index letter *i* refers to the current PCR number [29, 95].

$$PCR[i] = SHA1(PCR[i] \mid\mid < new\_hash >) \tag{2.1}$$

The extend operation provides some interesting advantages such as fix storage requirements and order preservation. The output of an extend PCR operation is a fixed-length hash code, which means a single PCR is usable for an unbounded number of extend operations. This makes the extend operation an optimal choice from a storage perspective.

The order preservation property can prevent an entity from manipulating the order in which the integrity measurements were performed and stored. A secure hash function satisfying the requirements defined in Subsection 2.2.1 can guarantee this property. Let us consider an attack scenario where entity B, which executed after A and before C, wishes to remove itself from the set of measured entities. Consider $x = H(A)||H(B)$ and $z = x||H(C)$. From requirement 6, it is computationally infeasible to find any pair (x,y) such that $H(x) = H(y)$. Therefore, it is not possible to remove B from the execution path without influencing the result of $H(z)$.

2.5.2 TPM Functionality

The components and properties of the Trusted Platform Module allow it to offer interesting security functionalities such as transitive trust, sealed storage, and remote attestation. The remainder of this subsection describes this set of functionalities.

Transitive Trust

The objective of transitive, or inductive, trust is the use of a single root of trust to enlarge the set of trustworthy entities in the platform. The process of transitive trust has three main steps. First, the initial root of trust performs an integrity measurement of the entity it wishes to add to the set of trustworthy entities. The measured value in transitive trust is program code, whose measurement digest is useful in reaching the decision of whether the entity can be trusted or not [29, 95].

Second, after performing the integrity measurement, the root of trust extends a platform configuration register with the hash code of the measured value. Storing this value is useful for logging and accountability.

Finally, if the root of trust considers the measured entity to be trustworthy, the set of trustworthy entities expands to include the functionality of the measured entity. Since the measured entity is deemed trustworthy, it can then determine if other entities are trustworthy or not. The process is iterative.

Sealed Storage

Sealed storage is a mechanism to protect private information from unauthorized accesses. The two main processes in this mechanism are the *sealing* and *unsealing* operations. This mechanism combines the use of encryption, the root of trust for storage (RTS), and the measurement digests to provide protected storage of data [29].

The required inputs for the sealing process are the data to protect, and the selected measurement digest(s) stored in platform configuration register(s). The seal process then uses a storage key, which can be the storage root key (SRK) itself or a storage key that is part of the SRK tree. This key encrypts the data creating the sealed data package. The sealing process is internal to the TPM [29].

The seal process binds the sealed data package to the platform configuration, including hardware and software, where it is performed. This implies that the sealed data can only be revealed when the platform is in the same state it was when the sealing operation took place. This behaviour is enforced through the inclusion of a nonce that only the TPM that performed the operation knows [29, 96].

The purpose of the unseal operation is to retrieve the data in a sealed data package. This operation is successful if and only if it is performed in the same TPM that created the sealed data package, data integrity was not compromised, and the measurement digest(s) in the used platform configuration register(s) are correct.

The first step is for the TPM to unseal the information within the sealed data package. This includes decrypting all the data using the right storage key and verifying the data was correctly decrypted. This step reveals the expected nonce and measurement digest(s).

The final steps use the nonce and measurement digest(s) to finish the operation. First, the retrieved nonce is matched against a nonce internally held in the TPM. If the values match then the TPM must be the TPM responsible for creating the sealed data package. Finally, the measurement digest(s) retrieved in the operation need to be compared with the measurement digest(s) stored in the expected platform configuration register(s). If these verifications are successful the operation returns the protected data otherwise it fails [29, 96].

Remote Attestation

The necessity for remote attestation arises from the need remote platforms have of verifying if a trustworthy platform configuration is present. The initiator in an attestation process is usually the remote verifier. The main attestation process is divided in two principal stages.

One of these stages takes place on the platform under verification. This platform receives a request for the execution of TPM's TPM_Quote command together with a set of platform configuration registers to quote, a nonce to prevent replay attacks, and the attestation identify key to use in the digital signature. Upon reception of this request the platform's TPM verifies the authorization to use the desired attestation identity key, creates a structure holding the measurement digests in the requested set of platform configuration registers, combines this information with the provided nonce, and produces a digital signature.

The next stage takes place on the platform performing the verification. After requesting the quote operation, this platform receives a fresh digital signature of the desired platform configuration registers. The public key of the private-public attestation identity key pair is used to verify the signature, the AIK key itself is verified, and then the measurement digests are checked. This last step evaluates the trustworthiness of the remote platform, which can either be trustworthy or not [29, 96].

Chapter 3

Literature Review of Virtualization and Security

This literature review presents the challenges of protecting against a malicious insider in cloud computing and the state-of-the-art solutions that try to prevent such threat. The chapter begins with a definition of the adversary model and the data execution challenge which is our main concern. Then, we review several security solutions grouping them according to the major prevention technique they use. The three prevention techniques we analyse in detail are cryptography, virtualization, and hardware-based approaches.

The difficulties associated with preventing insider threats creates necessity to devise insider prevention techniques that rely on cryptography, virtualization, hardware, or combinations of the three. The prevention difficulties are mostly due to the challenges behind the data execution problem in the cloud.

Data execution in cloud computing refers to running applications that perform operations over data that a cloud consumer entrusted to a cloud provider. When this data is security sensitive the system must assure the data's confidentiality and integrity while, at the same time, the system needs to perform operations over it. This is known as the data execution problem. Hence, the data must either reside in the system in plaintext or the system needs a mechanism to perform operations over encrypted data. However, the impossibility of solving the data execution problem has been proved for multiple cloud consumers using a cryptography-only approach in a scenario where there is no interaction between consumers [100]. This work highlights how cryptography alone cannot guarantee privacy in scenarios where a cloud service performs arbitrary operations over consumers' unencrypted data. Therefore, additional security enforcement approaches such as trusted computing and complex trust ecosystems need to be used to try and remedy this problem.

The remainder of this chapter defines an adversary model and reviews research work that addresses the data execution problem in cloud computing. We group the multiple works according to the major security mechanism used. This does not mean that a hardware-centred approach cannot take advantage of cryptographic primitives and vice versa. We designate the approaches as *centred* in the sense that they use a specific tool or primitive as its main mechanism for preventing a security threat. For example, a cryptography-centred uses cryptography as its main tool to prevent known security threats.

3.1 Adversary Model

The adversary considered in this thesis is a malicious insider with superuser privileges over the systems within a cloud computing infrastructure. Consider the conceptual definition of a malicious insider from the CERT Program at Carnegie Mellon University's Software Engineering Institute. According to which "a malicious insider is a current or former employee, contractor, or other business partner who has or had authorized access to an organization's network, system or data and intentionally exceeded or misused that access in a manner that negatively affected the confidentiality, integrity, or availability of the organization's information or information systems." [14].

From a more practical perspective, we assume the attacker has no physical access to cloud servers' hardware, which means our work does not consider hardware-based attacks such as connecting external peripherals to collect memory dumps or tamper with the integrity of hardware security modules (e.g., Trusted Platform Module).

It is assumed the attacker can change the virtual machine monitor, recompile it, and have a server execute with this new rebuilt version. We are aware of the possibility of attacks originating from a consumer's virtual machine but that type of threat is not considered in our work [108]. The adversary is capable of compiling and executing arbitrary software with superuser privileges within the realm of cloud management software.

These considerations reflect the requirements expected in a real cloud computing infrastructure where a cloud administrator uses superuser privileges to configure and maintain cloud servers.

3.2 Cryptography-centred Solutions

The features and benefits of using cryptography as a security mechanism have long been known and validated within the research community. Common uses of its capabilities include using encryption for strong confidentiality, or message authentication codes for data integrity. However, cloud computing creates new, not easily solvable, research challenges such as the data execution problem for which cryptography-only solutions are not enough [41]. This limitation does not invalidate the use of cryptographic primitives as strong security tools in different security approaches.

Cryptography-based solutions are a common approach to solving the data storage problem. The data storage problem occurs when a cloud consumer wishes to use cloud resources to safely store security sensitive data.

A discussion of multiple cryptography-based cloud storage security solutions for cloud computing can be found in [41]. Those security solutions include the secure enterprise-class file system Iris which provides data integrity for cloud storage to protect against data corruption and guarantees freshness to prevent rollback attacks [89]. The file system does not address the insider threat as discussed in this thesis.

HAIL is a high-availability and integrity layer for cloud storage which uses storage resources from multiple cloud providers to build a cost-effective and reliable cloud storage solution from untrusted resources [12]. We do not provide much detail about these solutions because they address the data storage problem which is not in the scope of our work. For a good discussion on these solutions we refer the reader to [41].

In what follows, we provide a description and analysis of a security solution that uses cryptographic mechanisms to address the data execution problem. To the best of our knowledge this is the only solution that, despite using other technologies, uses cryptography as its main mechanism to addresses the data execution problem. This research work was published with the following title:

- *Secure virtual machine execution under an untrusted management OS. (2010)* [50]

This cryptography-centred approach envisions a secure virtualization architecture to prevent attacks from a malicious management virtual machine executing in cloud servers. The purpose of such architecture is to guarantee data confidentiality and integrity for consumer virtual machines. A prototype was implemented using the Xen hypervisor [50].

Although it is a secure virtualization architecture, the authors use encryption to protect the confidentiality of a virtual machine's data such as its memory pages. The hypervisor encrypts the memory pages before handing them over to the management virtual machine.

The system computes a cryptographic hash code of the data before proceeding with its encryption. This hash code is used to guarantee data integrity through integrity checks when the data is decrypted and reloaded within the VM. The addition of version information

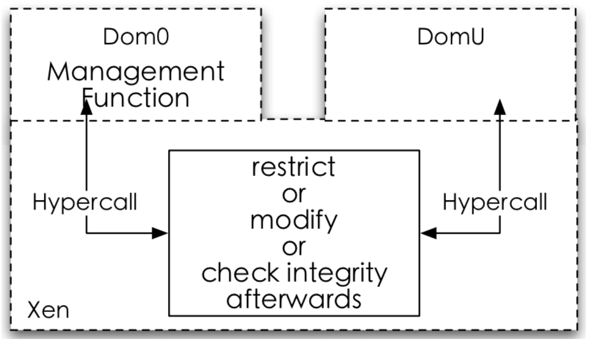


Fig. 3.1 Trusted Computing Base (Security Enforcing Operations) [50].

in the computation of the hash code assures data freshness. For example, a hash code would be calculated before saving a VM's state, and the same hash would be used to verify data integrity once the VM is restored. Operations of save and restore are common in virtualization solutions.

This cryptography-centred approach is effective in guaranteeing data confidentiality and integrity for consumer owned data. However, it has two limitations which are lack of versatility and the overhead associated with its cryptographic operations.

The lack of versatility is a considerable limitation in this approach as it prevents the development of virtual machine monitoring solutions. This happens because the state of a virtual machine is always encrypted. Therefore, it is not possible to implement cloud- based solutions that can offer services based on reading a virtual machine's state. The overhead of encryption and decryption operations combined with key management issues can also become a problem in this solution.

Figure 3.1 illustrates how the security enforcing operations of this approach are performed in the hypervisor layer. This effectively reduces the trusted computing base (TCB) by removing the operating system of the management virtual machine (Dom0) from it. Removing the management virtual machine from the TCB is a step forward because it reduces the attack surface for adversaries trying to exploit software vulnerabilities to compromise the system. However, the implementation of its functionalities is going to have an impact in the number of lines of code of the hypervisor.

3.3 Virtualization-centered Solutions

The idea of using a security kernel to enforce computer security has been around since the early 1970s. A security kernel emerges as a possible reference validation mechanism, which is the combination of hardware and software to implement the concept of a reference monitor [88]. The concept of a reference monitor states that all references by any program to any other program, data or device are validated against a policy that defines the authorization for each type of reference according to user and/ or program function within a computer system [88].

From our point of view, exploring the security capabilities of virtualization is no more than using a hypervisor as a reference validation mechanism which imitates a security kernel in commodity systems. This approach enforces the principle of complete mediation. To the best of our knowledge, the solutions that follow are the only known ones that exploit the security capabilities of virtualization which could be used to prevent the malicious insider threat.

In this section we discuss the following virtualization-centred solutions:

- SecVisor [83]
- HyperShot [85]
- Dom0 Disaggregation [59]
- NOVA: a microhypervisor architecture [90]
- sHype Hypervisor [76]
- CloudVisor [106]
- Xoar [16]
- VMGuard [24]
- Min-V [62]

3.3.1 SecVisor

SecVisor is a small code base hypervisor for commodity systems with the objective of protecting an operating system from advanced attacks such as kernel rootkits. A kernel rootkit is a set of tools that allow an attacker to maintain kernel-level privileges over a compromised system [34]. The three main objectives in SecVisor's design are a reduced trusted computing base to permit formal validation, reduced external interface to minimize the attack surface, and minimal kernel changes for easy porting of commodity kernels [83].

Executing at a virtual machine monitor privilege level allows SecVisor complete control over the protection of the kernel under which it executes. This level of privilege is paramount to ensure the integrity of kernel code.

SecVisor virtualizes physical memory to assure protection of a kernel's memory space. The option for virtualizing physical memory reduces the size of SecVisor's interface because it removes the need for a hypercall to maintain memory page tables. It also simplifies portability by removing the need for kernel changes to interact with the aforementioned hypercalls. These advantages respect the design principles defined by the authors.

In order to guarantee the integrity of kernel code, SecVisor sets the CPU to refuse the execution of any code that it has not approved. This means that the user defines a security policy where approved code is listed and SecVisor uses such policies to make its decisions. This does not mean that SecVisor prevents alteration of kernel code that can happen through code injection, but it does prevent the injected code from executing because it is not approved to execute.

The authors implemented SecVisor on top of an AMD-powered system taking advantage of AMD's Secure Virtual Machine (SVM) technology. SecVisor was used to protect a ported Linux kernel. Therefore, SecVisor does not target cloud computing systems but commodity systems. However, its design principles and security objectives are valid for a virtual machine monitor designed to assure a secure cloud computing environment.

3.3.2 HyperShot

HyperShot is a hypervisor-level mechanism designed to obtain trustworthy virtual machine snapshots [85]. It was implemented in a research prototype version of Microsoft's Hyper-V hypervisor. Since HyperShot is part of the hypervisor, which executes at an elevated privilege level, a malicious insider with control over the privileged management virtual machine (e.g., Dom0 for Xen or root VM for Hyper-V) cannot compromise the memory space allocated to HyperShot.

The integrity of snapshots is assured through the use of a cryptographic hash of the snapshot. This hash code is calculated in the hypervisor and sent to the privilege VM together with the snapshot. A malicious insider could attack the cryptographic hash to compromise the integrity of a snapshot, that is however not possible because the snapshot hash is signed using the signing capabilities available in a cloud server's TPM. HyperShot uses copy-on-write (CoW) for performance reasons and to guarantee consistency when a snapshot is built. As an example the authors mention it took 391 seconds to obtain a trusted snapshot of the privileged VM. A hypervisor that is HyperShot compliant allows a cloud consumer to perform a remote attestation of its virtual machine.

The Hypershot protocol includes four main participants: (1) the cloud consumer requesting the attestation, (2) the HyperShot service front-end responsible for routing the snapshot request to the correct cloud server, (3) the HyperShot proxy component which receives the request from the front-end and uses an hypercall to pass it to the hypervisor, and (4) the HyperShot mechanism inside the hypervisor responsible for generating the signed snapshot hash and sending it back to the cloud consumer through the reserve path of the snapshot request.

HyperShot assures data integrity for a VM's snapshot by relocating the snapshot feature into the hypervisor. Moving this operation to a higher privilege level of execution guarantees that malicious insiders do not have enough permissions to attack it. This solution is however not concerned with the confidentiality threats a consumer's data faces when executing in a virtual machine. Removing the privileged management virtual machine from the trusted computing base guarantees a smaller TCB for an HyperShot-enabled hypervisor when compared to typical solutions, e.g., an unmodified Xen version.

3.3.3 Dom0 Disaggregation

Murray et al. authored one of the first studies to discuss the problems of having a large trusted computing base in a virtualization environment. They propose a "trusted virtualization" solution where the privileged virtual machine is disaggregated to achieve better overall security [59].

In their work, besides the traditional number of lines of codes, the authors suggest two new criteria to measure the trustworthiness of a TCB. They propose the size of the interface and the size of the TCB state space as a complement to previously used criteria. These two new measurement properties are used to approve their disaggreation of Xen's management virtual machine, or Dom0, where the *domain builder* process is transferred to a different special purpose virtual machine. The domain builder process includes the required steps to launch a new virtual machine in a cloud server. Their final solution includes, a TCB comprised of the hypervisor, Dom0's kernel, and part of the domain builder, and the lines of code used for interface sanitizing code.

This disaggregation work is a step in the right direction because it moves the security sensitive operations to a special-purpose virtual machine. However, it still includes a considerable amount of code which enlarges the attack surface attackers can exploit, e.g., including a whole Linux kernel. Two main factors suggest this approach has room for improvement.

First, at the time of writing, the Linux kernel code base grew more than a million lines of code in one year to a total of almost seventeen million lines of code [64]. Second, a news article summarizing information from the 2012 Trustwave Global Security Report informs that the Linux Kernel had nine critical vulnerabilities, including two zero-days, and that the average response time to fix the zero-days was more than two years [33]. More recent data from an executive summary of Trustwave's report for the year of 2013 says that: "Linux had the worst response time, with almost three years on average from initial vulnerability to patch.".

This information shows that there is a considerable security risk in including a whole Linux Kernel in the TCB of a virtualization architecture. This solution does not use the hypervisor as a reference validation mechanism to assure confidentiality and integrity for consumer's data. Therefore, it is required to trust the whole TCB just discussed, which does not make this an ideal solution.

3.3.4 NOVA: a microhypervisor architecture

The NOVA approach is based on a microhypervisor architecture instead of the most common monolithic approach used in solutions such as the Xen hypervisor and the Linux Kernel-based Virtual Machine (KVM) [90].

The traditional monolithic approach is typically a large code base, including device drivers and the functionality required to support the execution of one or several guest operating systems. This architecture means that an exploitable security flaw found in the monolithic block of code can be used to subvert the hypervisor compromising the security properties it guarantees.

NOVA follows the microhypervisor architecture to reduce the trusted computing base by at least an order of magnitude when compared with other solutions. NOVA respects two fundamental design principles. The first principle states that it must provide a fine-grained functional decomposition of the virtualization layer into a microhypervisor, a root partition manager, a single virtual machine monitor per virtual machine, device drivers, and other system services. The second principle reads that the principle of least privilege is enforced between the multiple components just listed.

Although NOVA presents an alternative that can reduce the trusted computing base when compared with approaches such as Xen and Linux KVM, to the best of our knowledge it is not clear how such architecture assures data integrity and confidentiality when facing an insider threat. It is important to mention that NOVA does not target the malicious insider threat, it was developed as an alternative approach to the virtualization of computational resources.

3.3.5 sHype Hypervisor

The sHype hypervisor security architecture implements a mandatory access control policy-based reference monitor for the Xen hypervisor. The objective of this architecture is to use formal security policies that control the sharing of resources between virtual machines through the use of a reference monitor. The reference monitor mediates all security-sensitive operations [76].

Multiple security functions are provided, including secure services, resource monitoring, access control between VMs, isolation of virtual resources, and TPM-based attestation. The components of the sHype mandatory access control architecture are: the policy manager, the *access control module* (ACM), and *mediation hooks*. The policy manager is a special-purpose VM used to maintain the security policies. The ACM and mediation hooks are implemented in the hypervisor. The former is responsible for delivering authorization decisions based on the security policies, while the latter controls the access VMs have to the policy-affected resources.

A malicious administrator can attack this solution because they are responsible for defining the policies that state which security mechanisms should be deployed. Therefore, this approach is not ideal to face a malicious insider threat because administrators still have a considerable amount of control over the security of the platform.

The simple use of a TPM-based attestation is vulnerable to time-of-check time-of-use (TOCTOU) attacks [59]. This solution does reduce the trusted computing base but the definition of security policies is dependent on humans, so a malicious insider can compromise data integrity and confidentiality using a permissive security policy which can escape the attestation due to the TOCTOU attack.

3.3.6 CloudVisor

CloudVisor is yet another different secure virtualization architecture where a featherweight hypervisor is placed between the virtual machine monitor and the hardware [106]. The virtual machine monitor is deprivileged together with the management virtual machine. The main objective is to have CloudVisor monitor the hardware resources usage of the virtual machine monitor and the VMs. This positioning allows CloudVisor to enforce isolation while at the same time protect the resources used by each guest VM. Although CloudVisor assures isolation for the resources of VMs, it has a few disadvantages in terms of performance and some architectural limitations.

This solution guarantees confidentiality and integrity for a consumer's data executing in virtual machines. However, CloudVisor incurs serious performance penalties, in some cases over 22%, when compared with the Xen hypervisor. The suggested architecture impedes the use of tools that require access to the memory space of consumers' virtual machines to perform monitoring tasks. The TCB is reduced to the lines of code of CloudVisor which is a very good advantage.

3.3.7 Xoar

Xoar is another virtualization platform architecture that follows on the footsteps of previous work on improving Xen's security through disaggregation while introducing a few novel mechanisms. This new architecture introduces the modularity and isolation principles used in micro-kernels [16].

Some of the novel functionality Xoar supports includes disposable bootstrap and auditable configurations. A disposable bootstrap means having special purpose VMs responsible for performing operations that are only required at boot time, these VMs are destroyed once the boot process is concluded. Xoar keeps secure audit logs of different system configurations that can later be used in queries to discover, for example, a list of VMs associated with a known-vulnerable component.

Virtual machines that can be restarted to maintain freshness and isolation are also used to harden critical components, e.g., XenStore. The disaggregation overhead is quite low but having the VM that executes device drivers restarting causes noticeable overhead because the intermittent outages lead to TCP performance degradation. However, the administrators can configure the restart frequency and reduce the overall performance impact.

Even though Xoar does not target the malicious insider threat, its least privilege access approach is ideal to interact with a hypervisor prepared to enforce the principle of least privilege which can protect against malicious insiders. Since its protection mechanisms were not designed considering an adversary model that includes a malicious administrator, it is not a solution that guarantees data integrity and confidentiality when facing such an adversary.

3.3.8 VMGuard

VMGuard is an integrity monitoring system for management virtual machines such as Xen's Domain0. This solution achieves real-time monitoring through two special purpose virtual machines denominated *GuardDomain* and *GuardDomainU*. These VMs run on a server within a cloud infrastructure to monitor its co-resident management virtual machine [24].

GuardDomain collects integrity measurements of the privileged virtual machine, while GuardDomainU closes the memory semantic gap between hypervisor and consumer virtual machine. *GuardServer* is a third entity dedicated to storing the integrity measurements collected by GuardDomain and verifying the trustworthiness of management virtual machines. The authors implemented a prototype using the Xen hypervisor.

A serious limitation in this solution is that the same virtual machine it wishes to monitor is the one responsible for bootstrapping and verifying the integrity of GuardDomainU. This is not secure when a malicious insider has control over the management virtual machine. Hence, if GuardDomainU is changed to behave maliciously the verification process is not going to detect this because Domain0 can be under control of a malicious administrator. Therefore, this solution is not ideal to guarantee data integrity and confidentiality against malicious insiders.

3.3.9 Min-V

Min-V is a tiny hypervisor with a different approach to reducing the trusted computing base in a cloud computing environment. The authors argue that the largest percentage of software vulnerabilities seen in virtualization environments come from the software implementing virtual devices. Therefore, they propose disabling non-critical virtual devices and reducing the functionality in the critical ones [62].

Reducing the functionality of virtual devices a virtual machine requires has an obvious impact on the boot process of a commodity operating system. To solve this problem the authors devised delusional boot, a novel launch mechanism for virtual machines executing in a cloud computing environment that uses Min-V.

The delusional boot process requires an isolated node in the network to perform the initial boot of virtual machines. The process consists of three major steps. First, an image of the consumer's virtual machine is sent to a dedicated boot server. Second, a Min-V boot stack powered boot server receives the image, disconnects from the network, reboots into a full virtualization stack, and finally boots the virtual machine. The server then obtains a snapshot of the VM, reboots to the initial state, reconnects to the network, and finally sends

Solution	Confidentiality	Integrity	TCB	Cloud	Insider Threat
SecVisor	✔	✔	✔	✘	✘
HyperShot	✘	✔	✘	✔	✘
Dom0 Disaggregation	✔	✔	✔	✔	✘
NOVA	✘	✘	✔	✔	✘
sHype	✔	✔	✘	✔	✘
CloudVisor	✔	✔	✔	✔	✔
Xoar	✘	✔	✔	✔	✘
VMGuard	✘	✔	✘	✔	✘
Min-V	✘	✘	✔	✔	✘

Table 3.1 Virtualization-centred solutions summary. ✔ = guarantees/addresses; ✘ = not guarantees/addresses.

the snapshot to a production server. Finally, the VM snapshot is sent to a production server, which is running Min-V's virtualization stack. This server is responsible for replacing all the disabled virtual devices for a null device.

This approach achieves the objective of reducing the trusted computing base of the virtualization stack but at the same time it increases the complexity of the process of booting up a virtual machine. The design principles mentioned in the paper do not include the principle of least privilege, so it is not clear how this approach would behave when facing an insider threat. An attack point could be the boot servers.

3.3.10 Summary

This subsection includes a table summarising how the solutions we analysed in this section relate to the properties, environment, and threat we are considering in this thesis.

Table 3.1 shows how each of the solutions analysed in this section do with respect to *confidentiality*, *integrity*, *trusted computing base*, *cloud*, and *insider threat*. For the *confidentiality* and *integrity* properties we verified if the solutions are concerned with enforcing these properties for consumers' data resident in virtual machines' memory space. The *trusted computing base* column indicates if a solution reduces the TCB or not. Both the *cloud* and *insider threat* show if the solution in question was developed for a cloud environment and if it considers the malicious insider threat when trying to guarantee confidentiality and integrity for consumers' data resident in a virtual machine's memory space.

The work done on *Dom0 disaggregation* and *sHype* shows interesting approaches when considering an insider threat but they maintain a considerably larger TCB. The most noticeable aspect common to almost all the solutions analysed in this section is that only one of them (i.e, CloudVisor) considers the insider threat from a similar perspective to what we do in our work. This supports our conviction that more research needs to be done in order to improve prevention mechanisms to protect a cloud system from insider threats.

3.4 Hardware-centred Solutions

The first references to trusted computer systems date back to late 1970s and early 1980s [93]. Although the document mostly focuses on the strategy for computer security of the United States' Department of Defense (DoD), the importance of concepts such as trusted computing base is already considered. In this section, we consider the Terra architecture as the initial reference in more recent work regarding trusted computer systems in a virtualized host [27].

The complete list of hardware-centred approaches discussed in this section is the following:

- Terra [27]
- Trusted Virtual Datacenters (TVDc) [7]
- Private Virtual Infrastructure (PVI) [49]
- NoHype [44]
- Trusted Cloud Computing Platform (TCCP) [79]
- Excalibur [80]
- TrustVisor [54]
- myTrustedCloud [103]
- Strongly Isolated Computing Environment (SICE) [4]

3.4.1 Terra

Terra is a security architecture with its foundation in a virtual machine monitor but it is not intended for cloud computing. It is included in this document because to the best of our knowledge it is the oldest among recent work addressing security problems similar to the ones discussed in this thesis. Terra uses a virtual machine monitor to virtualize the hardware resources and it takes advantage of hardware-enforced properties to offer remote attestation of running software [27].

A Terra managed system offers an *open-box VM* or a *closed-box VM* as the two possible virtual machine abstractions. An open-box VM emulates the properties of standard open systems supporting the execution of commodity operating systems. A closed-box VM implements a closed system environment where the content is not accessible to the platform owner.

The Terra architecture also includes a management virtual machine responsible for operations such as assigning storage and memory resources to VMs, or starting and stopping VMs. Terra relies on the VMM to assure the isolation, extensibility, efficiency, compatibility, and security properties. The VMM is assumed to be root secured, which means not even the platform administrator can break its security properties. Functionality like remote attestation and a trusted path between user and application are also guaranteed through VMM properties.

From the information provided in the paper we assume the management virtual machine responsible for starting and stopping virtual machines must be implemented in a similar fashion to what is done in an off-the-shelf Xen hypervisor. Therefore, since the paper does not consider the malicious insider threat there is a high probability that a malicious administrator can violate the confidentiality and integrity of consumer's data.

3.4.2 Trusted Virtual Datacenters (TVDc)

Although IBM's Trusted Virtual Datacenters (TVDc) do not specifically address the malicious insider threat, it is relevant to discuss it here because such solution is referenced in Private Virtual Infrastructures as the foundation for its correct operation.

The virtualization of datacenters brings with it new infrastructural and management issues which IBM's TVDc as a security solution tries to address. In TVDc, a workload is denoted as a Trusted Virtual Domain (TVD) which is a group of virtual machines and resources that cooperate to achieve a common goal [7].

Strong isolation between TVDs is enforced through the sHype hypervisor, which enforces Mandatory Access Control (MAC) policies [76]. A unique security label is assigned to the members of a TVD in order to enforce the access control policies. For example, if a VM and a resource have security labels that match, the VM is granted access to the resource, otherwise if the security labels do not match that access is denied.

The access control policy defines which VMs can access which resources and which communication links can be established between VMs. Policies can also be used to define which VMs a hypervisor can run, or to configure anti-collocation rules restricting which VMs can execute simultaneously in the same cloud server.

Finally, a TVDc also offers integrity guarantees on the software executing in TVDs' VMs through the use of trustworthy computing technology. The hardware of a cloud server only has one physical TPM, to satisfy the need of having a TPM per VM, TVDc includes a virtual TPM (vTPM) per VM executing in a cloud server in its privileged management VM [6]. The vTPMs are used to provide assurances on the integrity of the software executing inside the VMs.

This solution does not address insider threats. Since cloud administrators are the ones responsible for establishing security policies in data centres, they have a privilege position to manipulate those policies. Previous research has demonstrated security vulnerabilities are present in the virtual TPM approach [59].

3.4.3 Private Virtual Infrastructure (PVI)

Private Virtual Infrastructure (PVI) suggests a virtual datacenter on top of the existing cloud infrastructure as a solution to assure privacy and security for data a consumer entrusts to a cloud provider [49]. This approach is mostly based on IBM's Trusted Virtual Datacenters solution except for the inclusion of a special purpose virtual machine denominated as Locator Bot (LoBot) and a consumer's PVI factory.

LoBot is a self-contained virtual machine with a vTPM bound to the physical TPM of a cloud server. Within LoBot there is an application responsible for collecting information on the physical cloud server including integrity measurements stored in the physical TPM's PCRs. This information is then sealed and sent to the consumer's PVI factory to allow the consumer to verify the trustworthiness of the remote cloud platform. If the cloud platform is deemed trustworthy the PVI factory configures and encrypts a consumer VM for launch in the cloud infrastructure. Only a platform classified as trustworthy is capable of deciphering the encrypted virtual machine.

The paper also suggests the use of LoBot as a means to achieve secure VM shutdown, data destruction, continuous monitoring, and auditing. It is not clear if this features were fully implemented and tested.

Both IBM's Trusted Virtual Datacenters (TVDc) and Private Virtual Infrastructure (PVI) suffer from the weaknesses inherited from the sHype hypervisor which they use as their virtualization layer. We have already discussed how giving control over the security policies to the administrator makes it vulnerable to insider threats.

Not only that but it was proved that the virtual TPM (vTPM), used in both solutions, is vulnerable to time-of-check-time-of-use (TOCTOU) attacks which undermine the security of the whole system [59]. These security issues affect the confidentiality and integrity of data cloud consumers entrust to cloud providers.

3.4.4 NoHype

NoHype proposes removing the virtualization layer to remove the security implications of using such solutions, e.g., side-channel vulnerabilities. The core idea in NoHype is to leverage current hardware virtualization technology to eliminate the need for a traditional virtualization layer [44].

The authors provide multiple evidences to demonstrate it is practical to remove the hypervisor. First, it is shown how the architecture can run multiple virtual machines in one physical server taking advantage of its hardware supported virtualization. Second, a discussion is given on how to partition main memory and Input/Output (I/O) devices. Third, the work demonstrates how networking can be moved to the physical network infrastructure. Finally, the authors also show how to start/stop/migrate virtual machines.

Assigning a CPU core to an individual VM is how the CPU resources are shared. NoHype uses the multi-core memory controller (MMC) to partition physical memory among VMs. A similar approach is used to share I/O devices but using the Input/Ouput Memory Management Unit (IOMMU). The networking layer eliminates the traditional virtual Ethernet switch and suggests using the physical network infrastructure to forward data between VMs. A system manager software package runs in one of the cores and is responsible for starting/stopping/migrating virtual machines.

The authors discuss two limitations for their solution, which are selling in extreme fine grain units (e.g., sell a 1/4 of a core) and highly over-subscribe a physical cloud server (i.e., sell resources over the available amount). However, they argue that future hardware will offer enough cores per chip to counter the first limitation, and that the second policy goes against the cloud model.

NoHype's design makes it strong in preventing side-channel attacks between co-resident virtual machines in order to offer data confidentiality. The isolation is guaranteed through hardware mechanisms such as the multi-core memory controller and the input/ output memory management unit (IOMMU). Therefore, threats like the timing cross-VM attack on cryptographic keys can be prevented if the hardware behaves as expected [108].

Although using hardware to enforce isolation between VMs is a very strong approach, the way the system manager starts virtual machines continues to leave some attack windows for malicious insiders to exploit. The process of starting a virtual machine involves having the system manager map the memory and disk of the new VM into its own memory space. This means a malicious administrator has complete access to the memory area of a VM and it is also responsible for getting the disk/VM image, so it is possible to tamper with the image and inject malicious software in it.

The system manager continues to be a threat to the integrity of disk images because it maps the memory space assigned to a VM and is responsible for getting the VM image scheduled to be loaded. Another concern arises from the information provided in the paper. Since the system manager is actually responsible for mapping and un-mapping the memory space assigned to a VM it means that a malicious version of the system manager could skip this step and keep full access to the memory space of consumer VMs.

Despite all these concerns it is important to remember that NoHype was not designed specifically to address the malicious insider threat. Therefore, this is an interesting alternative to using a virtualization layer that can surely be hardened to prevent insider threats.

3.4.5 Trusted Cloud Computing Platform (TCCP)

The Trusted Cloud Computing Platform (TCCP) leverages the features of the trusted platform module to build a safer cloud computing environment [79]. There are three key types of entities in a TCCP, a Trusted Coordinator (TC), zero or more trusted nodes, and an External Trusted Entity (ETE). The trusted computing base of a TCCP includes a Trusted Virtual Machine Monitor (TVMM) which runs in each individual trusted node, and the whole trusted coordinator.

The assumption about the TVMM is that it protects its integrity and prevents malicious administrators from inspecting the memory space of a consumer's virtual machine. It is also assumed a certified TPM-chip is available in each trusted node in order to enforce a trusted boot process for the TVMM.

The trusted coordinator is responsible for managing (e.g., adding and removing trusted nodes) the list of nodes (trusted nodes) allowed to execute consumer's VMs. An external trusted entity owns and maintains the trusted coordinator. This entity can be seen as a certification authority in common public-key cryptography infrastructure. A trusted node must comply with two requirements, it must be located within the security perimeter and execute a trusted virtual machine monitor. We are not going to discuss the details about the migration and launch protocols used in a TCCP.

This solution is more concerned with the management of trusted nodes within a cloud platform, which includes strategies to keep the infrastructure limited to nodes executing a trusted virtual machine monitor. In this thesis, our work focuses on how to assure that a TVMM can in fact be trusted to offer the features this solution uses to maintain security in a cloud infrastructure.

Guaranteeing that only trusted nodes running a TVMM execute consumers VMs assures data integrity and confidentiality to consumers. These security properties are guaranteed through security features assumed to be present in a TVMM.The trusted computing base in a TCCP includes the trusted coordinator node and for each trusted node it includes the TVMM. The inclusion of an external TC node makes the TCB unpredictable because the TC's software stack architecture is unknown.

3.4.6 Excalibur

Excalibur is another solution targeting the offering of trusted cloud services but through the use of policy-sealed data primitives. This solution is centred on a component denominated as *monitor*, which is responsible for enforcing the policy-sealed data policy [80].

Monitor is the only component with access to the TPM primitives in order to minimize TPM's negative performance impact. The TPM primitives relevant for this solution are seal and unseal. In the seal operation the TPM takes the data to protect, and the credentials that uniquely identify the platform as input and encrypts them. To unseal (i.e., decipher) the data a platform needs to have identical credentials/configuration [29].

In a cloud infrastructure there can be one or more instances of the monitor component to guarantee fault tolerance properties. The monitor is responsible for attesting a cloud server before it is accepted as a valid destination for consumer's data. A cloud server's unique configuration (e.g., hardware and software) are used as credentials to unseal policy-sealed data. Ciphertext Policy Attribute-Based Encryption (CPABE) is used to cryptographically enforce policies in a manner that can scale, is effective, and fault tolerant [8].

The monitor component allows cloud administrators to manage the mappings between attributes and fingerprints. Therefore, it is necessary to prove to consumers that the monitor component is trustworthy and is not accepting malicious mappings. This objective is achieved through the use of a certification chain which permits the verification of mappings before accepting them.

Excalibur does not address virtualization layer problems such as not complying with the principle of least privilege and guaranteeing cloud administrator access to consumer's security sensitive data. Therefore, it is effective in assuring integrity and confidentiality if the virtualization layer offers such properties. The use of data sealing based on TPM's primitives is effective as assured by the TPM and CPABE approach.

3.4.7 TrustVisor

TrustVisor is a virtual machine monitor designed with the objective of improving the security of commodity systems. It intends to assure code and data integrity for selected parts of executing applications while keeping the trusted computing base and performance overhead to a minimum [54].

Assuring code integrity with fine granularity while keeping the performance impact to a minimum is achieved through the use of a dynamic root of trust for measurement mechanism denominated TrustVisor Root of Trust for Measurement (TRTM). A limited set of TPM features are implemented in a software *micro-TPM* that is part of TrustVisor. TRTM together with the *micro-TPM* allow external entities to perform remote attestation of code blocks executing in a TustVisor-enabled commodity system.

The capabilities offered through the combination of a micro-TPM and TRTM guarantees data integrity and execution integrity. Execution integrity refers to the execution of a code block with a set of inputs generating the expected set of outputs. TrustVisor assures the execution of designated code blocks in isolation from the operating system, untrusted applications, and system devices.

A prototype of TrustVisor was implemented on an AMD platform with support for AMD's Secure Virtual Machine technology with a trusted computing base of 6351 lines of code. The performance overhead of adding TrustVisor to a commodity system is less then 7%.

Although TrustVisor is not designed for a cloud environment, its design principles are a solid foundation for the implementation of any trustworthy system. It is expected that the trusted computing base of a virtual machine monitor that targets the cloud to be bigger than TrustVisor's but the features suggested in this work can be adapted to such platforms.

3.4.8 myTrustedCloud

In the myTrustedCloud architecture, the authors describe the use of trusted computing technology to improve on the security of a cloud provider offering infrastructure as a service solutions [103]. The integration of trustworthy services was tested with the Eucalyptus cloud computing platform [63].

The main objective of myTrustedCloud is to permit remote attestation of virtual machines and elastic block storage. This service allows cloud consumers to verify the integrity of their VMs and storage volumes. The integrity of these two components are dependent on the integrity of node controller and storage controller which are part of the Eucalyptus system. The node controller is the software responsible for managing a cloud server including the life cycle of VMs running on that same server. A storage controller offers storage and retrieval of virtual machine images and user data [63].

The integrity measurements used to verify the integrity of the components aforementioned is guaranteed through the use of a trusted boot process and an Integrity Measurement Architecture (IMA) enabled Linux kernel [77]. The functions of the trusted platform module then allow the kernel to perform integrity measurements of VM images and applications executing inside verified virtual machines.

The problem with this architecture is the extremely large trusted computing base a cloud consumer needs to trust. This is true when compared with solutions that rely only on a bare-metal virtual machine monitor to enforce security properties. This approach includes the Linux kernel in its trusted computing base, which currently has a code base of approximately seventeen million lines of code [64]. It is known that the larger the number of lines of code the more likely it is for that code to contain exploitable security vulnerabilities [58].

3.4.9 Strongly Isolated Computing Environment (SICE)

Strongly Isolated Computing Environment (SICE) is a framework that guarantees an isolated execution environment for x86 hardware platforms. SICE further reduces the trusted computing base when compared to the NoHype approach. NoHype still relies on a system manager software package running in one of the cores to manage virtual machines. In SICE, the trusted computing base is reduced to the hardware, BIOS, and System Management Mode (SMM) [4].

The authors argue that its different and smaller trusted computing base gives SICE advantages when compared to previous microhypervisors and hardware-based isolation techniques. The advantages are a smaller attack surface, compatibility with existing software systems, and feasible hardware-based isolation.

SICE resides in System Management RAM (SMRAM), and it is responsible for assuring secure initialisation, memory isolation, and integrity attestation of the isolated environ-

Solution	Confidentiality	Integrity	TCB	Cloud	Insider Threat
Terra	✘	✘	✔	✘	✘
TVDc	✘	✘	✘	✔	✘
PVI	✘	✘	✘	✔	✘
NoHype	✘	✘	✔	✔	✘
TCCP	✔	✔	✘	✔	✔
Excalibur	✔	✔	✘	✔	✔
TrustVisor	✔	✔	✔	✘	✘
myTrustedCloud	✘	✘	✘	✔	✔
SICE	✔	✔	✔	✔	✘

Table 3.2 Hardware-centred solutions summary. ✔ = guarantees/addresses; ✘ = not guarantees/addresses.

ments. An isolated environment consists of an isolated workload and a security manager. The security manager of each isolated environment prevents it from accessing memory assigned to the legacy system. SMM is used to protect the isolated environment from the legacy system.

The framework supports time-sharing and multi-core operation modes. Time-sharing consists in multiplexing hardware resources usage between legacy system and isolated environments. The multi-core mode assigns one or multiple cores to each isolated environment while the remainder of the processing resources is used to handle the legacy system. The latter reminds the approach used in NoHype.

Although the authors argue that SICE differs from microhypervisors, later they affirm that its security manager is in fact similar to the functionality of a microhypervisor. The fact that this solution is only compatible with AMD-powered machines is also a limiting factor to this approach. Assuming the security manager is not guaranteed access to the memory space of virtual machines, the hardware-enforced isolation should be enough to assure data confidentiality and integrity to cloud consumers. Otherwise, a malicious insider could take advantage of a security manager access privileges to undermine the security of consumer's data.

3.4.10 Summary

This subsection includes a table summarising how the solutions we analysed in this section related to the properties, environment, and threat we are considering in this thesis.

Table 3.2 shows how each of the solutions analysed in this section do with respect to *confidentiality*, *integrity*, *trusted computing base*, *cloud*, and *insider threat*. For the *con-*

fidentiality and *integrity* properties we verified if the solutions are concerned with enforcing these properties for consumers' data resident in virtual machines' memory space. The *trusted computing base* column indicates if a solution reduces the TCB or not. Both the *cloud* and *insider threat* show if the solution in question was develop for a cloud environment and if it consider the malicious insider threat when trying to guarantee confidentiality and integrity for consumers' data resident in a virtual machine's memory space.

Hardware-centred solutions seem to be a promising avenue of research to prevent insider threats in cloud computing environments. However, such solutions have not been tested against insider threats the way we test the major virtualization software solutions in this thesis.

3.5 Conclusions

This chapter discussed cryptography, virtualization, and hardware centred solutions to improve security in cloud systems. There are several interesting approaches introduced in the research works we analysed, e.g, *closed-box VM* concept introduced in Terra and the disaggregation of Dom0 functionality [27, 59]. These approaches are valid and can help with building strong security foundations in a cloud system. However, from all the solutions analysed only two consider insider threats from a perspective similar to ours, i.e, CloudVisor and Secure virtual machine execution [50, 106].

Analysing these research works led us to the conclusion that more work is required when looking into an insider threat that targets memory confidentiality and integrity in cloud systems. CloudVisor can guarantee such properties but at the same time it has a considerable performance impact and does not allow the development of monitoring solutions [106]. The monitoring solutions considered here are tools that can be used to detect malicious use of cloud computing resources, which means these tools need to detect malicious behaviour in virtual machines. Secure virtual machine execution can also assure memory confidentiality and integrity but it is not very versatile because it impairs the development of monitoring solutions. This solution might also create some key management challenges [50].

Considering the insights from our analysis, we argue that there is a research gap for a prevention mechanism that assures memory confidentiality and integrity that does not impair the development of monitoring solutions. Therefore, our aim is to devise a solution that guarantees memory confidentiality and integrity by enforcing the principle of least privilege in a cloud system. These two security properties need to be assured even when an insider threat is considered.

Chapter 4

Security Design Flaw in Current Virtual Machine Monitors

This chapter presents evidence supporting the research problem we address in this thesis. We study the malicious insider threat in cloud computing. Throughout this chapter we demonstrate how a malicious insider can take advantage of a design flaw in current virtualization solutions to violate the confidentiality or integrity of cloud consumers' data.

The design flaw we identified is virtual machine monitors not enforcing the principle of least privilege. Failure to enforce this principle gives cloud administrators (malicious) access to data which they should not have the right to access, e.g., a cryptographic key resident in the memory space of a virtual machine [73].

The evidence presented in this chapter consists of attacks performed against virtual machine monitors from three major providers of virtualization software solutions. We chose to demonstrate the problem with the most commonly deployed commercial solution (i.e., VMWare ESXi) and the two major players from the open source community (i.e., Xen Hypervisor and Linux KVM) [25, 61]. Showing that the problem exists in multiple vendors argues in favour of a design flaw instead of an implementation fault in a particular virtual machine monitor.

The remainder of this chapter is organised as follows. We start with a description of the adversary model considered for our attacks. After, we provide a conceptual description of the attack we use against each of the platforms and introduce the virtual machine introspection library. It is important to introduce the virtual machine introspection library because it is used in two attack scenarios. Following these two sections, we provide a section dedicated to each virtualization software. Each section contains an introduction to the virtualization software under analysis, the attacks performed against it, and the outcome of those attacks.

4.1 Adversary Model

The adversary model for this chapter is the same as the one described in Section 3.1. Insider threats are under consideration. The main assumptions to remember are:

- *Assumption 1*: An attacker can rebuild the virtual machine monitor.
- *Assumption 2*: No hardware attacks are considered.
- *Assumption 3*: An attacker can compile and execute arbitrary software within the realm of cloud management software.

4.2 Attack Concept

The idea for the attacks described in this chapter has its origins in the infamous *cold boot* attack [32]. The authors of the cold boot attack demonstrate how random access memory retains its contents for a certain amount of time after the machine is powered down. This creates an attack window that an attacker can exploit.

The cold boot attack uses simple cooling techniques to preserve the content stored in extracted random access memory modules so they can then by connected to another system for forensic analysis. The attack shows how to successfully extract cryptographic keys used by popular disk encryption solutions. The attack requires physical access to the target machine (to extract the physical random access memory modules) but does not require any special devices or materials.

Figure 4.1 depicts how each virtual machine in a virtualization environment possesses its own virtual memory address space. Illustrated in the figure as a continuous array of byte-sized elements with a total size of $M-1$. The virtual machine monitor in the picture is the software layer with complete control over the physical random access memory.

The attacks demonstrated in this chapter became possible from realising that in a virtualization environment an attacker does not require physical access to the memory of a machine, a cloud server in our case. Physical access is not required because the virtual machine monitor layer manages and has total access to the whole memory space. The fact that the virtual machines share the same physical random access memory makes it possible for an attacker to compromise the contents of the random access memory areas assigned to each virtual machine [73].

We chose to attack the Linux operating system because it is an open source operating system which makes it easier to get access to its internals and respective source code.

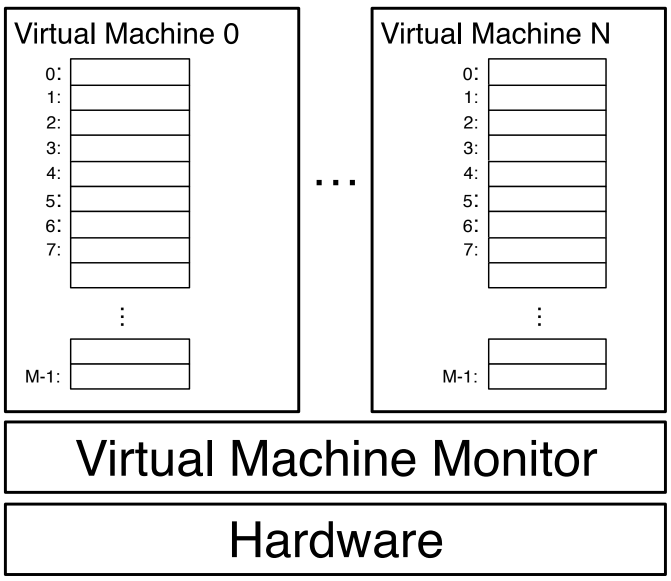


Fig. 4.1 Virtual Machines' Memory Space.

The relevance of these attacks is in demonstrating how not respecting the principle of least privilege can undermine the security of a virtualized host. The attacks are ideal to later illustrate how our approach to dealing with this design issue is effective in fixing the problem.

4.2.1 RSA Key Structure in Memory

The structure of a RSA private key representation while it is loaded in runtime memory is specified in PKCS #12 [75]. Its representation syntax can be found in the OSI networking and system aspects - Abstract Syntax Notation One (ASN.1), which is defined in a four part standard designated ITU-T X.680 [40].

The ITU-T X.680 or ISO 8824 defines standard notation for the definition of data types and values. According to the standard, a data type is a generic category of information (e.g., numeric or textual), whereas a data value is an instance of a particular data type. The ASN.1 notation is supplemented with a set of *encoding rules* that specify the value of the octets that carry application semantics, which are also known as *transfer syntax*.

The encoding rules to represent the abstract objects in binary form are described in ITU-T X.690 or ISO 8825 and include *Basic Encoding Rules (BER)*, *Canonical Encoding Rules (CER)*, and *Distinguished Encoding Rules (DER)*. The CER and DER definitions are subsets of BER and differ from each other in a set of restrictions.

The general rules for encoding are defined in ITU-T X.690 or ISO-8825-1 [39]. This

```
RSAPrivateKey ::= SEQUENCE {
    version           Version,
    modulus           INTEGER,  -- n
    publicExponent    INTEGER,  -- e
    privateExponent   INTEGER,  -- d
    prime1            INTEGER,  -- p
    prime2            INTEGER,  -- q
    exponent1         INTEGER,  -- d mod (p-1)
    exponent2         INTEGER,  -- d mod (q-1)
    coefficient       INTEGER,  -- (inverse of q) mod p
    otherPrimeInfos   OtherPrimeInfos OPTIONAL
}
```

Fig. 4.2 RSAPrivateKey ASN.1 type.

document defines that the encoding of a data value should be composed of four distinct components: *identifier octets*, *length octets*, *contents octets*, and *end-of-content octets*. These octets are order dependent and should appear in the order we just named them. The relevant component for our discussion is the *identifier octets*, which encodes the ASN.1 tag for the different types of data value, a list of such tags can be found in [38]. For example, an integer value has a tag with a hexadecimal value of *0x02*. The *identifier octet* is the starting byte of any ASN.1 encoding and is composed by three blocks: the two-bit classification, the constructed bit, and the primitive type.

The ASN.1 object identifier for a RSA private key is defined in [75], which defines object identifiers for both public and private RSA keys. Since the objective of our attack is to compromise a private key we focus on the representation of a private RSA key. Figure 4.2 illustrates the components of a binary representation for a RSA private key according to the *RSAPrivateKey* ASN.1 type.

The key search method introduced with the *cold boot* attack consists in looking for identifying features of the DER encoding. According to the authors the technique generated no false positives, which in our tests happened to be true.

The search algorithm starts by looking for the ASN.1 SEQUENCE type whose universal class tag is *0x10*, but since its encoding must be *constructed*, the SEQUENCE header byte changes to *0x30*. A constructed encoding means the *constructed bit* in the identifier octet is active. The next step is to locate the RSA version number together with the DER encoding tag of the next field. The RSA version number must be zero on almost every case except when multi-prime is used in its DER encoding. Therefore, the expected hexadecimal value for this final set of bytes is *0x02 0x01 0x00 0x02*, in which *0x02 0x01 0x00* is the RSA version and *0x02* is the type of the next field. Decomposing the RSA version bytes we can distinguish the *identifier octet* of an integer (*0x02*) that has one byte of length (*0x01*) with a

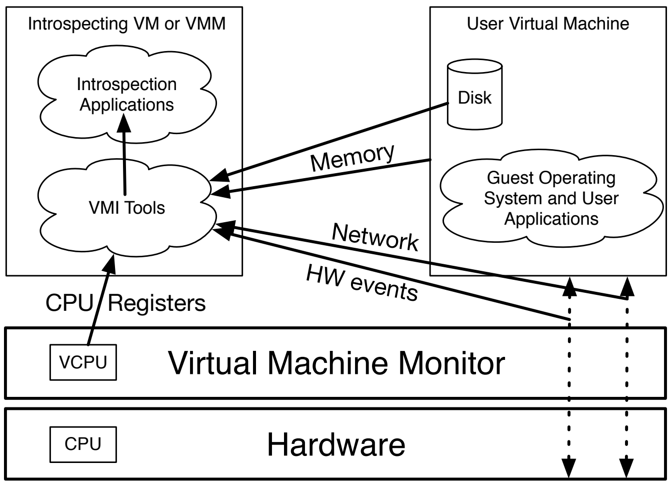


Fig. 4.3 libVMI Architecture.

value of zero (*0x00*).

We use this search algorithm to locate private RSA keys in memory dumps of virtual machines. More details can be found in Sections 4.5 and 4.6.

4.3 Virtual Machine Introspection Library

Virtual machine introspection (VMI) as previously mentioned in Subsection 2.3.5 performs introspection of virtual machines running on top of a virtualization layer. The virtual machine introspection library *(libVMI)* introduces a few techniques to obtain runtime access to the data of a virtual machine whilst it executes on top of a virtual machine monitor. This library is relevant to this section because it is used in two of our attacks.

The *libVMI* solution, previously known as *XenAccess*, offers virtual memory introspection and virtual disk monitoring capabilities [65]. The introspection takes place externally, usually from within a management virtual machine or the virtual machine monitor itself. Introspection operations are offered for the Xen hypervisor and Linux KVM platforms, as well as for memory snapshot files.

Figure 4.3 illustrates the entities involved in memory introspection when using *libVMI*. An introspection application uses features of the *libVMI* library (VMI Tools), which allow it to fetch a memory range from a virtual machine executing on top of a virtual machine monitor.

The application requests cause the library to perform a system call which is processed by a driver in the kernel executing on the virtual machine performing the introspection. This driver generates an hypercall to the virtual machine monitor. When the hypercall is complete the introspection application is granted access to the desired memory area. These steps are different when the introspection originates from within the virtual machine monitor layer. This layer has access to the whole memory space so it would simply retrieve the necessary memory areas to process.

Let us consider an example of using an introspection application to fetch a memory area that belongs to a virtual machine executing in a Xen-powered platform. First, the introspection application uses the features of *libVMI* to request a memory range from a particular virtual machine. Second, the library communicates with Xen's control library which performs a system call to obtain access to the desired memory. Third, this system call is handled by a driver in the kernel of domain-0 which generates a hypercall to guarantee access to the requested memory range.

A well known example of introspection applications is the intrusion detection solution VMWall. VMWall is a fine grained tamper-resistant process oriented firewall [84]. The isolation provided through the hypervisor shields VMWall from advanced malicious threats which can compromise the behaviour of normal application firewalls. VMWall uses virtual machine introspection to collect data from network communications, which allows it to correlate Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) traffic to block attacks from bots, worms, and backdoors.

4.3.1 Obtaining Virtual Memory Areas

Finding where the correct virtual memory areas are located in the memory range assigned to a virtual machine is a key operation in the process of attacking an application while it is executing.

The introspection library *libVMI* enables the operation of extracting information from virtual memory areas of a particular process executing in a guest virtual machine. In what follows, we provide a brief explanation of how Linux manages virtual memory for its processes.

Figure 4.4 illustrates the kernel data structures involved in the process of obtaining the virtual memory areas for a process executing in a virtual machine. Although the figure does not include in detail all the information these kernel data structures contain, it helps with understanding our explanation. The introspection application must traverse all the structures in the figure to reach the list of virtual memory areas in a particular process. Once that list is obtained it can be used to get access to the virtual memory space of a process.

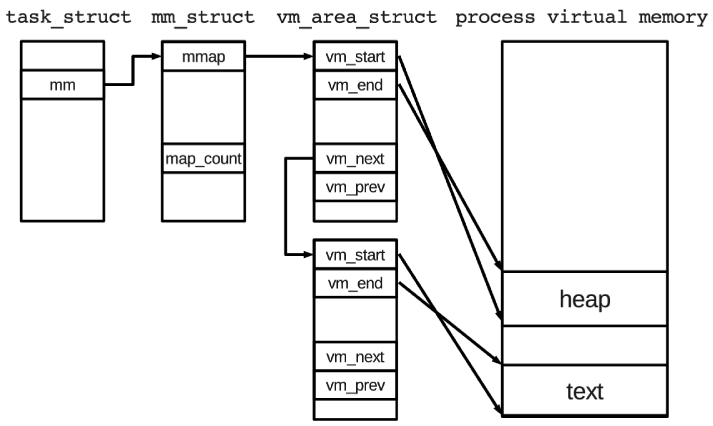


Fig. 4.4 Linux kernel data structures for virtual memory organisation.

The shapes in Figure 4.4 have different meanings, blocks represent a whole data structure, e.g., *task_struct*, the boxed names correspond to variables, and the arrows illustrate memory address pointers. For example, the *mmap* variable is a pointer variable to the list of virtual memory areas.

The Linux kernel maintains an individual task data structure (*task_struct* in the source code) for each process. This data structure is responsible for storing all the information required to run a process (e.g., the process identifier or PID and a pointer to the user stack). One of the members of a task structure, *mm*, is a pointer to a *mm_struct* data structure, which stores the current state for the virtual memory assigned to a process.

The kernel divides a process' virtual memory into *areas* to simplify its management. An *area* of virtual memory is a contiguously allocated memory range. A virtual memory area (*vm_area_struct* in the source code) keeps information about a virtual memory range allocated to a process (e.g., *vm_start*, which is a pointer to the beginning of an actual virtual memory area and *vm_next*, that is a pointer to the next *vm_area_struct* in a list of virtual memory areas). A list of *vm_area_struct* data structures, *mmap*, is found within the *mm_struct* data structure.

An introspection application requires several steps to get the list of virtual memory areas for a particular process. The first step is to traverse the kernel list of task structures and locate the process of interest. Analysing the task structure data structure in the source code makes it possible to calculate the offset to the *mm_struct* pointer. The second step uses the calculated offset to retrieve a pointer, which gives access to the data stored in the *mm_struct* data structure. The data located in the second step contains a variable that stores the size of the list of virtual memory areas and a pointer to the list that has the actual *vm_area_struct* data structures.

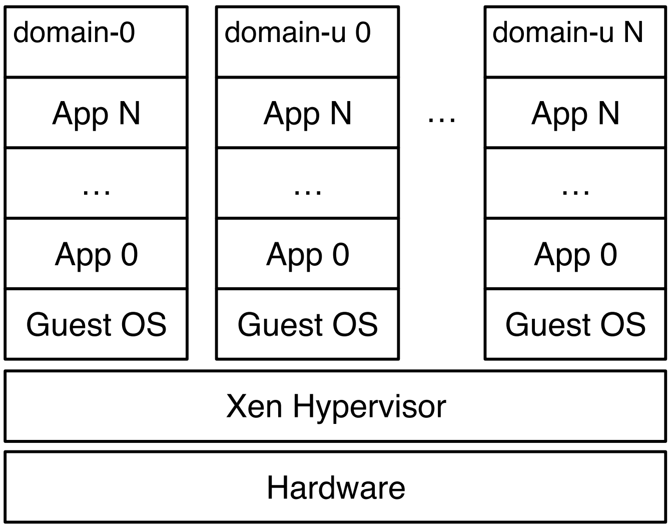


Fig. 4.5 Xen Architecture.

Access to the list of virtual memory areas makes it possible to traverse the virtual memory space of a particular process. The attack that follows uses access to this list to hunt for a desired byte pattern which can locate the intended task structure data structure and consequently its list of virtual memory areas.

4.4 Xen Hypervisor

Xen is an open source hypervisor which was initially designed to overcome the lack of support for full virtualization in the x86 architecture. It introduced paravirtualization instead of offering the traditional full virtualization approach, where unmodified operating systems can execute on top of the hypervisor [5, 104].

In full virtualization the provided virtual machine abstraction is identical to the underlying hardware allowing unmodified operating systems to run on top of it. In paravirtualization, on the contrary, the offered virtual machine abstraction is similar but it does not replicate the underlying hardware in its entirety. Therefore, an operating system requires modifications in order to operate as a paravirtualized guest. These concepts were described in detail in Section 2.3.

Figure 4.5 shows the basic architecture found in a Xen-powered cloud server. Xen is a bare-metal hypervisor, which means it executes directly on top of the hardware layer. In Xen lingo an executing virtual machine instance is referred to as domain or guest. Xen permits two different types of virtual machine instances to execute on top of it. The first type is a single virtual machine instance with elevated privileges denominated *privileged domain* or *domain-0*, also known as *dom0*. The second type can have zero or several instances running, these instances are referred to as *unprivileged domain(s)* or *domain-u*.

A Xen system is not useful without the presence of its domain-0. Xen launches the privileged domain (domain-0) at startup [105]. Domain-0 has direct access to the hardware and hosts the device drivers for all the devices in the system. The software stack responsible for the creation, termination, and configuration of unprivileged domains is also part of domain-0. The privileged domain (domain-0) needs a Xen-enabled kernel which is standard since Linux kernel version 3.0.

A domain-u is a virtual machine created through the privileged domain which executes independently in the system. An unprivileged guest can be paravirtualized or a hardware virtual machine (HVM).

Paravirtualization is efficient and does not require a central processing unit with virtualization extensions. However, a paravirtualized operating system requires modifications and it is also aware of the presence of a virtual machine monitor in the system. On the one hand, these properties make it ideal for configurations that have high performance requirements but, on the other hand, they also make it challenging to maintain or use in scenarios where applications cannot be aware that they are executing in a virtual machine, e.g., malware analysis.

A hardware virtual machine requires hardware virtualization extensions such as Intel VT and AMD-V. These hardware extensions are used to improve the performance of hardware emulation (e.g., PC Hardware or network adapter), which Xen provides through Qemu. Any unmodified operating system can run in a HVM domain. This type of domain-u is slower due to the costs associated with hardware emulation. Although they are slower than a paravirtualized virtual machine, they do not have high maintenance costs and the operating system should not be aware that it is operating in a virtualized host.

4.4.1 Inter-Virtual Machine Communication

Inter-Virtual Machine or inter-domain communication achieves higher throughput communication rates than traditional networked communications for data exchange between two virtual machines executing in the same physical host [107]. It uses a shared memory communications solution as opposed to a more traditional network communications protocol stack approach. It is important to introduce inter-domain communication because we chose to attack the virtual memory area of an application that uses this type of communication channel.

The latest release of Xen includes a new *virtual channel library* (i.e., *libvchan*) which implements high-throughput bidirectional communication channels between applications executing in different virtual machines. The channels are created in a client-server fashion using shared memory. The IDs of participating domains and a port number must be negotiated prior to initializing the communication channels. The domain acting as server is responsible for allocating the shared memory pages and determining the size of communication rings (one memory page by default). According to the information provided in Xen's source code, early testing has shown that this library can provide speeds comparable to pipes within a single Linux domain which is faster than network-based communication.

In what follows, we introduce some implementation details that are required to understand the attack we performed against Xen's inter-domain communication library. When an application executing in an unprivileged domain acts as the server in establishing an inter-domain communication channel, it requires a *libxenvchan* data structure in order to properly interact with the library.

Listing 4.1 *virtual channel library* data structures

```
struct ring_shared{
        uint32_t cons, prod;
};
struct libxenvchan_ring{
        struct ring_shared* sh;
        void* buffer;
        int order;
};
struct libxenvchan{
        union{
                xc_gntshr *gntshr
                xc_gnttab *gnttab;
        };
        struct vchan_interface *ring;
        xc_evtchn *event;
        uint32_t event_port;
        int is_server:1;
        int server_persist:1;
        int blocking:1;
```

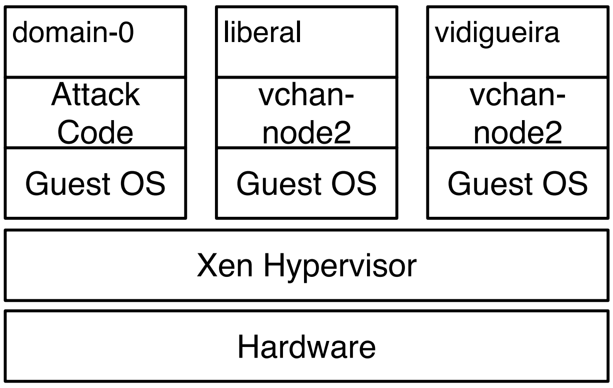


Fig. 4.6 Xen Test Environment.

```
20         struct libxenvchan_ring read, write;
21 };
```

Listing 4.1 shows a data structure which contains critical information in the establishment of inter-domain communications. The data stored using such structures informs if the application is a server, which port number is in use to communicate events, and it also contains pointers to the shared memory area. This data structure creates identifiable data patterns in memory making it possible for an attacker to locate and extract the information it contains.

One of those data patterns consists of the last six members of the data structure (line 16–line 20), starting from the variable *event_port* through to *struct libxenvchan_ring write*. The value of *event_port* can be obtained using the management tool chain available in domain-0. Possible combinations for the bit fields *is_server*, *server_persist* and *blocking* members are 2^3, but from the source code it is easy to figure out that a server application only uses the *is_server* bit field. The *read* and *write* variables are *libxenvchan_ring* data structures and contain two pointers and the variable *order*, which sets the size of the ring. The *order* is another predictable value, it is either ten or eleven. Attack code to locate this data pattern in memory can be implemented using this knowledge, such an attack is discussed in Subsection 4.4.3.

4.4.2 Test Environment

The test environment comprised a single desktop. In terms of hardware configuration the server includes an Intel Core i7-870 64-bit CPU and 4GB of main memory. The server installation consisted of Xen 4.2 unstable with a Fedora 16 domain-0 running a 64- bit

Linux kernel, version 3.1.5-1. The first virtual machine was running a 32-bit Linux kernel with physical address extension (PAE), version 3.3.2-6 for Fedora 16. The second virtual machines' operating system was also Fedora 16 but using a 64-bit Linux kernel with version 3.3.2-6.

Figure 4.6 depicts the test environment used to attack consumers' data when virtual machines execute on top of Xen. In this scenario we have two unprivileged domains with hostnames *liberal* and *vidigueira*.

The *liberal* virtual machine runs a Fedora 32-bit Linux kernel (3.3.2-6) with PAE. The *vchan-node2* application is executed so *liberal* can act as a server in an inter-domain communication channel with *vidigueira*. The purpose of this communication channel is the exchange of text messages between client and server. The *vidigueira* virtual machine uses *vchan-node2* to connect as a client to the instance executing in *liberal*.

4.4.3 Memory Confidentiality and Integrity

The attack described in this subsection to test memory confidentiality and integrity in a Xen-powered platform is an advanced attack that takes advantage of data security vulnerabilities to prove the validity of our solution. We explain later how this attack against Xen's virtual channel library can be used to illustrate how our solution prevents this type of attack.

The attack can be divided into two main steps. First, it finds the *libxenvchan* data structure pattern in the memory space of a target process. Second, it uses the captured information to poll the shared memory area in regular intervals in order to read the data exchanged between the domains using the communication channel.

To explain the attack we need an example scenario. Consider the communication channel established between the unprivileged domains *liberal* and *vidigueira* from Figure 4.6. The unprivileged domain *liberal* acts as a server and has the following values for the data structures in listing 4.1:

- *event_port*: **0x17**.
- *is_server*: **0x1**.
- *read.order*: **0xA**.
- *write.order*: **0xB**.

The values assumed here were valid after server reboots and even after Xen was recompiled. The order variable for a ring is found in the *libxenvchan_ring* data structure (Listing

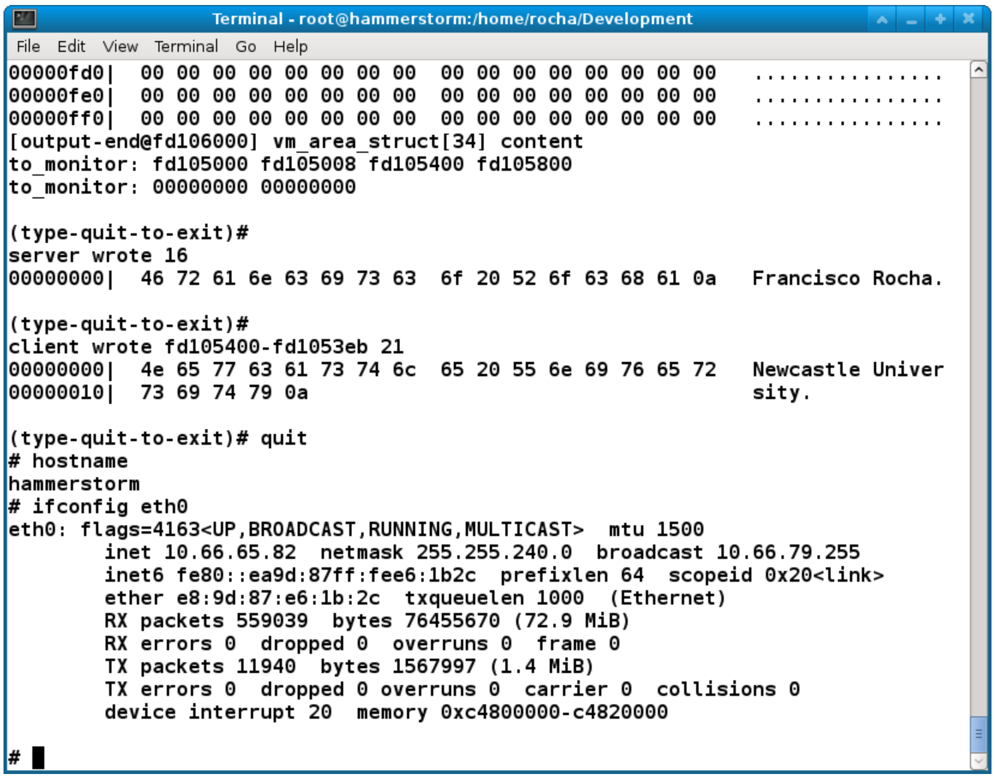
4.1: line 7). The variables (*sh* and *buffer*) before variable *order* are two pointers (Listing 4.1: line 5-6). Assume a 32-bits operating system, which means four bytes long pointers.

The attack code verifies a set of conditions before concluding it found a valid data pattern for a *libxenvchan* data structure. The list that follows enumerates the conditions that need to be true. The code only check the next condition if the previous one is valid. The attack code looks for the following conditions in the virtual memory areas assigned to the process of *vchan-node2* executing in *liberal*:

1. Locate the four byte representation for **0x17** (*event_port*).
2. Check if the next four bytes contain *0x1*, which means the bit field *is_server* is set.
3. The bit field variables (Listing 4.1: line 17-19) only require four bytes of memory. Therefore, the offsets to **0xA** (*read.order*) and **0xB** (*write.order*) are 12 and 24 bytes, respectively. To obtain twelve we add the bit fields (4 bytes) with the *read.sh* (4 bytes) and *read.buffer* (4 bytes) pointers. The offset 24 comes from adding *read.order* (4 bytes), *write.sh* (4 bytes), and *write.buffer* (4 bytes) to the previous 12 bytes.
4. If all the previous conditions are valid, it has a lock on the expected pattern for a *libxenvchan* data structure.
5. In the event that a *libxenvchan* data structure is not detected, the attack code does not find the required byte pattern in memory and exits without setting up any memory sniffing mechanisms.

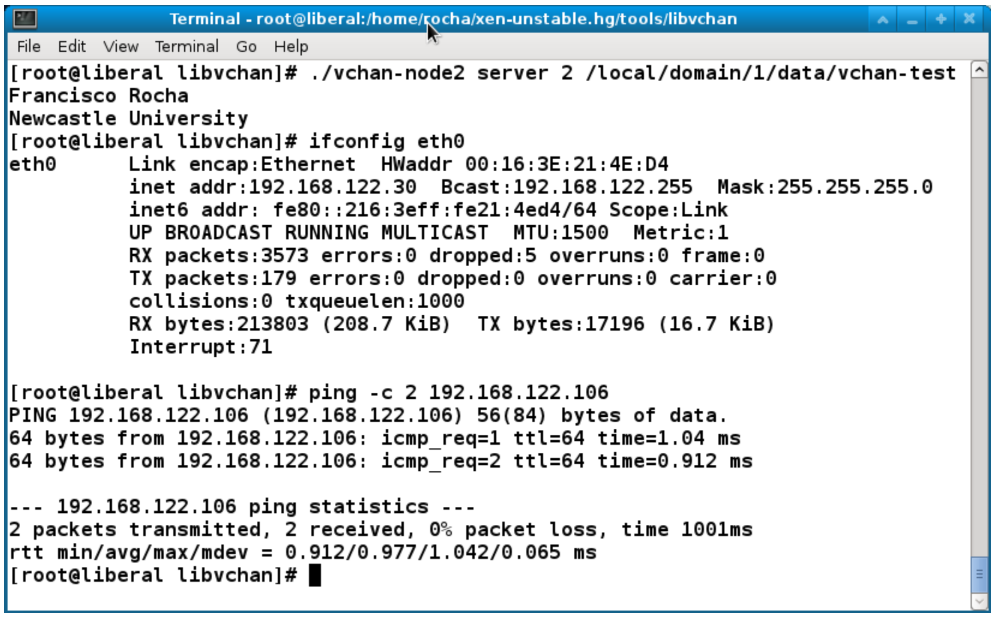
After locating the desired data pattern it reads the whole data structure and extracts the pointers to the shared memory area (i.e., variables *ring* and *buffer* in Listing 4.1 lines 14 and 6, respectively.) and offsets into the ring (i.e., field *shr* in Listing 4.1 line 5 for both variables *read* and *write*). From our experiments, we have observed that this searching stage only needs to be performed once because every time the server application launches, it ends up reusing the same port for the event channel and the same memory locations for the shared space. This is true even after rebooting the server or rebuilding Xen.

When the location and offsets are known, a thread is launched to poll the memory locations every ten milliseconds. Figure 4.7 illustrates the output of the attack code program. When the last read value is inferior to the new value, the difference gives the number of bytes written to the buffer (i.e., either read or write) associated with an offset. This allows the code to extract the data exchanged between a client and server that are using Xen's virtual channel library. In the figure we can see that the server wrote 16 bytes of data whereas the client wrote 21 bytes.


Terminal - root@hammerstorm:/home/rocha/Development
File Edit View Terminal Go Help

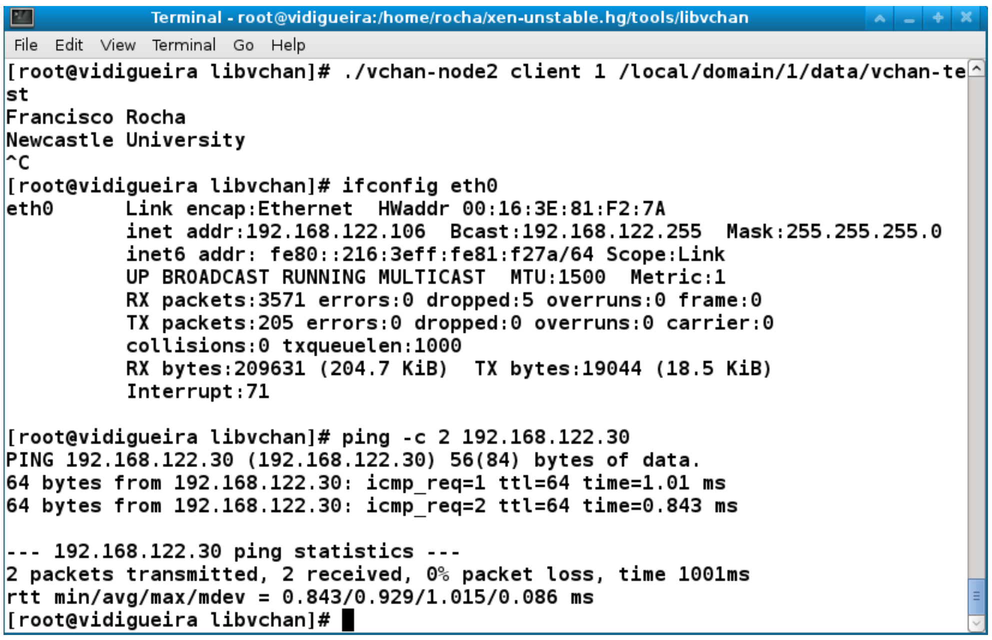
```
00000fd0|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000fe0|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
00000ff0|  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
[output-end@fd106000] vm_area_struct[34] content
to_monitor: fd105000 fd105008 fd105400 fd105800
to_monitor: 00000000 00000000

(type-quit-to-exit)#
server wrote 16
00000000|  46 72 61 6e 63 69 73 63  6f 20 52 6f 63 68 61 0a   Francisco Rocha.

(type-quit-to-exit)#
client wrote fd105400-fd1053eb 21
00000000|  4e 65 77 63 61 73 74 6c  65 20 55 6e 69 76 65 72   Newcastle Univer
00000010|  73 69 74 79 0a                                     sity.

(type-quit-to-exit)# quit
# hostname
hammerstorm
# ifconfig eth0
eth0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.66.65.82  netmask 255.255.240.0  broadcast 10.66.79.255
        inet6 fe80::ea9d:87ff:fee6:1b2c  prefixlen 64  scopeid 0x20<link>
        ether e8:9d:87:e6:1b:2c  txqueuelen 1000  (Ethernet)
        RX packets 559039  bytes 76455670 (72.9 MiB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 11940  bytes 1567997 (1.4 MiB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
        device interrupt 20  memory 0xc4800000-c4820000

# 
```

Fig. 4.7 Xen attack code executing.


Terminal - root@liberal:/home/rocha/xen-unstable.hg/tools/libvchan
File Edit View Terminal Go Help

```
[root@liberal libvchan]# ./vchan-node2 server 2 /local/domain/1/data/vchan-test
Francisco Rocha
Newcastle University
[root@liberal libvchan]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:16:3E:21:4E:D4
          inet addr:192.168.122.30  Bcast:192.168.122.255  Mask:255.255.255.0
          inet6 addr: fe80::216:3eff:fe21:4ed4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3573 errors:0 dropped:5 overruns:0 frame:0
          TX packets:179 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:213803 (208.7 KiB)  TX bytes:17196 (16.7 KiB)
          Interrupt:71

[root@liberal libvchan]# ping -c 2 192.168.122.106
PING 192.168.122.106 (192.168.122.106) 56(84) bytes of data.
64 bytes from 192.168.122.106: icmp_req=1 ttl=64 time=1.04 ms
64 bytes from 192.168.122.106: icmp_req=2 ttl=64 time=0.912 ms

--- 192.168.122.106 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.912/0.977/1.042/0.065 ms
[root@liberal libvchan]# 
```

Fig. 4.8 Server virtual machine.


Terminal - root@vidigueira:/home/rocha/xen-unstable.hg/tools/libvchan
File Edit View Terminal Go Help

```
[root@vidigueira libvchan]# ./vchan-node2 client 1 /local/domain/1/data/vchan-te
st
Francisco Rocha
Newcastle University
^C
[root@vidigueira libvchan]# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:16:3E:81:F2:7A
          inet addr:192.168.122.106  Bcast:192.168.122.255  Mask:255.255.255.0
          inet6 addr: fe80::216:3eff:fe81:f27a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:3571 errors:0 dropped:5 overruns:0 frame:0
          TX packets:205 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:209631 (204.7 KiB)  TX bytes:19044 (18.5 KiB)
          Interrupt:71

[root@vidigueira libvchan]# ping -c 2 192.168.122.30
PING 192.168.122.30 (192.168.122.30) 56(84) bytes of data.
64 bytes from 192.168.122.30: icmp_req=1 ttl=64 time=1.01 ms
64 bytes from 192.168.122.30: icmp_req=2 ttl=64 time=0.843 ms

--- 192.168.122.30 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1001ms
rtt min/avg/max/mdev = 0.843/0.929/1.015/0.086 ms
[root@vidigueira libvchan]# █
```

Fig. 4.9 Client virtual machine.

Figure 4.8 shows the command to execute the application *vchan-node2* as a server. In this case it sets up as a server accepting connections from the client virtual machine with ID equal to two (*vidigueira*), using the specified virtual channel. We also decided to print the IP address for the server virtual machine (*liberal*) and the result of sending ping requests to the client virtual machine (*vidigueira*). This is just to prove that they are actual virtual machines executing on the same network. The exchanged messages captured by the attack code are also visible in this figure confirming that the attack code read the correct memory locations.

The client virtual machine's command line is shown in Figure 4.9. The *vchan-node2* command connects as a client to the server with ID equal to one (*liberal*), using the specified virtual channel. This figure displays the IP address for *vidigueira* and the output of a ping command directed at the server virtual machine (*liberal*). Again, the objective is just to show that these are separate virtual machines executing in the same network. The figure also confirms that the messages captured by the attack code are the correct ones.

4.4.4 Conclusions

The success of our attack demonstrates that a Xen-powered platform is susceptible to the insider threat. It is clear that consumers' data resident in a virtual machine's virtual memory area can be compromised by a malicious insider. This proves that Xen is not enforcing the principle of least privilege regarding the data a cloud administrator needs to access in order to perform the necessary administrative tasks.

The attack we demonstrate against the Xen hypervisor is more complex than the ones that follow because we need it to prove the effectiveness of the solution we later devise and implement using Xen. This attack targets a single memory page which is typically the basic unit for memory allocation as mention in Section 5.1. This level of granularity later allows us to reason about the security of our approach.

4.5 Linux KVM

Linux Kernel-based Virtual Machine (KVM) is an alternative approach to the more traditional VMM-based solutions such as Xen [68]. The most common virtualization approaches resort to a thin virtualization layer which performs basic scheduling and memory management. A virtual machine monitor typically relies on a privileged virtual machine (e.g., domain-0) to perform management and input/output tasks.

A few design decisions dictate how the Linux KVM approach diverges from other open source solutions. Linux KVM was designed as a loadable kernel module which transforms the Linux kernel into a bare metal hypervisor. Two key design aspects helped KVM mature into a stable and high performance hypervisor.

First, a KVM hypervisor requires CPUs that include hardware assisted virtualization technology such as Intel VT-x and AMD-V. KVM uses the features of the virtualization enabled hardware to virtualize the CPU, this allows it remove the need to support legacy hardware or perform modifications to guest operating systems.

Second, instead of reimplementing the core functionalities of a hypervisor which include, for example, a memory manager, a process scheduler, and a network stack. KVM takes advantage of the Linux kernel already including a solid implementation of the core features required by a hypervisor. This becomes a strong argument when you think of a hypervisor as a special purpose operating system that executes virtual machines instead of applications. However, the trusted computing base of the Linux kernel is considerably larger than that of hypervisor solutions implemented from scratch. This fact is important when considering the security of a platform.

Figure 4.10 depicts the typical architecture for a platform that uses Linux KVM as its virtualization layer. Using KVM means that virtual machines and virtual CPUs are no more than a regular Linux process executing alongside traditional Linux applications. This means that virtual machines are integrated in the Linux ecosystem and can take advantage of all

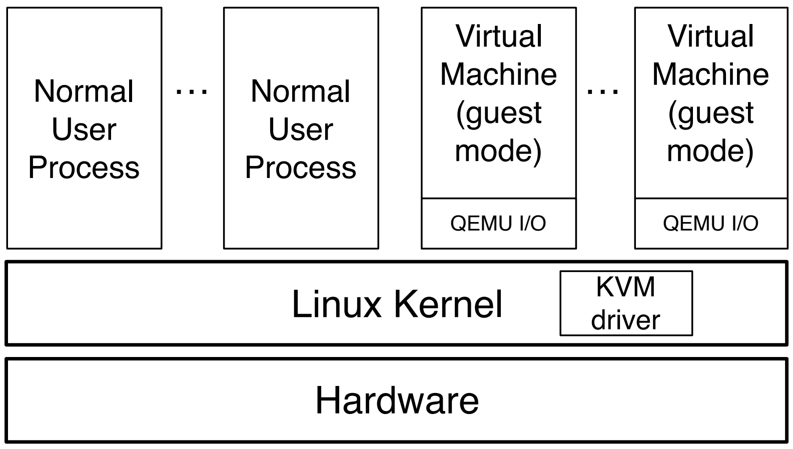


Fig. 4.10 Linux Kernel-based Virtual Machine Architecture.

the features of the Linux kernel.

Linux KVM uses a customized version of QEMU to handle device emulation [23]. The machine emulator features of QEMU are used to provide an emulated BIOS, PCI bus, USB bus, and a standard set of devices such as disk controllers and network cards. Device emulation is required to provide *full virtualization*. A *fully virtualized* virtual machine communicates with QEMU emulated devices that compose a complete abstract physical machine, which leads the virtual machine to believe it is executing on real physical hardware.

KVM offers full virtualization but it also supports a form of *hybrid virtualization* that relies on an optimized I/O interface in the place of emulated devices to guarantee high performance for input/output network and block devices. Hybrid virtualization uses the virtualization standard for network and disk device drivers *Virtio* [67]. Using this approach means the operating systems running in the virtual machines get most of the performance benefits of paravirtualizion, but they are also aware that they are executing in a virtualized platform.

Since the security of a virtual machine's memory space is the topic under analysis in this thesis, it is important to mention how KVM manages such resources. This is one of the mechanisms that KVM inherits from the Linux kernel. The memory of a virtual machine is stored just like any other Linux process and can take advantage of the memory management features Linux offers such as swapping and backup.

Linux's stable support for Non-Uniform Memory Access (NUMA) offers virtual machines efficient access to large amounts of memory. A Linux kernel feature called Kernel Same-page Merging (KSM) scans memory assigned to virtual machines and merges identical memory pages into a single shared copy. In the event of a virtual machine requesting

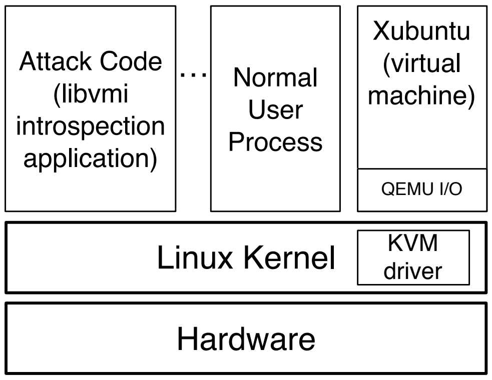

Attack Code (libvmi introspection application)
...
Normal User Process
Xubuntu (virtual machine)
QEMU I/O
Linux Kernel
KVM driver
Hardware

Fig. 4.11 Linux KVM Test Environment.

changes to such page, it receives its own private copy.

4.5.1 Test Environment

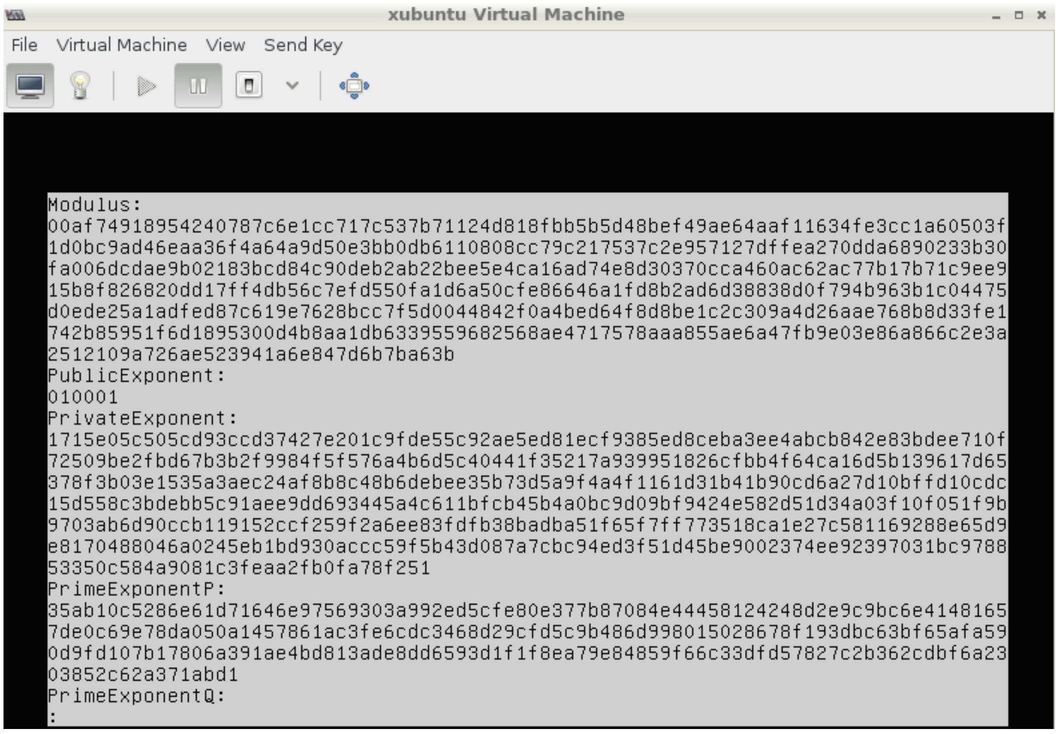
The test environment comprised a single desktop. In terms of hardware configuration the server includes an Intel Core i7-870 64-bit CPU and 4GB of main memory. The host/server operating system installed was Fedora 18. We used a 64-bit Linux kernel with version 3.6.10-4 for the server. The qemu-KVM version was 1.2.2. A single virtual machine was set up using Xubuntu 12.04. We chose a 64-bit Linux kernel with version 3.2.0-29 for the virtual machine.

Figure 4.11 illustrates the test environment used to test our attack against consumers' data resident in a virtual machine executing on top of Linux KVM. Our attack code executes as a normal system process and creates a memory dump file for a specific application running in the Xubuntu virtual machine. We describe the attack in detail in the section that follows.

4.5.2 Memory confidentiality and Integrity

The attack we devised to test memory confidentiality and integrity in a Linux KVM platform can be divided in two step. First, we developed attack code in the form of a malicious introspection application that retrieved the memory range assigned to a Java application when it was executing in our Xubuntu virtual machine. Second, the memory dump of the Java application was searched for the typical pattern a RSA private key exhibits while loaded in runtime memory [32].

In more detail, the attack code consists of an introspection application that uses the features of the virtual machine introspection library *libVMI* to identify and extract the virtual



```
Modulus:
00af74918954240787c6e1cc717c537b71124d818fbb5b5d48bef49ae64aaf11634fe3cc1a60503f
1d0bc9ad46eaa36f4a64a9d50e3bb0db6110808cc79c217537c2e957127dffea270dda6890233b30
fa006dcdae9b02183bcd84c90deb2ab22bee5e4ca16ad74e8d30370cca460ac62ac77b17b71c9ee9
15b8f826820dd17ff4db56c7efd550fa1d6a50cfe86646a1fd8b2ad6d38838d0f794b963b1c04475
d0ede25a1adfed87c619e7628bcc7f5d0044842f0a4bed64f8d8be1c2c309a4d26aae768b8d33fe1
742b85951f6d1895300d4b8aa1db6339559682568ae4717578aaa855ae6a47fb9e03e86a866c2e3a
2512109a726ae523941a6e847d6b7ba63b
PublicExponent:
010001
PrivateExponent:
1715e05c505cd93ccd37427e201c9fde55c92ae5ed81ecf9385ed8ceba3ee4abcb842e83bdee710f
72509be2fbd67b3b2f9984f5f576a4b6d5c40441f35217a939951826cfbb4f64ca16d5b139617d65
378f3b03e1535a3aec24af8b8c48b6debee35b73d5a9f4a4f1161d31b41b90cd6a27d10bffd10cdc
15d558c3bdebb5c91aee9dd693445a4c611bfcb45b4a0bc9d09bf9424e582d51d34a03f10f051f9b
9703ab6d90ccb119152ccf259f2a6ee83fdfb38badba51f65f7ff773518ca1e27c581169288e65d9
e8170488046a0245eb1bd930accc59f5b43d087a7cbc94ed3f51d45be9002374ee92397031bc9788
53350c584a9081c3feaa2fb0fa78f251
PrimeExponentP:
35ab10c5286e61d71646e97569303a992ed5cfe80e377b87084e44458124248d2e9c9bc6e4148165
7de0c69e78da050a1457861ac3fe6cdc3468d29cfd5c9b486d998015028678f193dbc63bf65afa59
0d9fd107b17806a391ae4bd813ade8dd6593d1f1f8ea79e84859f66c33dfd57827c2b362cdbf6a23
03852c62a371abd1
PrimeExponentQ:
:
```

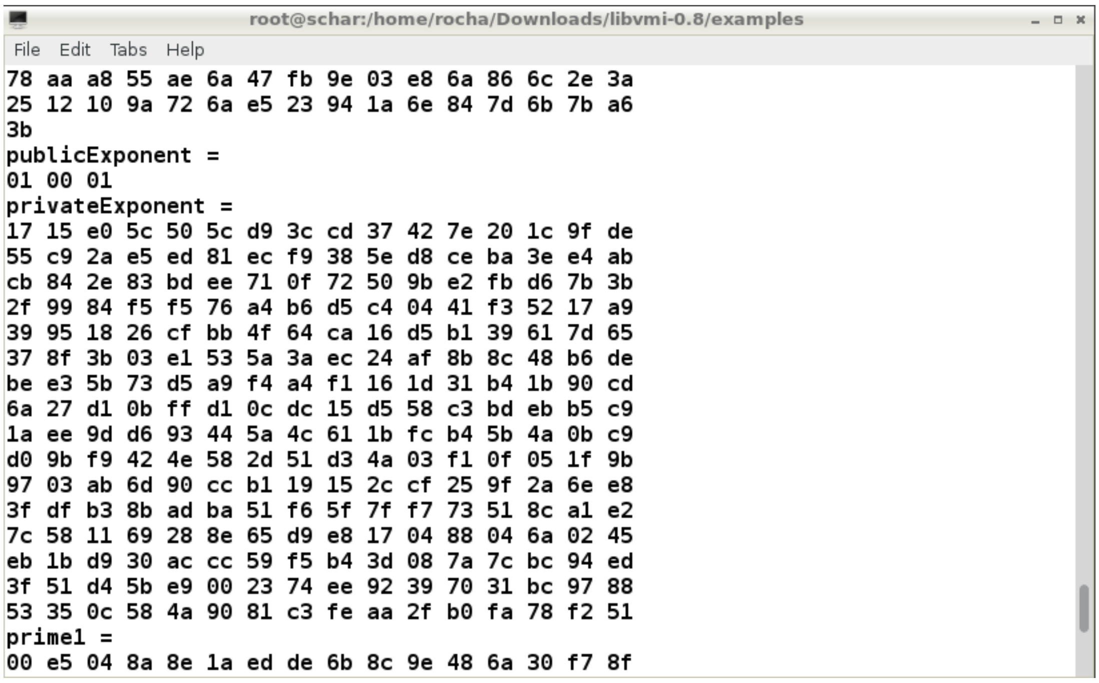
Fig. 4.12 Generated private RSA key.

memory areas associate with the Linux process created for the Java application we implemented. The Java application simply loads a RSA private key into memory, prints the various members that compose the key, and then enters an infinite iteration until the process is killed. When the introspection application captures its virtual memory areas it creates a memory dump over which we can run the key search algorithm introduced with the *cold boot* attack.

We provide two figures to help us demonstrate that we compromised the private RSA key generate by the Java application executed in the Xubuntu virtual machine. Figure 4.12 displays the value for the private exponent the application would use in decryption operations. The same value can be seen in Figure 4.13. This value was captured using the key search algorithm suggested in the *cold boot* attack.

4.5.3 Conclusions

Compromising a consumer's private RSA key demonstrates that our attack was successful. The attack demonstrated here clearly violates the confidentiality of a consumer's data. A cloud administrator does not require access to a private RSA key a security sensitive application uses while executing in a consumer owned virtual machine.



```
78 aa a8 55 ae 6a 47 fb 9e 03 e8 6a 86 6c 2e 3a
25 12 10 9a 72 6a e5 23 94 1a 6e 84 7d 6b 7b a6
3b
publicExponent =
01 00 01
privateExponent =
17 15 e0 5c 50 5c d9 3c cd 37 42 7e 20 1c 9f de
55 c9 2a e5 ed 81 ec f9 38 5e d8 ce ba 3e e4 ab
cb 84 2e 83 bd ee 71 0f 72 50 9b e2 fb d6 7b 3b
2f 99 84 f5 f5 76 a4 b6 d5 c4 04 41 f3 52 17 a9
39 95 18 26 cf bb 4f 64 ca 16 d5 b1 39 61 7d 65
37 8f 3b 03 e1 53 5a 3a ec 24 af 8b 8c 48 b6 de
be e3 5b 73 d5 a9 f4 a4 f1 16 1d 31 b4 1b 90 cd
6a 27 d1 0b ff d1 0c dc 15 d5 58 c3 bd eb b5 c9
1a ee 9d d6 93 44 5a 4c 61 1b fc b4 5b 4a 0b c9
d0 9b f9 42 4e 58 2d 51 d3 4a 03 f1 0f 05 1f 9b
97 03 ab 6d 90 cc b1 19 15 2c cf 25 9f 2a 6e e8
3f df b3 8b ad ba 51 f6 5f 7f f7 73 51 8c a1 e2
7c 58 11 69 28 8e 65 d9 e8 17 04 88 04 6a 02 45
eb 1b d9 30 ac cc 59 f5 b4 3d 08 7a 7c bc 94 ed
3f 51 d4 5b e9 00 23 74 ee 92 39 70 31 bc 97 88
53 35 0c 58 4a 90 81 c3 fe aa 2f b0 fa 78 f2 51
prime1 =
00 e5 04 8a 8e 1a ed de 6b 8c 9e 48 6a 30 f7 8f
```

Fig. 4.13 Compromised private RSA key.

The success of this attack means that a Linux KVM platform does not enforce the principle of least privilege when it comes to the capabilities entrusted to system administrators. Therefore, it is vulnerable to the insider threat just like the Xen hypervisor.

4.6 VMWare ESXi

VWare ESXi introduces VMWare's next-generation of hypervisor software solutions. This new generation architecture breaks with the old dependency on a general purpose operating system. Past architectures had a Linux based console operating system (COS), or service console, which used to support management and monitoring services. In this new architecture, the hypervisor's memory footprint was reduced to less than 32MB due to the removal of the service console.

The service console used to be the principal management interface for the virtualized host. In previous architectures the service console was primarily used to deploy VMWare management agents and other infrastructure service agents (e.g. name service, logging, etc). Remote command line interfaces were added to the most recent ESXi architecture in order to replace the outdated Linux-based service console.

Figure 4.14 shows the organisation of the main components in the ESXi architecture.

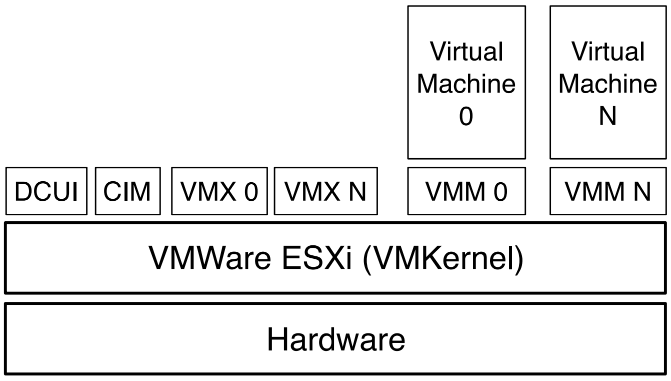


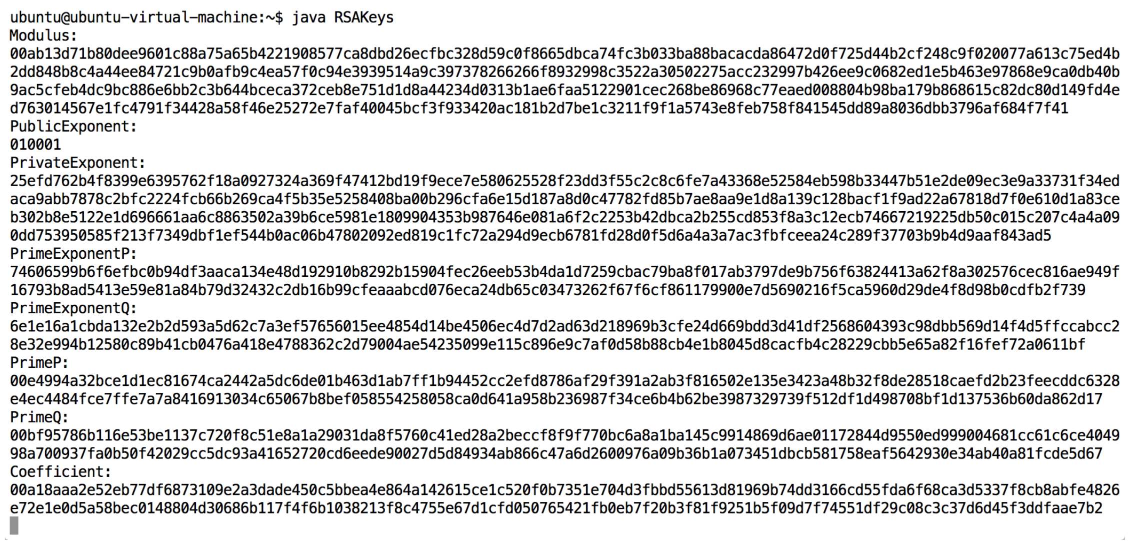
Fig. 4.14 Architecture for VMWare ESXi.

The key component in this architecture is the VMKernel operating system which provides the primitives required to execute all the processes in the system. Example processes include management applications and virtual machines. The inclusion of all device drivers means that VMKernel controls all the hardware devices on the server. Therefore, it is responsible for managing resources for applications.

Some examples of the main processes executing on top of VMKernel are the Direct Console User Interface (DCUI), Virtual Machine Monitor (VMM), and the Common Information Model (CIM) system. The DCUI consists of low-level management and configuration interfaces which are important in performing an initial basic configuration. These interfaces are available through the server console. A virtual machine monitor is the execution environment for a virtual machine, and each VMM process as a helper process denominated VMX. A pair consisting of a VMM and a VMX is assigned to each virtual machine. The CIM system comprises a set of standard application programming interfaces that enable remote applications to execute hardware-level management functions.

Memory management in VMWare ESXi is similar to what is done for Xen. When a virtual machine requires the *allocation* of memory space, the hypervisor's memory management functionality behaves just like a traditional operating system would and assigns the virtual machine a set of the total number of memory pages in the host.

The operating system that runs inside the virtual machine then claims all that memory using the traditional virtual memory approach to manage it. However, in this configuration an extra level of address translation is required because multiple virtual machines execute in the same physical host sharing its memory resources. Detailed descriptions of virtual memory and memory virtualization are available in Chapter 5.



```
ubuntu@ubuntu-virtual-machine:~$ java RSAKeys
Modulus:
00ab13d71b80dee9601c88a75a65b4221908577ca8dbd26ecfbc328d59c0f8665dbca74fc3b033ba88bacacda86472d0f725d44b2cf248c9f020077a613c75ed4b
2dd848b8c4a44ee84721c9b0afb9c4ea57f0c94e3939514a9c397378266266f8932998c3522a30502275acc232997b426ee9c0682ed1e5b463e97868e9ca0db40b
9ac5cfeb4dc9bc886e6bb2c3b644bceca372ceb8e751d1d8a44234d0313b1ae6faa5122901cec268be86968c77eaed008804b98ba179b868615c82dc80d149fd4e
d763014567e1fc4791f34428a58f46e25272e7faf40045bcf3f933420ac181b2d7be1c3211f9f1a5743e8feb758f841545dd89a8036dbb3796af684f7f41
PublicExponent:
010001
PrivateExponent:
25efd762b4f8399e6395762f18a0927324a369f47412bd19f9ece7e580625528f23dd3f55c2c8c6fe7a43368e52584eb598b33447b51e2de09ec3e9a33731f34ed
aca9abb7878c2bfc2224fcb66b269ca4f5b35e5258408ba00b296cfa6e15d187a8d0c47782fd85b7ae8aa9e1d8a139c128bacf1f9ad22a67818d7f0e610d1a83ce
b302b8e5122e1d696661aa6c8863502a39b6ce5981e1809904353b987646e081a6f2c2253b42dbca2b255cd853f8a3c12ecb74667219225db50c015c207c4a4a09
0dd753950585f213f7349dbf1ef544b0ac06b47802092ed819c1fc72a294d9ecb6781fd28d0f5d6a4a3a7ac3fbfceea24c289f37703b9b4d9aaf843ad5
PrimeExponentP:
74606599b6f6efbc0b94df3aaca134e48d192910b8292b15904fec26eeb53b4da1d7259cbac79ba8f017ab3797de9b756f63824413a62f8a302576cec816ae949f
16793b8ad5413e59e81a84b79d32432c2db16b99cfeaaabcd076eca24db65c03473262f67f6cf861179900e7d5690216f5ca5960d29de4f8d98b0cdfb2f739
PrimeExponentQ:
6e1e16a1cbda132e2b2d593a5d62c7a3ef57656015ee4854d14be4506ec4d7d2ad63d218969b3cfe24d669bdd3d41df2568604393c98dbb569d14f4d5ffccabcc2
8e32e994b12580c89b41cb0476a418e4788362c2d79004ae54235099e115c896e9c7af0d58b88cb4e1b8045d8cacfb4c28229cbb5e65a82f16fef72a0611bf
PrimeP:
00e4994a32bce1d1ec81674ca2442a5dc6de01b463d1ab7ff1b94452cc2efd8786af29f391a2ab3f816502e135e3423a48b32f8de28518caefd2b23feecddc6328
e4ec4484fce7ffe7a7a8416913034c65067b8bef058554258058ca0d641a958b236987f34ce6b4b62be3987329739f512df1d498708bf1d137536b60da862d17
PrimeQ:
00bf95786b116e53be1137c720f8c51e8a1a29031da8f5760c41ed28a2beccf8f9f770bc6a8a1ba145c9914869d6ae01172844d9550ed999004681cc61c6ce4049
98a700937fa0b50f42029cc5dc93a41652720cd6eede90027d5d84934ab866c47a6d2600976a09b36b1a073451dbcb581758eaf5642930e34ab40a81fcde5d67
Coefficient:
00a18aaa2e52eb77df6873109e2a3dade450c5bbea4e864a142615ce1c520f0b7351e704d3fbbd55613d81969b74dd3166cd55fda6f68ca3d5337f8cb8abfe4826
e72e1e0d5a58bec0148804d30686b117f4f6b1038213f8c4755e67d1cfd050765421fb0eb7f20b3f81f9251b5f09d7f74551df29c08c3c37d6d45f3ddfaae7b2
```

Fig. 4.15 Generated private RSA key.

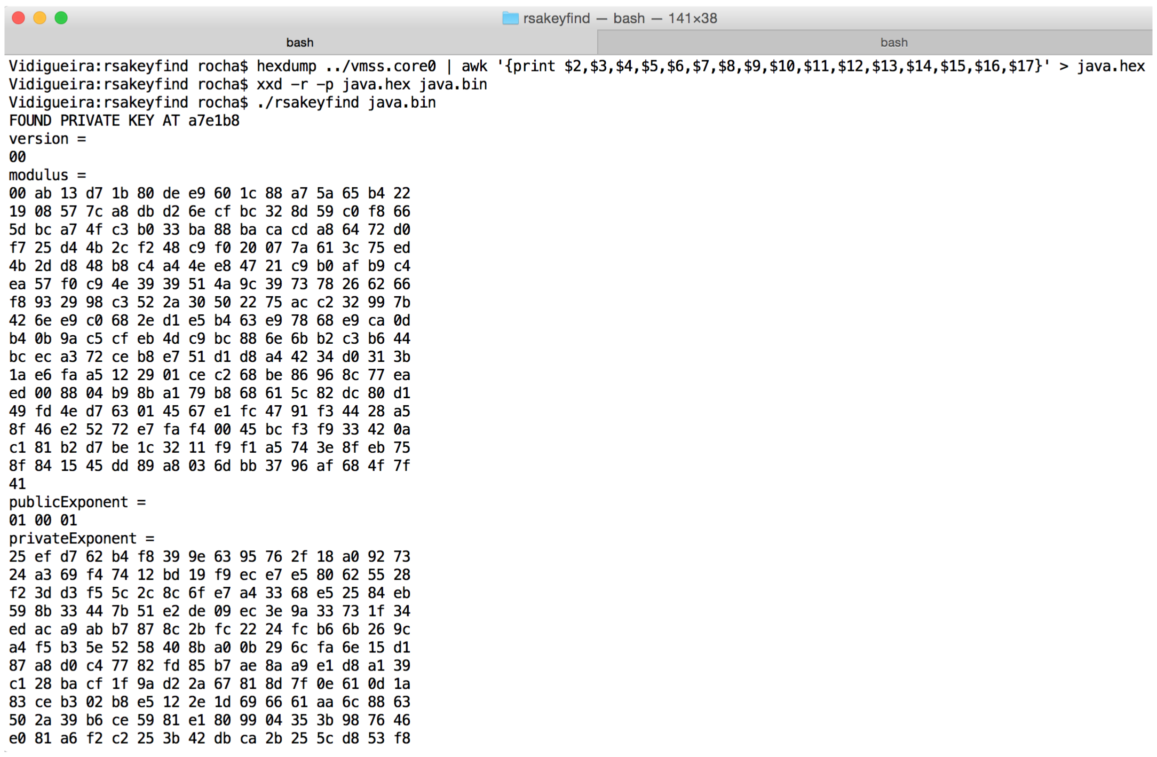
4.6.1 Test Environment

The host/server we used to configure VMWare ESXi was a Dell PowerEdge R720 server with an Intel Xeon E5-2620 and 256GB of memory. We installed VMWare ESXi 5.5.0 Update 1 which was the latest version at the time we performed our security tests. VMWare's vSphere Web Client was the solution we chose as management software for our virtualized server. Using the web client we installed and configured a virtual machine with the Ubuntu 13.10 Linux operating system. The kernel was 64-bit with version 3.11.0-18.

Figure 4.15 shows the values for the generated private RSA key. This private RSA key is part of the public-private RSA key pair a Java application generated. The application was executing in the Ubuntu virtual machine. In the figure it is possible to verify that the user *ubuntu* is logged in at the host named *ubuntu-virtual-machine* (the virtual machine's hostname).

4.6.2 Memory Confidentiality and Integrity

To test memory confidentiality and integrity in this environment we chose to reproduce one of the memory attacks described in [73]. The attacks consisted in searching a virtual machines' memory snapshot for login credentials and RSA private keys. We decided to only look for a private RSA key in a memory snapshot. The attack has two main stages. First, we need to obtain a memory dump (also referred to as memory snapshot) of the virtual memory space assigned to the virtual machine executing the Ubuntu 13.10 Linux operating system.



```
Vidigueira:rsakeyfind rocha$ hexdump ../vmss.core0 | awk '{print $2,$3,$4,$5,$6,$7,$8,$9,$10,$11,$12,$13,$14,$15,$16,$17}' > java.hex
Vidigueira:rsakeyfind rocha$ xxd -r -p java.hex java.bin
Vidigueira:rsakeyfind rocha$ ./rsakeyfind java.bin
FOUND PRIVATE KEY AT a7e1b8
version =
00
modulus =
00 ab 13 d7 1b 80 de e9 60 1c 88 a7 5a 65 b4 22
19 08 57 7c a8 db d2 6e cf bc 32 8d 59 c0 f8 66
5d bc a7 4f c3 b0 33 ba 88 ba ca cd a8 64 72 d0
f7 25 d4 4b 2c f2 48 c9 f0 20 07 7a 61 3c 75 ed
4b 2d d8 48 b8 c4 a4 4e e8 47 21 c9 b0 af b9 c4
ea 57 f0 c9 4e 39 39 51 4a 9c 39 73 78 26 62 66
f8 93 29 98 c3 52 2a 30 50 22 75 ac c2 32 99 7b
42 6e e9 c0 68 2e d1 e5 b4 63 e9 78 68 e9 ca 0d
b4 0b 9a c5 cf eb 4d c9 bc 88 6e 6b b2 c3 b6 44
bc ec a3 72 ce b8 e7 51 d1 d8 a4 42 34 d0 31 3b
1a e6 fa a5 12 29 01 ce c2 68 be 86 96 8c 77 ea
ed 00 88 04 b9 8b a1 79 b8 68 61 5c 82 dc 80 d1
49 fd 4e d7 63 01 45 67 e1 fc 47 91 f3 44 28 a5
8f 46 e2 52 72 e7 fa f4 00 45 bc f3 f9 33 42 0a
c1 81 b2 d7 be 1c 32 11 f9 f1 a5 74 3e 8f eb 75
8f 84 15 45 dd 89 a8 03 6d bb 37 96 af 68 4f 7f
41
publicExponent =
01 00 01
privateExponent =
25 ef d7 62 b4 f8 39 9e 63 95 76 2f 18 a0 92 73
24 a3 69 f4 74 12 bd 19 f9 ec e7 e5 80 62 55 28
f2 3d d3 f5 5c 2c 8c 6f e7 a4 33 68 e5 25 84 eb
59 8b 33 44 7b 51 e2 de 09 ec 3e 9a 33 73 1f 34
ed ac a9 ab b7 87 8c 2b fc 22 24 fc b6 6b 26 9c
a4 f5 b3 5e 52 58 40 8b a0 0b 29 6c fa 6e 15 d1
87 a8 d0 c4 77 82 fd 85 b7 ae 8a a9 e1 d8 a1 39
c1 28 ba cf 1f 9a d2 2a 67 81 8d 7f 0e 61 0d 1a
83 ce b3 02 b8 e5 12 2e 1d 69 66 61 aa 6c 88 63
50 2a 39 b6 ce 59 81 e1 80 99 04 35 3b 98 76 46
e0 81 a6 f2 c2 25 3b 42 db ca 2b 25 5c d8 53 f8
```

Fig. 4.16 Compromised private RSA key.

Second, we use the key search algorithm introduced in the *cold boot* attack to locate the intended private RSA key in the memory dump file.

VMWare's vSphere Web Client provides a set of management features among which you can find a memory snapshot functionality that an administrator can use to perform crash analysis of virtual machines. This option is also available through the ESXi shell using the command *vim-cmd vmsvc/snapshot.create vmid*. These are two valid approaches to obtaining a memory dump of the Ubuntu virtual machine, which is the first step in our attack.

The second part of this attack consists in using the key search algorithm introduced in the *cold boot* attack. This algorithm tries to locate instances of private RSA keys in a memory dump file. Since the memory dump contains the private private RSA key our Java application generated, the search algorithm is successful and returns the values for that key. Figure 4.16 displays the whole modulus value, the public exponent, and part of the private exponent. It is possible to verify that this value match the ones shown in Figure 4.15.

4.6.3 Conclusions

The attack we designed to check if VMWare's ESXi is vulnerable to the insider threat was successful. Therefore, we can compromise private RSA keys resident in memory of virtual machines that run on top of VMWare's ESXi hypervisor software.

This proves that VMWare is not enforcing the principle of least privilege to protect consumers' data. Hence, vulnerable to malicious insiders compromising confidentiality and/or integrity of consumers' data resident in virtual machines memory space. This is common in all the three platforms we tested in our project.

4.7 Related Approaches

This section discusses previous work addressing attacks against memory confidentiality and integrity. To the best of our knowledge, the work we presented here is novel. An approach closely related to ours was the cold boot attack where attackers with physical access to a machine could extract its random access memory in order to compromise data resident in the obtained storage units [32]. This attack was already discussed in this chapter.

The novelty of our work is on how it removes the need for physical access to random access memory and tests the same attack principles in a cloud computing environment. The cold boot attack was intended against personal computers. To the best of our knowledge, previous studies involving insider threats focused on employees compromising their employers systems or data [43]. In our scenario, an insider can compromise data that belongs to entities besides their employer.

4.8 Conclusions

This chapter presents the results which led us to conclude that a security design flaw exists in current virtualization solutions. We designate it as a security design flaw because the vulnerability we exploit to obtain access to consumer security sensitive data is related to design decisions for the virtualization software layer.

The tested virtualization solutions were not selected at random. According to a recent study Xen, Linux KVM, and VMWare are the most commonly used platforms in virtualization servers. The Aberdeen Group ran a survey that shows how Xen, Linux KVM, and VMWare combined are the primary choice of hypervisor software for 79% of virtualization servers. These three solutions can account for 83% of virtualization servers but not only as a primary solution. Furthermore, 58% of the inquired are evaluating one of these three platforms as a solution for future deployment [20].

Since we managed to execute successful attacks against all three platforms, it is safe to assume that the majority of servers running virtualization solutions is vulnerable to such attacks. Therefore, this is a relevant security problem that needs to be addressed.

The fact that our attacks reveal the same security problem in the major virtualization platforms does raise the question of how can these products not address a security problem such as the one identified in this chapter. From our perspective, it is probably related to the complexity of integrating the level of security that would better prevent insider threats. Therefore, to facilitate the implementation and operation of cloud ecosystems the virtualization software might opted to relax the security requirements in favour of functionality. The complexity of creating a cloud ecosystem that takes insider threats into account is demonstrated in the challenges and trade-offs discussed in the next chapters.

Chapter 5

Lightweight Mandatory Memory Access Control (LMMAC)

This chapter introduces the change we propose to a virtual machine monitor's memory management mechanisms. The change we suggest is the addition of a prevention mechanism which can eliminate the memory confidentiality and integrity problem we demonstrated on Chapter 4. The objective of these changes is to enforce the principle of least privilege in order to assure data security.

The principle of least privilege states that a subject is only given the privileges that it requires in order to complete its task [11]. Therefore, our objective is to devise a prevention mechanism that enforces the principle of least privilege to memory access whilst allowing a cloud administrator to perform the required operations to maintain a cloud infrastructure.

The solution we propose is a lightweight mandatory memory access control (LMMAC) prevention mechanism. Mandatory access control (MAC) is when a system mechanism controls access to an object and an individual user cannot alter that access [11]. LMMAC is a system mechanism that introduces controls in the virtual machine monitor that prevent a cloud administrator (user) from accessing the memory areas (object) assigned to a consumer's virtual machine.

The remainder of this chapter is organised as follows. We start with a detailed introduction to virtual memory systems and how those systems are implemented when a virtualization layer is present. Next, we discuss the privilege levels offered by processors with support for virtualization technology. After, we focus on our prevention mechanism explaining how it is feasible for one memory page, and how it can be applied to the memory space of a consumer's virtual machine to protect security sensitive data. Finally, we discuss the limitations of our work.

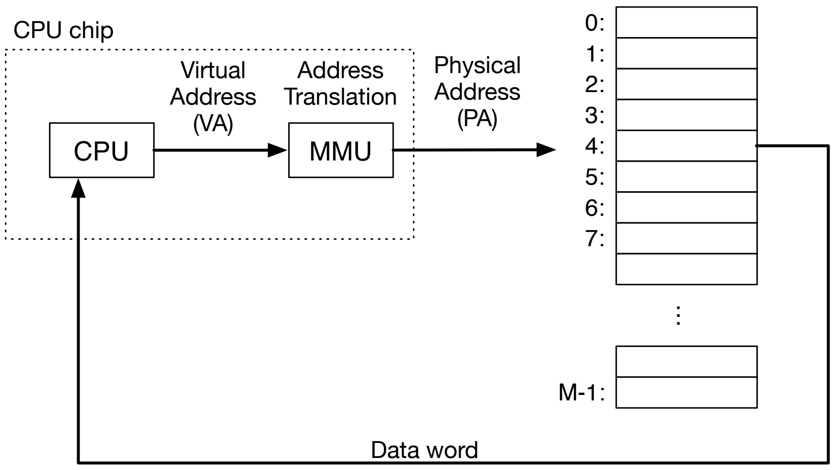


Fig. 5.1 Virtual address translation.

5.1 Virtual Memory

Consider a traditional system where an operating system manages multiple processes while executing directly on top of the hardware layer. In this type of system, processes share between them resources such as CPU and main memory. Sharing these resources creates several and distinct challenges to system developers. The challenges of sharing main memory include providing memory space for running processes and preventing processes from writing in each others memory spaces which can lead to unpredictable process failure states.

The efficient approach that most modern general-purpose operating systems use to manage main memory with fewer errors is denominated *virtual memory* (VM). Virtual memory is an abstraction of main memory which encloses hardware exceptions, hardware address translation, main memory, disk files, and kernel functionality that guarantees each process with a large, uniform, and private address space [13]. For our discussion assume a *linear address space*, which means a set of consecutive positive integer addresses.

There are a few important concepts that need to be clear in order to understand the inner workings of virtual memory. The first one is the difference between a *physical address* (PA) and a *virtual address* (VA). The main memory in a computer system consists of an array of *M* contiguous byte-sized compartments. A *physical address* uniquely identifies each byte in the array. A CPU can use physical addresses to access data stored in the *physical address space*, this technique is known as *physical addressing* and is used in systems like digital signal processors and embedded microcontrollers [13]. This is a simple and efficient mechanism to access main memory. However, most recent processors prefer an addressing scheme known as *virtual addressing*.

A processor using a *virtual addressing* approach uses *virtual addresses* to access memory cells in a *virtual address space*. The process of translating a *virtual address* to a *physical address* is referred to as *address translation*. A *virtual address* is always converted to a *physical address* before the access request is sent to physical memory. The *address translation* step is handled between processor hardware and the operating system. The operating system generates and maintains a look-up table in main memory, which a CPU hardware module known as *memory management unit* (MMU) uses to translate *virtual addresses* on the fly.

Figure 5.1 illustrates how a *virtual address* can translate to a different *physical address*. The *memory management unit* translates the *virtual address* to the corresponding *physical address* and sends the access request to physical memory. Main memory reads the memory location and sends the data back to the CPU.

5.1.1 Paging

Paging is a memory management technique where an address space is simulated with a small amount of physical RAM and some disk storage. The basic unit in a paging approach is a memory page. The address space is divided into memory pages, which typically have a fixed-size of 4 KBytes each [37]. A memory page can be unallocated, loaded in physical memory or stored on the disk [13].

A virtual memory system uses paging as a memory management solution. A memory page is used as a basic unit to transfer data between disk and physical memory. The virtual address space is divided into *virtual pages*. Each virtual page as a corresponding *physical page* (also know as *page frames*). In a system with more than one process, each process as its own virtual address space divided in virtual pages that map to physical pages. Processes can even share physical pages, which means different virtual pages mapping to the same physical page.

Assuming a two-level page table approach, similar to the one used in 32-bit paging. Figure 5.2 illustrates how a virtual address can be translated to a physical address. The operating system or virtual machine monitor uses a page directory and a set of page tables to keep track of memory pages. When a process tries to access a memory location, the processor uses the page directory and page tables to perform an address translation from virtual to physical address. The processor then executes the read or write operation on the requested memory location.

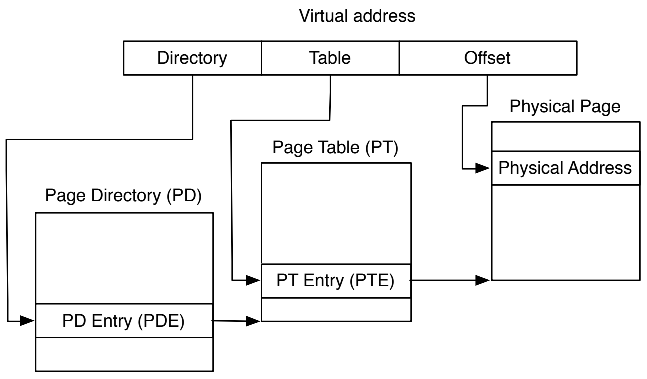


Fig. 5.2 Paging: virtual address translation.

5.1.2 Memory Virtualization

The virtual memory approach we just discussed is a solution mainly used in systems running a single operating system. In this subsection, we explain how the concepts of virtual memory are adapted in a virtualized platform. The terminology might be confusing, so there are a few definitions we require before proceeding with explaining how virtual memory behaves in a virtualized environment.

The necessary terminology for explaining how virtual memory is implemented in virtualized platforms includes, *host physical memory*, *guest physical memory*, and *guest virtual memory*. Definitions for these concepts follow.

- *Host physical memory* is the actual system hardware memory, which consists of an array of M byte-sized cells.
- *Guest physical memory* is the memory a guest operating system running on a virtual machine perceives as hardware memory. This is the memory a virtual machine monitor makes visible to the guest operating system.
- *Guest virtual memory* is the abstraction of main memory a guest operating system exposes to its applications. It consists of a contiguous array of N byte-sized cells.

The system hardware memory or host physical memory provides the storage space required by guest physical memory. Therefore, the virtual machine monitor maintains a table that maps guest to host memory [15, 31].

The virtual machine monitor creates virtual machines, in this process it assigns a contiguous memory address space to each virtual machine. A virtual machine is for a virtual machine monitor what an application is for a traditional general-purpose operating system.

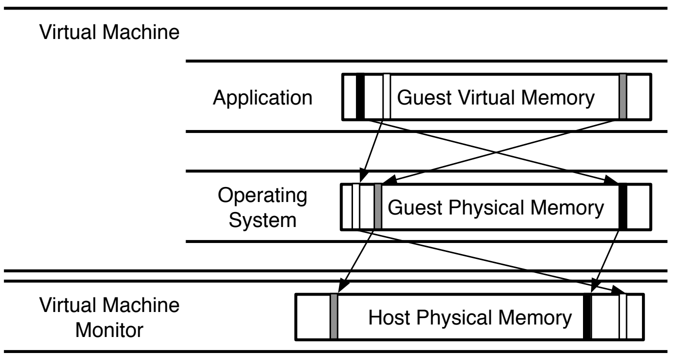


Fig. 5.3 Levels of indirection for systems with a hypervisor.

The memory space assigned to each virtual machine offers the same properties as the memory space an operating system assigns to its applications. Hence, a virtual machine monitor supports running multiple virtual machines in the same physical host while guaranteeing memory isolation between them. From the perspective of an application executing inside a guest operating system, having a virtual machine monitor present means an extra level of address translation that finds the host physical address for a given guest physical address [31].

Figure 5.3 depicts the address translation steps before the system can retrieve data resident in the host physical memory. The virtual machine monitor typically maintains a mapping table for each virtual machine in order to offer address translation from guest physical memory to host physical memory. The interception of virtual machine instructions for guest memory management allows the virtual machine monitor to maintain shadow page tables that contain translations from guest virtual to host physical memory. Data in shadow page tables is consistent with guest virtual to guest physical memory translation kept in the guest operating system page tables and with guest virtual to host physical memory translation stored in the mapping table.

The latest hardware-assisted virtualization technology offers support for memory virtualization. Intel through its Extended Page Table (EPT) technology and AMD with its Rapid Virtualization Indexing (RVI) approach. Both offer one layer to store guest virtual memory to guest physical memory translation, and a second layer for guest physical to host physical mapping [9, 10, 31].

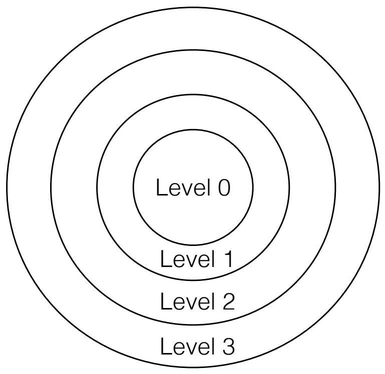


Fig. 5.4 Protection Rings.

5.2 Privilege Levels

A typical x86 processor offers protection based on the concept of a 2-bit privilege level. The privilege levels go from 0 to 3, where the lower the number the highest the privilege [37, 99]. Therefore, level 0 is the most-privileged whereas level 3 is the least-privileged.

These privilege levels are commonly represented as a set of protection rings as depicted in Figure 5.4. The central ring is used for critical software such as an operating system. The remainder of the rings are used for less critical software like user applications. A typical configuration in traditional systems is to have a general-purpose operating system executing in level 0 and user applications running in level 3, so only two privilege levels are used.

The purpose of the different levels of privilege is to prevent software running in a lesser privilege level from accessing data that belongs to higher privilege levels. The processor uses access rights to determine if the requesting software is granted access. For example, accesses to memory areas can be qualified as either a *supervisor-mode access* or a *user-mode access*. The processor keeps track of the current level of privilege and uses it to decide if the requesting software is granted access. Main memory areas assigned to supervisor-mode can never be accessed by a user-mode access request.

5.2.1 Privilege Levels and Virtualization

The protection rings approach is efficient in systems with a general-purpose operating system which supports the execution of several user applications. However, when virtualization is introduced a few challenges emerge such as the ring compression problem.

The introduction of a virtual machine monitor requires the use of ring de-privileging to keep it isolated from virtual machines executing on the same physical host. Executing in the most-privileged ring is useful to provide isolation between virtual machines. In this configuration, the VMM runs in ring 0, virtual machines can run in either ring 1 or 3, and user applications always run in ring 3. This creates the (0/1/3) and (0/3/3) models [37, 99].

A x86 CPU with support for 64-bit extensions needs to resort to paging to protect the VMM from guest operating systems. However, paging does not distinguish between rings 0, 1, and 2. Therefore, the only model available would be the (0/3/3) model which does not offer protection to guest operating systems from their own user applications because they both execute in ring 3. This is known as the ring compression problem [37, 99].

Hardware-assisted virtualization introduces two new CPU operation modes that can solve the ring compression problem. These new operation modes are known as VMX root mode and VMX non-root mode [99]. A virtual machine monitor executes in VMX root mode and it uses VMX non-root mode to execute its virtual machines. Both operation modes support the four protection rings which means the operating system can run at the expected privilege level.

These two new modes create two mode transitions denoted as *VM entry* and *VM exit*. The transition from VMX root mode (virtual machine monitor) to VMX non-root mode (virtual machine) is known as *VM entry*. *VM exit* refers to the transition from VMX non-root mode (virtual machine) to VMX root mode (virtual machine monitor). A new data structure is used to manage these two transition states and processor behaviour in VMX non-root state. When in VMX non-root mode, many instructions and events cause *VM exits*. These transition states can be managed by the virtual machine monitor.

The new modes introduced with hardware-assisted virtualization create the adequate environment to properly isolate virtual machine monitor from virtual machines and their user applications. The addition of these modes also maintains proper isolation between virtual machines' guest operating systems and their user applications.

5.3 Mandatory Memory Access Control: Single Page

This section presents our approach to testing the feasibility of the prevention mechanism we envisioned to assure memory confidentiality and integrity when considering an insider threat in cloud computing. The feasibility of our approach is demonstrated by testing if it prevents the security problem identified in Chapter 4. This is an example of devising a prevention mechanism as described in Subsection 2.1.3. Our approach choice is prevention because we possess detailed knowledge of the security problem.

We provide the details on our implementation of a mandatory memory access control scheme for a single virtual memory page, which is the basic unit in virtual memory systems

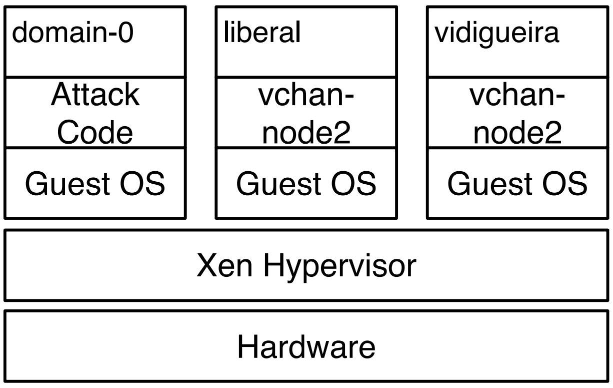


Fig. 5.5 Solution: Xen Test Environment.

as previously mentioned in Section 5.1. Therefore, if we can guarantee memory confidentiality and integrity for the basic unit we can then scale it to protect the whole virtual memory space of a virtual machine. We chose the Xen hypervisor as virtual machine monitor to implement our solution.

We chose to use inter-domain (or inter-virtual machine) communication as a test scenario because the communication channel uses a single memory page, which is ideal to test if our prevention mechanism can assure memory confidentiality and integrity for a single page.

5.3.1 Test Environment

The test environment comprised a single desktop. In terms of hardware configuration the server includes an Intel Core i7-870 64-bit CPU and 4GB of main memory. The server installation consisted of Xen 4.2 unstable with a Fedora 16 domain-0 running a 64-bit Linux kernel, version 3.1.5-1. The first virtual machine was running a 32-bit Linux kernel with physical address extension (PAE), version 3.3.2-6 for Fedora 16. The second virtual machines' operating system was also Fedora 16 but using a 64-bit Linux kernel with version 3.3.2-6.

Figure 5.5 depicts the test environment used to implement our prevention mechanism. In this configuration, we have two virtual machines with hostnames *liberal* and *vidigueira*. Both virtual machines are hardware virtual machines (HVM). The *liberal* virtual machine runs a Fedora 32-bit Linux kernel (3.3.2-6) with PAE. The *vchan-node2* application is executed so *liberal* can act as a server in an inter-domain communication channel with *vidigueira*. The purpose of this communication channel is the exchange of text messages between client and server. The *vidigueira* virtual machine uses *vchan-node2* to connect as a client to the instance executing in *liberal*. Domain-0 is a malicious insider entity which tries to execute attack code to compromise security sensitive data that belongs to cloud consumers.

5.3.2 Secure Inter-Virtual Machine Communication

The vulnerability we identified shows that a malicious insider can compromise the data exchanged between virtual machines (typically server and client) communicating with each other through an inter-virtual machine communication channel. The attack was demonstrated in Subsection 4.4.3. Therefore, we classify the current inter-virtual machine communication solution as insecure.

In this subsection, we provide a detailed explanation of how we can prevent an insider from exploiting the identified vulnerability. Hence, we classify our approach to be a secure inter-virtual machine communication solution. The prevention mechanism we propose requires changes to the Linux kernel and Xen's memory management.

We first need to map the memory virtualization concepts used in Xen to the generic memory virtualization approach we introduced in Subsection 5.1.2. In Xen, a physical address is known as a *machine address* (maddr) which can also be referred to as a *machine frame number* (mfn). For hardware virtual machines (HVM), we have *guest machine frame number* (gmfn) and *guest machine address* (gmaddr). A guest machine frame number or address (gmfn/gmaddr) is what a guest believes to be the real physical machine frame or address. A HVM guest runs in auto-translated mode, so it has gmfn different from mfn. The guest operating systems executing in hardware virtual machines use a virtual address space as they normally do when operating directly on top of the hardware layer. The virtualization layer handles the translation from gmfn to the physical memory (mfn) [5].

The Linux kernel uses a *page frame number* (PFN) to index either virtual (VPFN) or physical (PPFN) memory pages [52]. Understanding these concepts is paramount to follow the changes we have made to both the Linux kernel and Xen.

Shared Memory

The Xen hypervisor has a mechanism for sharing memory between virtual machines. This mechanism defines that a memory page can be owned by at most one virtual machine at any time. The mechanism offers the owning virtual machine means of forcing reclamation of mappings from misbehaving virtual machines. Therefore, shared memory mappings are classified as *asynchronous* and *transitory* [26].

A foreign mapping is when virtual machine A maps a memory page owned by virtual machine B. For a foreign mapping to be successful the requesting virtual machine must present a valid *grant reference* to Xen. A grant reference is composed of the identity of the virtual machine granting mapping permission and an index into that virtual machine's *grant table*.

A grant table's entries are of form (*grant*, *D*, *P*, *R*, *U*). Each entry can be read as: virtual machine *D* has permission to map memory page *P* into its own memory address space. Flags *R* and *U* are for read-only mode and mapping currently in use, respectively.

Xen maintains a private *active grant table* (AGT) for each virtual machine with entries of the same format as a virtual machine's grant table. Every time a grant reference (V, Idx) is passed to Xen, it searches for index *Idx* in the AGT for matches. If no match is found, Xen retrieves the necessary entry from the virtual machine's grant table, performs the required checks (e.g., if the requesting virtual machine matches *D* in the entry), and if all the checks are successful Xen copies the entry into the virtual machine's AGT and sets the in use (*U*) flag.

From a practical perspective, when *liberal* and *vidigueira* (Figure 5.5) establish an inter-virtual machine communication channel they go through the stages that follow. Assume *liberal* is granting mapping permission to *vidigueira*.

1. The *gntalloc* device in *liberal* allocates a page (*P*) of kernel memory.

2. An entry is added to *liberal*'s grant table that says *vidigueira* can map memory page *P*. No hypercall involved at this stage, unless the grant table needs to allocate more space for its entries.

3. The generated *grant reference* is sent to *vidigueira*.

4. Virtual machine *vidigueira* can now map page *P*. The mapping can either be done directly into kernel memory, or through the *gntdev* device so the page can be mapped into the address space of a user application executing in *vidigueira*. At this stage Xen performs permission checks.

5. The two virtual machines can communicate when an user application executing in *vidigueira* maps the shared page.

As an example of using this mapping process. In our test environment, the *vchan-node2* application executing in *liberal* performs a system call to the *gntalloc* device to allocate a memory page to share with *vidigueira*. The *vchan-node2* process executes as a user application in ring 3, so it needs to request a system call so the processor transitions to ring 0 and executes the *gntalloc* device (Step 1). The execution then follows the remainder steps until *vidigueira*'s *vchan-node2* process maps the same page. After this final step, the virtual machines can start using the shared memory page to communicate.

Changes: Linux Kernel

From Linux 3.0 onwards support exists for the native kernel to run as management virtual machine (domain-0). This support includes the drivers required in the management of *grant tables*. Grant tables are used in Xen as a method to allow memory sharing between virtual machines as we just discussed.

The establishment of a shared memory page for communication does not include a step that communicates which memory page is being used by which user application. The only strategy to obtain a list of shared memory pages with virtual machine *X* would be to traverse the *active grant table* page Xen maintains for virtual machine *X*. However, this table does not contain information about which application is using the memory page. Therefore, we decided to introduce a minor change to the sharing memory process so we could know which memory page was being used by the inter-virtual machine communication application *vchan-node2*.

The objective of our changes to the Linux kernel is to share with Xen which memory page is being used to establish an inter-virtual machine communication channel between two virtual machines. We introduced a call to a hypercall in the *gntalloc* device with the purpose of reporting to Xen the machine frame number (mfn) used by the inter-virtual machine communication channel, in our case between *liberal* and *vidigueira*.

Listing 5.1 Changes to the *gntalloc* device.

```

static int add_grefs(struct ioctl_gntalloc_alloc_gref *op,
        uint32_t *gref_ids,
        struct gntalloc_file_private_data *priv)
{
        ...
        +unsigned long mfn;
        ...
        for (i = 0; i < op->count; i++) {
                ...

```

```
                        +mfn = pfn_to_mfn(page_to_pfn(gref->page));
        }

+#define XENMEM_flag_pages 27
        +i = HYPERVISOR_memory_op(XENMEM_flag_pages, &mfn);
        ...
}
```

Listing 5.1 illustrates our changes to the *gntalloc* device which is part of the Linux Kernel. These changes store the machine frame number (mfn) in a variable and then use a memory operation hypercall to communicate the value of *mfn* to the virtual machine monitor (Xen). This value is not the actual machine frame number, it is the guest machine frame number (gmfn) of the memory page used to establish the inter-virtual machine channel. There is a local definition of *XENMEM_flag_pages* to avoid rebuilding the whole Linux kernel. At this stage, we need to modify Xen to process this hypercall and accept the information passed on from the operating system's kernel.

Changes: Xen

For our solution to properly protect the memory page used in an inter-virtual machine communication channel from unauthorized accesses, it is necessary to perform mandatory memory access control in the virtual machine monitor. To implement mandatory memory access control in Xen we added an hypercall to process the guest machine frame number sent from the Linux kernel, used an access control flag for that specific page, and enforced the access control when domain-0 requested access to map the memory page.

Memory management in Xen uses a data structure named *page_info* to keep track of how a memory frame is being used by the hypervisor. This data structure represents a single memory page, typically 4kb in a x86 architecture. For example, the *count_info* field serves as reference count and there are multiple flags to use with it such as *PGC_page_table*, which is set when the page is being used as a page table.

Listing 5.2 Changes to Xen's memory management.

```

xen/include/public/memory.h

#define XENMEM_flag_pages 27

```

```
xen/include/asm-x86/mm.h

#define _PGC_inv_dom0    PG_shift(10)
#define PGC_inv_dom0              PG_mask(1, 10)

xen/common/memory.c

long do_memory_op(unsigned long cmd, XEN_GUEST_HANDLE(void)
    arg)
{
            ...
            case XENMEM_flag_pages:{
            unsigned int gmfn, mfn;
            struct page_info *page = NULL;
            struct domain *md = NULL;
            p2m_type_t p2mt;
            uint64_t start, end;

            md = current->domain;

            if(md != NULL)
                        gdprintk(XENLOG_INFO,
                                    "--␣flag_pages:␣%d␣%d␣\n", op, md->
                                        domain_id);

            if(copy_from_guest(&gmfn, arg, 1)){
                        gdprintk(XENLOG_INFO, "--␣flag_pages:␣
                            copy_from_guest␣failed.\n");
                        return -ENOSYS;
            }

            start = tsc_ticks2ns((uint64_t)NOW());

            mfn = mfn_x(get_gfn(md, gmfn, &p2mt));
            gdprintk(XENLOG_INFO, "--␣flag_pages:␣conversion␣
                gmfn:0x%08x␣mfn:0x%08x\n", gmfn, mfn);

        page = mfn_to_page(mfn);

        if(page != NULL){
                gdprintk(XENLOG_INFO, "−−␣flag_pages:␣
                    count_info:0x%16lx\n", page->count_info);

                page->count_info |= PGC_inv_dom0;

                gdprintk(XENLOG_INFO, "−−␣flag_pages:␣
                    count_info:0x%16lx\n", page->count_info);
        }

        end = tsc_ticks2ns((uint64_t)NOW());

        gdprintk(XENLOG_INFO, "−−␣flag_pages:␣time␣elapsed:%
            lu␣ns\n", end − start );

        return 0;
    }
    ...
}

xen/arch/x86/mm.c

int
get_page_from_l1e(
    l1_pgentry_t l1e, struct domain *l1e_owner, struct
        domain *pg_owner)
{
...
struct domain *real_pg_owner;
...
if(mfn_valid(mfn) && page->count_info & PGC_inv_dom0){
```

```
68          real_pg_owner = page_get_owner_and_reference(page);
69
70          if(real_pg_owner->domain_id != 0 && curr->domain->domain_id == 0){
71                  MEM_LOG("Access from (%u) to map %081x from (%u) rejected\n",
72                                                  curr->domain->domain_id, mfn, real_pg_owner->domain_id);
73                  return -EPERM;
74          }
75  }
76  ...
77  }
```

Listing 5.2 displays the changes made to the memory management code of the Xen virtual machine monitor. The first couple of alterations were the definition of a new hypercall number to match the *XENMEM_flag_pages* used in the Linux kernel, and the definition of a new access control flag for memory pages (*PGC_inv_dom0*).

The hypercall number *XENMEM_flag_pages* is added as a new entry to the function (*do_memory_op*) that processes memory related hypercalls. The code we added performs the following steps to enable the access control flag (*PGC_inv_dom0*) to the correct memory page.

1. Copy the guest machine frame number (gmfn) into a local variable (*gmfn*), i.e., resident in Xen's memory space (line 29).

2. Obtain the machine frame number for the *gmfn* and store it in local variable *mfn* (line 36).

3. Use the machine frame number to get the *page_info* data structure for the memory page. A pointer to the data structure is kept in variable *page* (line 39).

4. Use the memory page's data structure to enable the *PGC_inv_dom0* access control flag on variable *count_info* (line 44).

The remainder of the code we added enforces the mandatory access control to the memory page used in the inter-virtual machine communication channel. The steps that follow implement our access control policy.

1. Verify if the machine frame number (mfn) is valid and check if the access control flag *PGC_inv_dom0* is enabled (line 67).

2. If the verifications in the previous step are successful, retrieve the data structure with the details of the domain that owns the memory page and store the data structure in *real_page_owner* (line 68).

3. Access is denied with *operation not permitted* (EPERM) if domain-0 is performing the access request but the page is not owned by domain-0 (line 70-74).

Putting It All Together: Secure Inter-VM Communication

We defined a new access control flag to use in memory pages we wish to isolate from domain-0. Using the information passed through the Linux kernel we can set that flag in the correct memory page. With that memory page's flag set we can later enforce an access control policy in Xen, preventing domain-0 from mapping a protected memory page. Access control has to be performed at the hypervisor level because the hypervisor is responsible for executing security sensitive operations.

This level of access to security sensitive information means that the hypervisor's code needs to be verified using trustworthy computing mechanisms. This verification can later be used to reported back to cloud consumers in order to assure them that the virtual machine monitor is isolating their data from potential insider threats. A cloud architecture discussing in detail how our approach can be used is presented in Chapter 6.

Figure 5.6 illustrates the steps involved in using the hypervisor to enforce memory access control policies. The steps are as follow. First, the Linux kernel with support for Xen (including our modifications) performs a hypercall informing Xen of which memory page is used to set up the inter-virtual machine communication channel (1). Second, Xen modifies the flags field for the referenced memory page and restricts access to domain-0, which in our scenario is the attacker (2). Finally, when domain-0 tries to map the memory page used for inter-virtual machine communication (3), Xen checks the flags for that memory page and since the restriction is in place it rejects domain-0 access to it (4). This configuration assures secure inter-virtual machine communication.

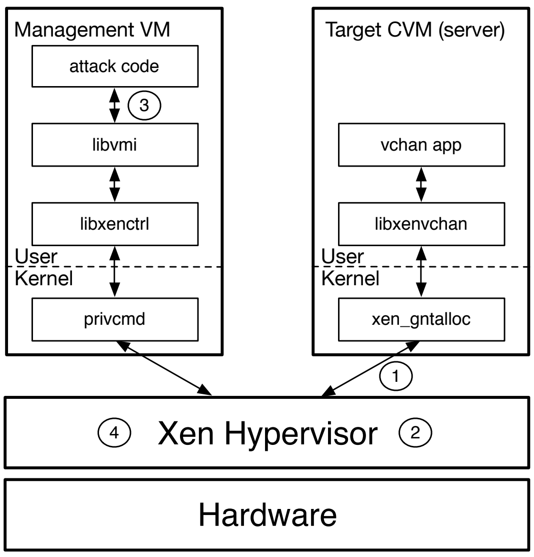


Fig. 5.6 Enforcing access control.

5.4 Lightweight Mandatory Memory Access Control

The previous section demonstrates the feasibility of using a memory page access control flag to enforce mandatory memory access control to a single memory page. Since we can guarantee proper isolation for a memory page, which is the basic unit in paging memory management solutions. We can expand this approach and apply it to the memory space assigned to a virtual machine.

In this section, we introduce the details of how a virtual machine monitor's memory management can use mandatory memory access control to enforce the principle of least privilege. Enforcing this principle prevents insider threats when considering memory confidentiality and integrity vulnerabilities. A widely accepted definition of the principle of least privilege defines it has a subject being given only the privileges that it needs in order to complete its task [11].

The design weakness we demonstrated in Chapter 4 allows a malicious cloud administrator to obtain access to consumers' security sensitive data such as cryptographic keys. This isolation problem spawns from adopting a very permissive memory access policy for the software managing (e.g., Xen's domain-0) a system's resources. An example of this permissive model is granting full access to the memory space of a consumer's virtual machine.

A solution to prevent this type of security issue needs to use the virtual machine monitor as the software layer responsible for enforcing the principle of least privilege when the management software tries to access resources assigned to consumers' virtual machines

[74]. To evaluate this solution we implemented a prototype using the Xen hypervisor. In what follows, we explain the details of our implementation.

Listing 5.3 Xen with LMMAC.

```

xen/tools/libxl/xl.h

int main_elmmac(int argc, char **argv);
int main_dlmmac(int argc, char **argv);

xen/tools/libxl/libxl.h

int libxl_domain_lmmac(libxl_ctx *ctx, uint32_t domid, int
    enable);

xen/tools/libxc/xenctrl.h

int xc_domain_lmmac(xc_interface *xch, uint32_t domid, int
    enable);

xen/tools/libxl/xl_cmdtable.c

struct cmd_spec cmd_table[] = {
        ...
        { "elmmac",
          &main_elmmac, 0, 1,
          "Enable LMMAC",
          "<Domain>",
        },
        { "dlmmac",
          &main_dlmmac, 0, 1,
          "Disable LMMAC",
          "<Domain>",
        },
}

```

```
xen/tools/libxl/xl_cmdimpl.c

int main_elmmac(int argc, char **argv)
{
    int opt;

    if ((opt = def_getopt(argc, argv, "", "elmmac", 1)) !=
      -1)
        return opt;

    find_domain(argv[optind]);
    libxl_domain_lmmac( ctx, domid, 1);

    return 0;
}

int main_dlmmac(int argc, char **argv)
{
    int opt;

    if ((opt = def_getopt(argc, argv, "", "dlmmac", 1)) !=
      -1)
        return opt;

    find_domain(argv[optind]);
        libxl_domain_lmmac(ctx, domid, 0);

    return 0;
}

xen/tools/libxc/xc_domain.c

int xc_domain_lmmac(xc_interface *xch, uint32_t domid, int
   enable){
        if(enable) return do_memory_op(xch, XENMEM_elmmac, &
            domid, sizeof(domid));
        else return do_memory_op(xch, XENMEM_dlmmac, &domid,
            sizeof(domid));
}

xen/tools/libxl/libxl.c

int libxl_domain_lmmac(libxl_ctx *ctx, uint32_t domid, int
    enable){
        return xc_domain_lmmac(ctx->xch, domid, enable);
}

xen/include/public/memory.h

#define XENMEM_elmmac                                   28
#define XENMEM_dlmmac                                   29

xen/include/asm-x86/mm.h

#define _PGC_inv_dom0    PG_shift(10)
#define PGC_inv_dom0            PG_mask(1, 10)

xen/common/memory.c

long do_memory_op(unsigned long cmd, XEN_GUEST_HANDLE(void)
    arg)
{
        ...
        case XENMEM_dlmmac:{
                ...
                md = rcu_lock_domain_by_id(domid);

                spin_lock(&md->page_alloc_lock);

                page_list_for_each ( page, &md->page_list ){
                        page->count_info &= PGC_inv_dom0;
                }

                spin_unlock(&md->page_alloc_lock);

                rcu_unlock_domain(md);

                return 0;
        }
    case XENMEM_elmmac:{
                ...
                md = rcu_lock_domain_by_id(domid);

                spin_lock(&md->page_alloc_lock);

                page_list_for_each ( page, &md->page_list ){

                        if(!( (page->count_info &
                            PGC_count_mask) > 1 && (page->u.
                            inuse.type_info & PGT_count_mask)
                             == 0 ) )
                        page->count_info |= PGC_inv_dom0;
                }

                spin_unlock(&md->page_alloc_lock);

                rcu_unlock_domain(md);

                return 0;
        }
    ...
}

xen/arch/x86/mm.c

```

```
int
get_page_from_l1e(
    l1_pgentry_t l1e, struct domain *l1e_owner, struct
        domain *pg_owner)
{
...
if(mfn_valid(mfn) && page->count_info & PGC_inv_dom0){
        if(pg_owner->domain_id != 0 && curr->domain->
            domain_id == 0)
                return -EPERM;
}
...
}
```

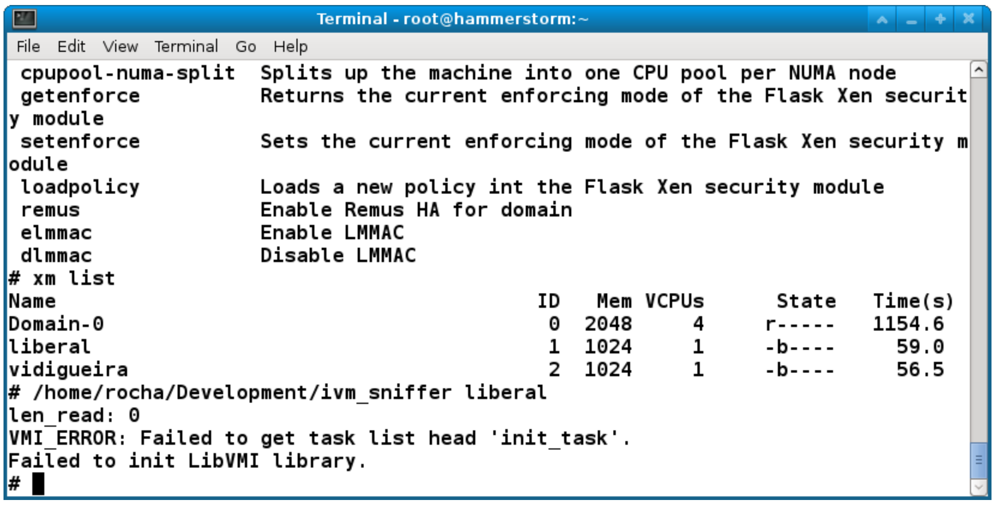
Listing 5.3 contains code excerpts of the additions made to Xen's management tool (*xl)* and Xen's memory management.

The changes to Xen's management tool (*xl*) consisted in the addition of two extra command line options that can be issued to enable (*elmmac*) or disable (*dlmmac*) LMMAC on the specified virtual machine.

The commands issued through Xen's management tool are handled by two new hypercalls added to the memory management operations section of Xen. The enable LMMAC option protects the memory pages for the virtual machine with the provided ID. This option does not enable the access control flag if the memory page is already in use with a special purpose such as a page table page. The disable LMMAC option simply clears the access control flag bit for all the memory pages assigned to a virtual machine.

Our changes to Xen's memory management show how it is possible to enforce the principle of least privilege in a Xen-powered cloud environment. We focused our study on the use of hardware virtual machine (HVM) guests. No tests for paravirtualized virtual machines were performed. This type of virtual machine should become less relevant with hardware-assisted virtualization. Our implementation is restricted to changes to the memory management of a virtual machine monitor (i.e., Xen hypervisor).

The first step to assure that Xen is respecting the principle of least privilege is to identify which memory accesses from domain-0 into a guest virtual machine memory space are critical to guarantee the virtual machine's correct operation. These memory accesses depend on the type of guest virtual machine as follows. First, domain-0 maps a guest virtual


Terminal - root@hammerstorm:~
File Edit View Terminal Go Help

```
 cpupool-numa-split  Splits up the machine into one CPU pool per NUMA node
 getenforce          Returns the current enforcing mode of the Flask Xen securit
y module
 setenforce          Sets the current enforcing mode of the Flask Xen security m
odule
 loadpolicy          Loads a new policy int the Flask Xen security module
 remus               Enable Remus HA for domain
 elmmac              Enable LMMAC
 dlmmac              Disable LMMAC
# xm list
Name                                        ID   Mem VCPUs      State   Time(s)
Domain-0                                     0  2048     4     r-----    1154.6
liberal                                      1  1024     1     -b----      59.0
vidigueira                                   2  1024     1     -b----      56.5
# /home/rocha/Development/ivm_sniffer liberal
len_read: 0
VMI_ERROR: Failed to get task list head 'init_task'.
Failed to init LibVMI library.
# 
```

Fig. 5.7 Xen with LMMAC.

machine's memory to load the Xen virtual firmware, which is a virtual BIOS used to assure hardware virtual machines possess the expected standard start-up instructions. A second mapping takes place when Xen emulates the hardware used in hardware virtual machines that operate with a special purpose application executing in domain-0.

The changes to Xen consisted in using the list of memory pages allocated to a consumer's virtual machine and the flags associated to each memory page to distinguish between free (the virtual machine can use the memory page) and special (used by domain-0 for the purposes mentioned in the previous paragraph) memory pages (line 111). Traversing the page list contained in the main domain data structure, makes it possible to achieve this objective. While traversing the list we check if the flags of a memory page indicate it is mapped to domain-0 for special purposes, if it is the code does nothing, else it enables our memory access control security flag. Iterating through the complete list of memory pages assigned to a virtual machine guarantees that all but the special purpose memory pages have the restricted memory access flag active. Therefore, domain-0 will only have access to the special purpose memory pages and be denied access for request to access any other memory pages.

After implementing this lightweight mandatory memory access control mechanism we retried the previously successful attacks against a Xen-powered cloud server. Figure 5.7 shows it is no longer possible to perform the attack demonstrated in Section 4.4. We enabled LMMAC for the virtual machine with hostname *liberal* before trying to run the attack code.

The results show that our approach can enforce the principle of least privilege and protect data from insider threats. The attack code used before was unable to compromise any consumer data. Furthermore, the changes made to the virtualization layer prevent any kind of memory introspection to be performed on virtual machines.

This implementation is a generalisation of what was demonstrated with secure inter-virtual machine communication. The key point we try to demonstrate with this prototype is that it is adequate to use an access control memory page flag to enforce a mandatory memory access control policy that enforces the principle of least privilege to memory accesses. It is important to clarify that this solution applies to a single cloud server, which means that to secure a cloud ecosystem an architecture that builds on this secure foundation needs to be devised.

5.4.1 Security Analysis

This security analysis focuses on how a lightweight mandatory memory access control policy guarantees *confidentiality* and *integrity* to the memory space assigned to a consumer's virtual machine. Our proposal aims at preventing unauthorised accesses to physical RAM, we do not consider secondary storage security problems such as access to consumers' stored unencrypted data [73].

The objective of this analysis is to show how a virtual machine monitor with support for a mechanism such as lightweight mandatory memory access control (LMMAC) can prevent insider threats for memory confidentiality and integrity by guaranteeing proper isolation between virtual machines and management software. We do not consider any availability related issues. The assumptions for this analysis take into account that even if a virtual machine monitor implements our prevention mechanism it still needs to use hardware-assisted remote attestation to prevent attackers from compromising its system integrity. This contrasts with the adversary model used for Chapter 3 and Chapter 4.

For our analysis, we consider the set of assumptions that follows.

- *Assumption 1*: an attacker cannot compromise a virtual machine monitor's system integrity.
- *Assumption 2*: No hardware attacks are considered.

An insider threat cannot compromise *data confidentiality* if the virtual machine monitor restricts access to the memory pages assigned to a consumer's virtual machine. A VMM using a prevention mechanism like our lightweight mandatory memory access control approach prevents management software from mapping memory pages that might contain security sensitive data. The paragraphs that follow compare our prevention mechanism with two previous solutions that assure data confidentiality for consumers' security sensitive data while it is stored in the memory space of a virtual machine [50, 76].

The first approach we consider guarantees data confidentiality of memory pages through the use of cryptographic primitives. It encrypts the memory pages in the virtual machine monitor before access is granted to any management software such as Xen's domain-0 [50]. When compared to our prevention mechanism, this approach has a couple of limitations. First, the use of cryptography means the handling of encryption and decryption of memory pages introduces some processing overhead and key management challenges. Second, the fact that pages are encrypted before management software can have access to it invalidates the implementation of virtual machine monitoring solutions.

The second approach is very similar to ours, but it works using a set of policies which in turn insert hooks in the virtual machine monitor's code. This solution is more dynamic than ours [76]. However, it requires loading a separate module and it is not very clear how such an approach would behave when trustworthy computing is used to offer remote attestation to consumers. One of the requirements of an integrity-protected virtual machine monitor is the use of a dynamic root of trust, as provided with the latest trustworthy computing technology [101]. In this scenario, our solution simply adds a few lines of code to virtualization layer not interfering with the use of a dynamic root of trust to boot the virtual machine monitor in a trusted fashion.

Not permitting management software to map any of the security sensitive memory pages of a virtual machine assures that our solution guarantees *data integrity* for consumers' data executing in their virtual machines. Previous work uses encryption to protect a virtual machine's memory pages from unwanted accesses [50]. However, the management software still gets access to the encrypted pages to perform save, restore and migrate operations. At this stage, the attacker has the memory pages in his possession, so he can keep the encrypted memory pages for later cryptanalysis. In our approach, the management software does not have access to virtual machines' memory pages.

Assuring these security properties creates a strong foundation to build secure cloud computing architectures that can offer trustworthy services to cloud consumers. The chapter that follows presents a secure cloud computing architecture that takes advantage of these properties.

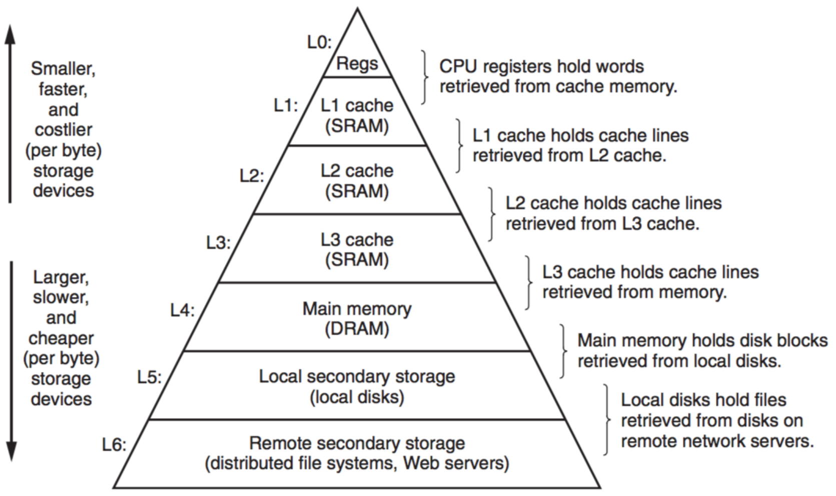

Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0: Regs
L1: L1 cache (SRAM)
L2: L2 cache (SRAM)
L3: L3 cache (SRAM)
L4: Main memory (DRAM)
L5: Local secondary storage (local disks)
L6: Remote secondary storage (distributed file systems, Web servers)
CPU registers hold words retrieved from cache memory.
L1 cache holds cache lines retrieved from L2 cache.
L2 cache holds cache lines retrieved from L3 cache.
L3 cache holds cache lines retrieved from memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote network servers.

Fig. 5.8 The memory hierarchy [13].

5.4.2 Memory Performance

Since our changes affect Xen's memory management mechanisms, we decided to perform some memory performance measurements to evaluate if there is an impact on the memory performance of virtual machines. We chose to measure the *latency* of a memory read operation together with the *throughput* for a few different memory operations. Latency is the time it takes to complete a single memory operation whereas throughput gives the number of operations the system can complete per unit time [13]. These two measurements give us an idea of the performance impact of our prevention mechanism.

Memory Hierarchy

Memory hierarchy refers to the different levels of memory storage organised according to the time a central processing unit requires to access each different level. The memory hierarchy diagram is included in Figure 5.8. This diagram lists the different memory storage units from the smallest, faster, and costlier to the lager, slower, and cheaper ones.

For our discussion, the relevant memory types are the L1, L2, and L3 caches plus main memory. Two storage technologies are used for these types of memory. Cache memories use the fast and expensive Static Random-Access Memory (SRAM) which can be access in a few CPU clock cycles. Whereas main memory uses a slower and cheaper technology known as Dynamic RAM (DRAM) which can be accessed in tens to hundreds clock cycles [13].

The measurements presented in this subsection where performed in a virtual machine

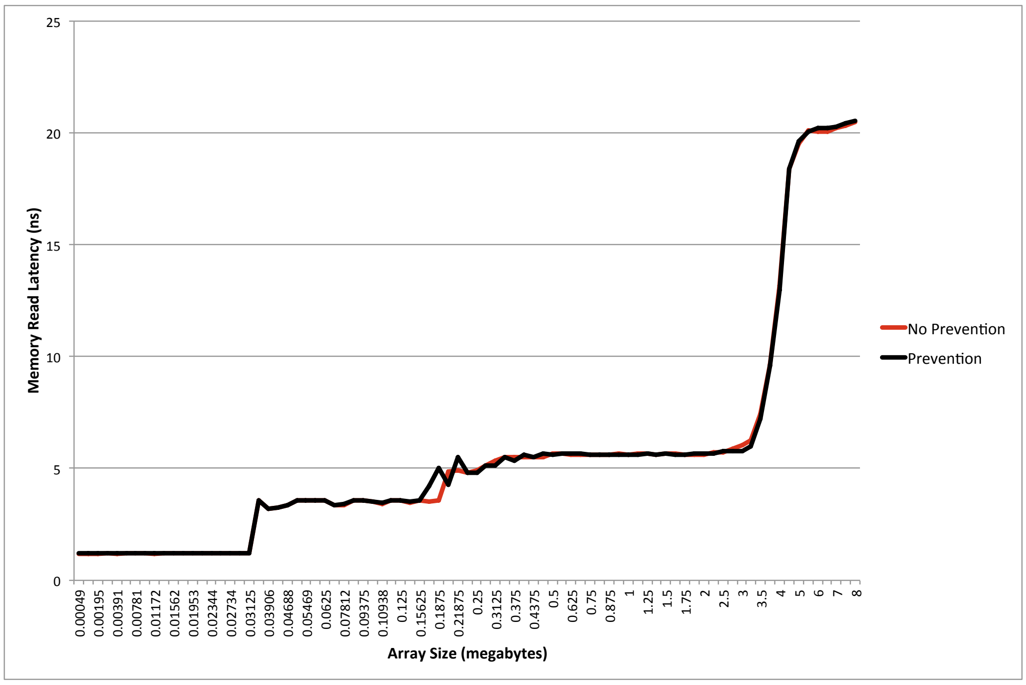


Fig. 5.9 Memory latency.

with 32 kilobytes of L1 and L2 cache, 4 megabytes of L3 cache, and 1 gigabyte of main memory.

Latency

To collect the latency of memory read operations we used the *lat_mem_rd* tool from the LMBench benchmark suite for the Linux operating system [56]. The memory read benchmark requires a parameter specifying the maximum array size. The stride is automatically established using the size in bytes of a pointer, i.e., if the system is 64 bits a pointer is 8 bytes which originates a stride of 64 (512/8) bytes. The stride can be changed if passed as a parameter.

To measure memory latency the benchmark then uses the stride size to create a ring of pointers to loop through one million times while measuring each read operation. This process is repeated for multiple array sizes from 512 bytes until the maximum array size is reached.

Figure 5.9 displays the latency results for memory read operations when our prevention mechanism is enabled and when it is disabled. The graph presents the relationship between memory latency in nanoseconds and multiple array sizes in percentages of a megabyte ranging from 512 bytes to 8 megabytes. To populate this graph we collected measurements for

Megabytes	No Prevention	Prevention
6	20.05	20.195
6.5	20.066	20.231
7	20.196	20.282
7.5	20.307	20.449
8	20.463	20.538

Table 5.1 Memory Latency.

array sizes from 512 bytes to a maximum of 8 megabytes. The maximum value was chosen so it would exceed the capacity of the virtual machine's cache memories. The stride value was set to 128 bytes. This guarantees that read operations to main memory are measured as well [56].

From the graph it is noticeable that the prevention mechanism has a small negative impact in the memory latency of read operations. However, it is a negligible negative impact when you consider the security benefits of making the virtualization layer secure by default in terms of memory isolation. It is also evident when the cache memories have an impact on read performance and then main memory is required. The lowest latency values are due to L1 cache, followed by L2 cache, then L3 cache, and finally the value rise to close to 20 nanoseconds when main memory is required.

Since the different is very small we include table 5.1 to help with interpreting the results. The values in the table are in nanoseconds and correspond to the latency of a memory read operation of the array size in the megabytes column. The table makes it evident that there is a small negative performance impact when the prevention mechanism is enabled. We chose to display array size values greater than the cache because the difference is more evident when main memory is in use.

Throughput

Throughput performance measurements were obtained with the memory bandwidth tool part of the LMBench benchmark suite [56]. This tool requires two parameters defining the memory size and the operation to be tested. The size parameter can be specified in kilobytes (ending with *k*, e.g., 8k) or megabytes (ending with *m*, e.g., 1m). The operation relevant for our tests is the memory read operation which is selected using *rd* as a second parameter.

Measuring the throughput of memory read operations is done by allocating the amount of memory specified as a parameter, zeroing it, and then recording the reading times for that memory as a series of integer loads and adds. For each four byte integer a load and

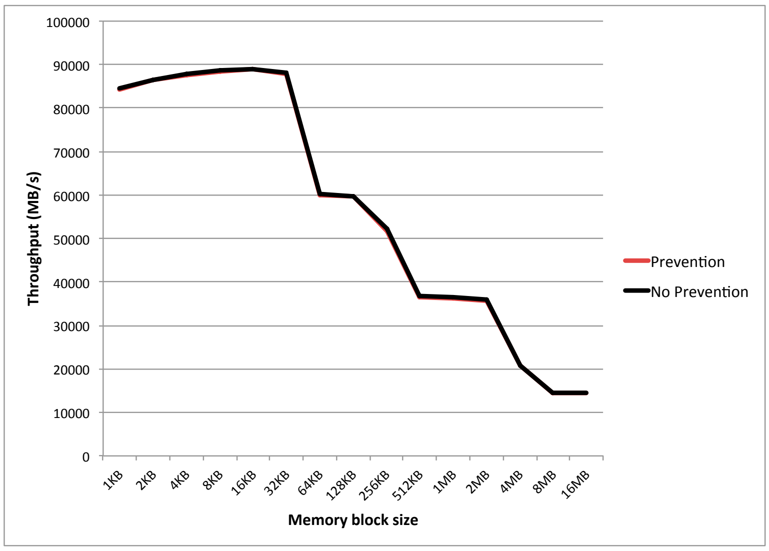


Fig. 5.10 Memory Throughput.

add operations are performed. The add operation is assumed to take one clock cycle and therefore it does not influence the results [56].

The graph in Figure 5.10 illustrates the results for memory read throughput with multiple memory test blocks. Each memory block was tested a thousand times. The difference in throughput when our prevention mechanism is negligible as expected because the latency overhead was not considerable. It is not easy to notice but the red line correspondent to throughput values when our prevention mechanism is enabled is visible below the black line. In this graph it is also noticeable the influence of cache memories for smaller memory test block. The throughput stabilizes once frequent accesses to main memory are required. Table 5.2 contains the values used to plot the graph to provide the actual numerical difference.

Megabytes	No Prevention	Prevention
1	36440.91178	36352.41286
2	35954.24325	35799.09897
4	20834.06299	20734.16386
8	14550.42808	14524.38718
16	14506.69734	14468.06397

Table 5.2 Memory Throughput.

5.5 Related Approaches

This section presents other research approaches that can be alternatives to our prevention mechanism. These solutions can also assure data confidentiality and integrity against insider threats but have distinct limitations from ours.

The first solution is a mechanism to guarantee secure virtual machine execution in an untrusted cloud environment. This mechanism encrypts a virtual machine's memory pages before handing them over to the management software. This assure confidentiality is kept even in the presence of a malicious insider with control over such management software. However, this solution is too restrictive because it cannot implement cloud service that require access to cloud consumer's data.

Cloudvisor is a lightweight virtual machine monitor that is placed between the hardware and the virtual machine monitor. The aim of this location is so that CloudVisor can intercept and manage every resource usage for VMM and VMs. Granting CloudVisor a level of privileges higher that the virtual machine monitor and virtual machines allows it to enforce isolation to memory resources. The main limitation of this approach is negative performance impacts as high as 22%.

A completely different alternative is propose in NoHype [44]. This two approach suggests the removal of the virtualization layer and the use of hardware-assisted virtualization technology to guarantee the execution and isolation of virtual machines. A limitation of this solution is the higher costs due to the inability of assigning fine grain units (e.g.. sell a 1/4 of a core) to virtual machines. However, reducing the trusted computing base to this order of magnitude would be ideal.

Compared to this approaches our solution can be classified as more flexible than the first two because it allows the definition of cloud architectures that can guarantee data confidentiality and integrity while offering services that can access unencrypted consumer's data. When compared to NoHype it would simply be a question of the level of security required by the consumers using the infrastructure. If costs are not a problem a server using NoHype is a very good solution assuming NoHype is secure against the kind of attack we tested in our work.

5.6 Limitations

The prevention mechanism we proposed in this chapter has two main limitations. One limitation is in the prototype we implemented while the second one is in the architecture we used to implement that prototype.

Our implementation has limitations because it does not allow the execution of virtual machine management operations that require access to memory pages. This happens because the virtual machine monitor is not granting management software access to any memory pages assigned to a virtual machine. We implemented the *disable LMMAC* option to clear the memory page flags but some operations do not resume proper execution even after we use the disable option. However, the implementation achieves the main objective we had for it, it demonstrates that a mandatory access control at the virtual machine monitor level can prevent insider threats and guarantee memory confidentiality and integrity. This constrain is limited to our implementation, it does not mean a commercial solution using our approach would be unable to circumvent this problem.

The second limitation is in the architecture we used which derives from the design of the Xen hypervisor. This architecture contemplates a monolithic virtual machine monitor which composes the virtualulization layer. However, the approach followed in monolithic virtual machine monitors has limitations when compared with microhypervisors such as the NOVA hypervisor [90].

The major limitation of a monolithic approach when compared to microhypervisors is that it has a considerably larger trusted computing base [90]. The Department of Defense (DoD) defines a trusted computing base to be “the totality of protection mechanisms within a computer system, including hardware, software and firmware, the combination of which is responsible for enforcing a security policy” [19]. The architecture of microhypervisors is ideal to take advantage of the root-mode protection rings introduced with virtualization extensions such as Intel VT-x and AMD-V. These protection rings were discussed in Subsection 5.2.1. This means a microhypervisor can run security critical code in ring 0 in root-mode and de-privilege other functionalities to ring 3 still in root-mode. This approach creates a smaller trusted computing base which means a lower probability of it having exploitable software vulnerabilities [58].

Chapter 6

Trustworthy Cloud Computing Architecture (TCCA)

This chapter introduces a trustworthy cloud computing architecture that uses the security properties offered by a virtual machine monitor that enforces the principle of least privilege such as the one we suggest in Chapter 5. These security properties are a strong building block to provide trustworthy cloud computing services to cloud consumers.

For a cloud computing service to be considered trustworthy it needs to report an integrity measurement to an enquiring cloud consumer. A cloud consumer can use the integrity measurement to decide if it trusts the service. We define this security property as *transparency*.

Transparency is a requirement that a remote untrusted system presents integrity measurements to consumers through the use of trustworthy mechanisms. These measurements allow a cloud consumer to verify the system integrity and/or trustworthiness of the remote system. This chapter explains how we can use a trustworthy virtual machine monitor and trusted computing technology to build a trustworthy cloud computing architecture.

The remainder of this chapter is organised as follows. First, we introduce the security critical software components that compose a cloud computing server. Second, the architecture requirements to build a trustworthy cloud computing ecosystem are described in detail. Finally, we discuss how these components and mechanisms come together to create a trusted virtualization environment and give an example scenario where this trusted environment is used to offer trustworthy cloud management operations.

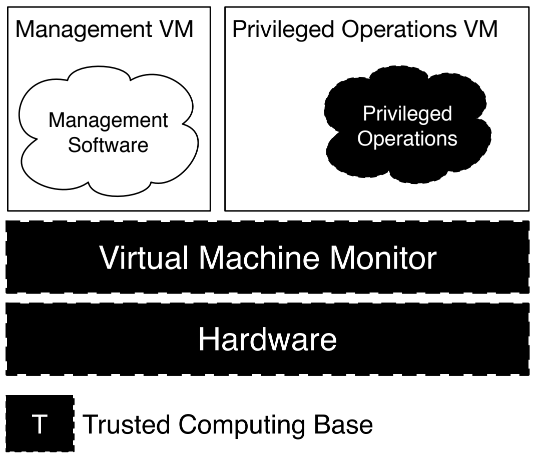


Fig. 6.1 Cloud server TCB components.

6.1 Cloud Server Components

The components of a cloud server differ from traditional approaches and each component is responsible for distinct security related tasks. The virtual machine monitor, as previously discussed, is the virtualization layer which makes it possible for multiple virtual machines to execute and share resources on the same physical server.

In our architecture, in addition to those tasks, a virtual machine monitor is also the policy decision and enforcement point for memory access. This is a critical difference in terms of security because in the past the management virtual machine had access to the whole memory space assigned to a consumer's virtual machine. Although there are special cases where consumer virtual machines can have direct access to hardware pass through solutions, the management virtual machine is usually responsible for supplying hardware drivers, virtual storage, and network access.

Current cloud architectures use the *management virtual machine* for managing and monitoring virtual machines, which opens an easy and direct attack vector for malicious insiders as demonstrated in Chapter 4. In the architecture we propose in this chapter, however, the privileges of the management virtual machine were reviewed, and the operations that allowed attacks on consumer's data were moved to an isolated special-purpose virtual machine. This isolation implies that the virtual machine monitor must be compromised for an attacker to obtain access to the whole memory range.

The *privileged operations virtual machine*, is the special-purpose virtual machine in charge of performing the security sensitive operations on consumer's data, e.g., launching and migrating a virtual machine. These operations are very sensitive because they involve

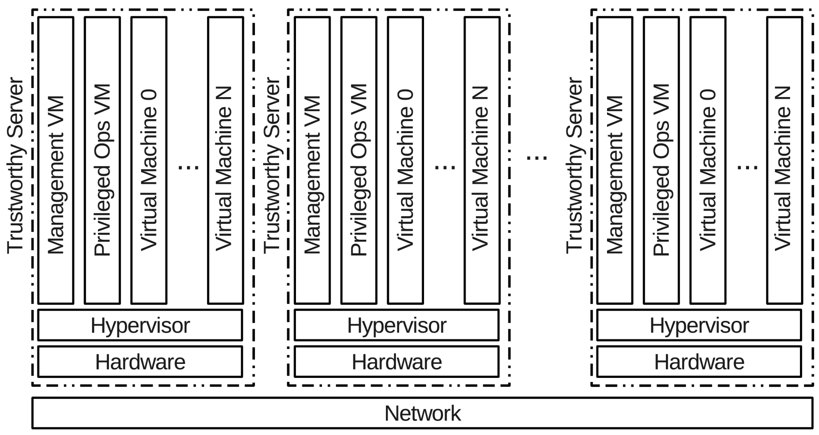


Fig. 6.2 Trustworthy Cloud Architecture.

accessing the whole memory space of a consumer's virtual machine. These virtual machines can have a reduced trusted computing base if they use solutions such as unikernels, e.g., Mirage OS and OSv [46, 53]. These kernels were developed to permit the execution of a single application on top of the virtual machine monitor, reducing the trusted computing. The last component is the *consumer virtual machine*, which can be zero or more depending on the number of requests.

The major difference from past approaches is the dumbing down of the management virtual machine by reducing its privileged operations. This alteration facilitates a more appropriate virtual machine isolation, which is important to assure better overall security. The management virtual machine behaves more like a terminal for the administrator, the operations will then be performed in a different virtual machine with enough privileges to process the request whilst keeping security sensitive information isolated.

This redesign does not offer holistic protection from insider threats because it is still possible for an administrator to originate denial of service (DoS) attacks trying to prevent required administration tasks such as stopping a virtual machine from launching. However, it is an improvement when compared with current solutions.

The current memory access model for the virtualization layer is excessively permissive which does not comply with the security design principle of least privilege. Figure 6.1 shows how the components present in the trusted computing base of a single cloud server are organised in the trustworthy cloud computing architecture we propose. The most critical component is the virtual machine monitor, or hypervisor, which is where the security policies are enforced. Therefore, the virtual machine monitor is responsible for enforcing the principle of least privilege to the memory access model and deny access to the management software when it tries to obtain access to memory pages assigned to a consumer's virtual machine. This virtual machine monitor functionality was demonstrated as being achievable in Chapter 5.

This cloud server architecture suggests the use of mandatory memory access control at the virtual machine monitor layer, preventing the software managing the system resources from accessing the memory space where consumer's security sensitive data resides. One of the advantages of this approach is that it is in accordance with the principle of isolation in virtualization, which affirms that the hypervisor is responsible for isolating the multiple virtual machines running on the same physical server [81]. Another advantage is the reduced trusted computing base when compared to previous approaches. Our solution includes the hypervisor, and the special- purpose software dedicated to performing privileged operations, e.g., building a new virtual machine. This means it does not need to worry about the full blown Linux Kernel used in traditional management virtual machines such as Xen's domain-0.

6.2 Architecture Requirements

Our proposed solution is a trustworthy architecture that supplies extra security mechanisms to consumers, but also requires a few adjustments in terms of basic cloud functionality when compared with current approaches. The architecture we propose in this chapter focuses on the data execution problem as discussed in Chapter 3. The problem of protecting data-at-rest is not address in our work.

Figure 6.2 provides a view of the trustworthy cloud architecture containing multiple physical servers interconnected through a local network. A typical trustworthy server has a few components, including the hypervisor, a management virtual machine, a privileged operations virtual machine, and a variable number of consumer virtual machines. These servers are required to support trustworthy computing such as Intel's Trusted execution Technology (TXT) [30].

Providing a trustworthy cloud architecture has some requirements. In what follows, we present the set of requirements we consider paramount to build such an architecture. These requirements were inspired by work the NIST has done addressing the issues of BIOS integrity measurement [69]. We determined the key requirements for a consumer to evaluate the trustworthiness of a cloud platform to be:

- A trusted authority responsible for generating golden integrity measurements for platform software or software agents.

- Components capable of generating and collecting integrity measurements.
- Tamper resistant or tamper evident storage for integrity measurements.
- A secure communication channel for transferring integrity measurements between consumer and cloud provider.
- A protocol to establish the trustworthiness of software executing in a remote platform.

How to satisfy these requirements is explained in detail below. Our explanation elaborates on how basic security principles are achieved in order to guarantee cloud consumers can trust the secure cloud ecosystem we propose.

Golden Integrity Measurements

A fundamental requirement for this architecture is the existence of a trusted authority responsible for receiving the original software from cloud providers and performing the required security tests before generating a *golden integrity measurement* for it. A *golden integrity measurement* is a cryptographic hash code of the original trustworthy binary code and/or data the trusted authority verified. It is also the correct cryptographic hash a consumer expects to receive when verifying the trustworthiness of a particular cloud platform software such as the hypervisor. In our propose cloud ecosystem, golden measurements measurements are intended to offer functionality similar to the one possible using reference integrity metric (RIM) certificates described in the mobile trusted module specification [94].

The use of a cryptographic hash is the logic design choice because cryptographic hash functions are collision-resistant, which means that it is computationally impracticable to find a different input that provides the same output/hash. Therefore, if a certain block of code has a hash *h1*, it is impossible to use a different block of code, as input to the cryptographic hash function, and obtain the same hash *h1*. Subsection 2.2.1 offers a more detailed discussion on the security properties of hash functions. This property is key in assuring that a particular integrity measurement is unequivocally bound to a single block of binary code and/or data. However, a golden integrity measurement does not have the objective of providing information on the presence or absence of vulnerable properties, e.g. buffer overflows, in the measured binary code and/or data.

We envision the generation of golden integrity measurements as a strong addition to the security development lifecycle. The security development lifecycle consists in a set of traditional software development lifecycle phases with the addition of security steps that intend to reach a final product as resilient as possible against malicious attacks. The phases include requirements, design, implementation, verification, and release. The verification phase already requires an independent team to perform a final security review of the product [51]. Therefore, it should be trivial to bring in the generation of valid golden integrity measurements.

Although a secure hash function offers certain security properties, it is not enough to assure the authenticity of a golden integrity measurement. Let us consider a scenario where a cloud provider develops his own infrastructure management software and generates himself the golden integrity measurements. In such cases, if consumers trust the provided golden integrity measurements then they are still vulnerable to attack originating from an insider threat within the cloud provider environment. Hence, we suggest the use of trusted authorities such as the ones in the well established public-key infrastructure (PKI). This approach builds a chain of trust which the consumer can trust to verify the authenticity of golden integrity measurements. In what follows, we explain in detail how such chain of trust is achieved.

The primary goal of implementing a public-key infrastructure is to enable secure, convenient, and efficient distribution of public keys. A public-key infrastructure (PKI) is defined as the set of physical or logical entities and procedures required to achieve such an objective. The required procedures deal with the creation, management, storage, distribution, and revocation of digital certificates based on asymmetric cryptography [86].

The digital or public-key certificate is the key component in a public-key infrastructure because it is a secure and scalable means for the distribution of public keys [48]. Previous approaches relied on a central public-key authority which would hold every public key and distribute them to users through a direct request. Therefore, it would always be involved in the transactions becoming a single-point of failure and a bottleneck. The use of digital certificates allows users to exchange public keys without the intervention of a public-key authority. However, the existence of a trusted third party denominated *certificate authority* (CA) is required. A certificate authority is typically a government, financial institution, or security company trusted within the user community.

The creation of a digital certificate requires the user to transmit its public key to the certificate authority in a secure fashion. An unsigned certificate is basically the user's public key, data that uniquely identifies the user, and certificate authority identification details. The certificate authority uses a secure hash function to obtain a hash code of all this data and encrypts that hash code using its private key to create a digital signature. This signature is then attached to the certificate. The user can then make the certificate public for anyone to use or send it as a reply to individual requests.

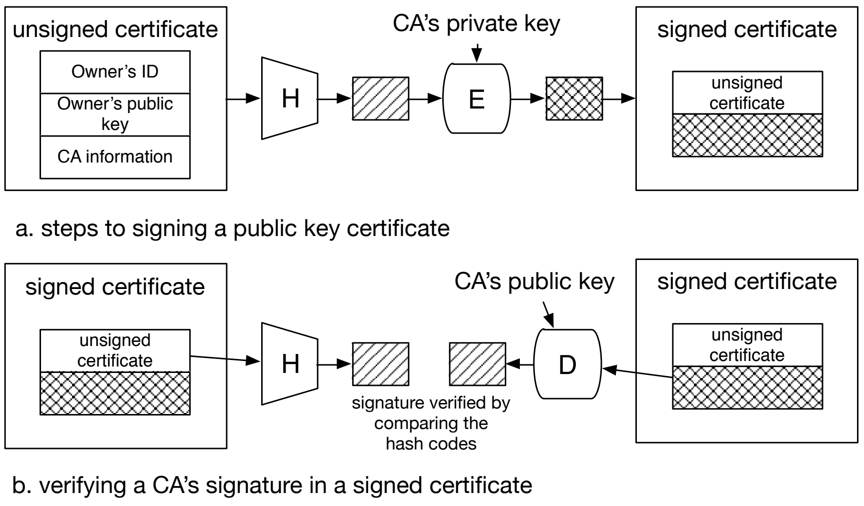


Fig. 6.3 Public-key certificate verification.

Figure 6.3 illustrates how the party using the certificate can verify the user's public key through the digital signature attached to it. This signature is verified using the certificate authority's public key to decrypt the signed hash code and comparing it to a hash code the verifying party generates from the certificate data. A match between the hash codes guarantees the user's public key can be trusted. For a more detailed description on the contents of a digital certificate we refer the reader to the X.509 specification [17].

Let us consider as an example the generation of a golden integrity measurement for a specific hypervisor version. During the verification phase of the security development life-cycle an independent and trusted authority would perform security checks on the software. In case those verifications were successful, it would generate a golden integrity measurement for the software and sign it (i.e., encrypt the hash code of the software) using its private key. The public key of this trusted authority can be obtained through its public-key certificate.

It is important to clarify that using golden integrity measurements to offer trustworthy software does not mean only a software version is valid. A version is considered valid as long as the golden integrity measurement in use as a valid signature attached to it.

For a cloud consumer to verify the trustworthiness of the hypervisor it just needs to have in his possession the correspondent golden integrity measurement and the trusted authority's public-key certificate. When communications are initiated with the cloud provider's infrastructure, the consumer receives a fresh signed integrity measurement of the hypervisor in use together with public-key certificate from the cloud provider. For the purpose of this discussion it is enough to know there is a digital certificate linked with the cloud provider, more details about which certificate and why we can trust the provided integrity measurement are given later.

The consumer can then check if the received integrity measurement matches the golden integrity measurement. If there is a match then the consumer can trust the hypervisor version execution in the cloud infrastructure. The public keys of both trusted authority and cloud provider allow the client to verify the authenticity of the measurements used in this verification.

Managing Integrity Measurements

This section discusses how our architecture possesses components capable of generating and collecting integrity measurements, and offers tamper resistant or tamper evident storage for integrity measurements. These requirements are satisfied through the support for hardware-based security such as Intel's Trusted Execution Technology (TXT). Therefore, the servers require the presence of a Trusted Platform Module (TPM).

The TPM-chip provides the foundation for hardware-based security, and contains cryptographic functional units (e.g., random number generator), non-volatile, and volatile storage. It is tamper-proof and the hardware root of trust in a trustworthy environment. A root of trust is a hardware or software mechanism that the user implicitly trusts [29]. The definition of trust considered in this document comes from the Trusted Computing Group (TCG): "An entity can be trusted if it always behaves in the expected manner for the intended purpose.". The TPM generates, collects, and offers the tamper resistant storage for integrity measurements. However, using the TPM directly to manage integrity measurements has proved to be a scalability and performance issue [55].

Our approach to address the performance issues related to intensive use of the TPM is combining TPM functionality with a minimal software implementation of the TPM standard denominated *micro-TPM*. The concept of a *micro-TPM* was introduced in TrustVisor as a solution to offer fast integrity measurements of *pieces of application logic (PAL)* [54]. For the use of a software-based *micro-TPM* to be successful it must be combined with the specific functionality from the hardware-base TPM. The relevant TPM functionality in this case is transitive trust, which is described with some detail in Subsection 2.5.2. In what follows, we discuss in detail how these two requirements allow for trustworthy management of a platform's integrity measurements.

The first requirement is to extend the hypervisor to implement the concept of a *micro-TPM*. This is necessary because frequent usage of hardware support for obtaining integrity measurements incurs in high performance overhead as demonstrated in Flicker [55]. A *micro-TPM* avoids this problem by executing at high speed in a platform's primary CPU, while offering a restricted set of the functionality from the TPM specification. A *micro-TPM* implementation should offer basic randomness, measurement, attestation, and data sealing capabilities. The *micro-TPM* is the mechanism used to generate and collect integrity measurements. In terms of tamper evident storage we can either use the *micro-TPM* data sealing capabilities, or the TPM's own data storage.

Trustworthy computing can use a static or a *Dynamic Root of Trust for Measurement (DRTM)*. The *Static Root of Trust for Measurement (STRM)* was the first generation of trusted computing. Typically, BIOS code was used as the STRM. When using a STRM the trusted computing base (TCB) included BIOS, Option ROMS, Bootloader, OS and applications [29, 42]. Guaranteeing a secure boot means trusting every line of code in the software involved in the boot sequence. Including all these elements creates a considerably large TCB, which is not beneficial for the security of the platform.

The advent of new technology mechanisms, such as extensions for AMD's Secure Virtual Machine (SVM) and Intel Trusted Execution Technology (TXT) extensions to the x86 architecture brought with them the concept of *Dynamic Root of Trust for Measurement (DRTM)*.

This second generation allows the late launch of a measured and isolated block of code. It is denominated late launch because it can be executed at any time. The SKINIT (AMD) and SENTER (Intel) instructions allow a disruptive event (soft reset) to happen. These instructions atomically initialize the CPU to a state similar to INIT, load the Secure Loader Block (SLB) code (or authenticated code SINIT module when using Intel TXT), enable DMA protection for the entire SLB, send the SLB contents to the TPM, and transfer control to the SLB. The SLB code can then proceed to launch applications starting from a trustworthy state. This new method removes BIOS, Option ROMS and Bootloader from the TCB.

In our architecture, the integrity measurements follow a two-level approach as introduced in TrustVisor [54]. The hardware-based security TPM-chip is used in a DRTM process to obtain an integrity measurement of the hypervisor and store it in its physical PCRs. Transitive trust happens in this stage with the addition of the hypervisor to the set of trusted entities in a particular cloud server. This step removes the need for constant use of a platform's TPM consequently suppressing the performance overhead incurred from such use. The *micro-TPM* capabilities of the hypervisor can be used to generate and store integrity measurements for security sensitive *pieces of application logic (PAL)*.

A piece of application logic, or PAL, needs to be registered for the system to put in

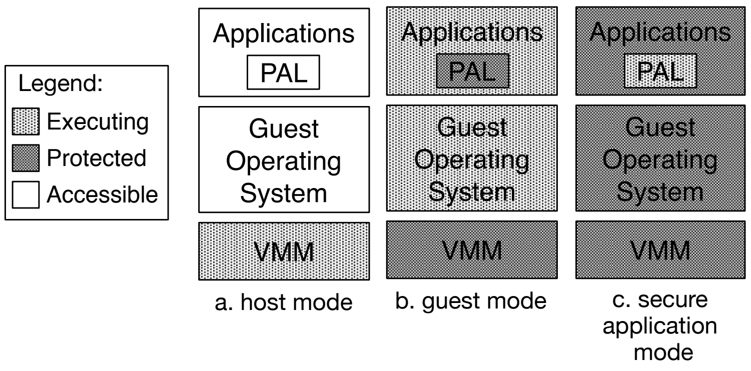


Fig. 6.4 System execution modes.

place the appropriate security measures for its execution. This security measures assure data confidentiality and integrity, and code and *execution integrity*. Execution integrity consists in guaranteeing that when code P executes with inputs P_{inputs}, it always generates outputs $P_{outputs}$ [54]. The processes a PAL goes through include registration, invocation, termination, and removal.

Figure 6.4 illustrates the three existing execution modes and how memory protection differs in each other. The highest privilege level is granted in *host mode* when the virtualization layer is executing and has control of the system. The virtualization layer has control over all of the system memory, which means it can manipulate the memory space of PALs, guest operating systems, and the applications running on top of guest operating systems. A guest operating system executes in *guest mode*, when the system is in this mode the virtualization layer must protect its memory space and the memory space associated to PALs. The most restrictive execution mode is the *secure application mode*, which is when a PAL is executing. In this mode, the virtualization layer isolates the PAL's memory space from all the remaining system entities.

The registration process is already secure against an insider threat as demonstrated in our security analysis of the proposed architecture. Since security sensitive operations are performed in a special-purpose virtual machine, the remaining security concerns are related to the operating system over which those operations are executed. The registration process is guaranteed through an application-level hypercall interface. Developers can use such an interface to register sets of functions as security sensitive.

A registration process consists in specifying a list of function entry points, and expected input and output parameters. The virtualization layer is responsible for verifying that the provided entry points belong to the calling application, while guaranteeing that the executing operating system does not perform any illegal access to the memory pages specified as security sensitive. Anyway, the security sensitive code must have a golden integrity measurement which is going to allow consumers to verify its integrity.

Invoking a PAL happens as if the affected application is executing normally on top of the hosting operating system. However, after registration, the memory space of a PAL is no longer accessible to the owning application and hosting operating system. The virtualization layer is responsible for handling the operations when a PAL is invoked in within its owning application. Once a PAL is invoke, the virtualization layer assumes control and prepares the environment for a PAL's execution as follows: (1) locate the registered PAL to which the executing sensitive code belongs to, (2) change from *guest mode* to *secure application mode* so the memory access is restricted to the memory pages of the executing PAL, and (3) prepare the execution environment so control can be passed to the executing PAL.

After a PAL's execution is concluded it returns to the calling application, at this stage control is again passed to the virtualization layer. When executing in *secure application mode* any attempt to execute code that does not belong to the PAL memory pages results in returning control to the virtualization layer. The virtualization layer can then process the PAL's output results and make them available to the calling application. Execution mode is changed from *secure application mode* to *guest mode*, in which a PAL's memory space is no longer accessible. The calling application can recall the PAL at any time with different input parameters.

The removal of a PAL can originate from the application that requested its registration or in special cases from the operating system in which it executes. When a PAL is removed from the security sensitive code list its memory pages are zeroed and made available to the controlling operating system.

6.3 Cloud Platform Trustworthiness

In this section we present how the trustworthiness of a cloud platform can be shared with cloud consumers increasing the level of transparency when compared with current approaches.

To protect the confidentiality and integrity of data kept in the cloud, the cloud platform has to prevent certain attacks and give consumers the ability to assess that this protection is in place. The latter requirement may seem excessive, but it arises from the concern we are dealing with: the insider threat. A malicious insider is in a sense part of the cloud, so he or she can provide false information to the consumer. In our solution, trust is grounded on hardware – the TPM or CPU extensions for DRTM – instead of cloud operators.

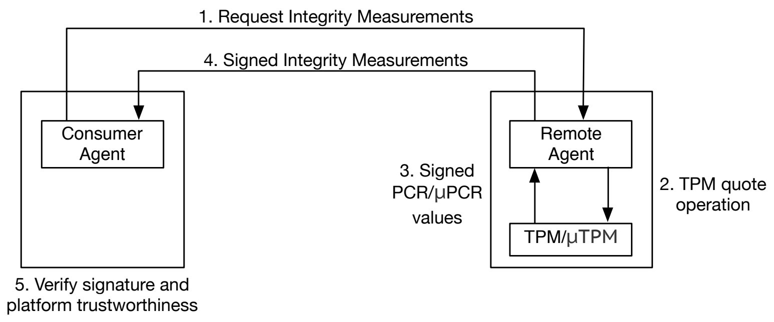

1. Request Integrity Measurements
4. Signed Integrity Measurements
Consumer Agent
Remote Agent
3. Signed PCR/μPCR values
2. TPM quote operation
TPM/μTPM
5. Verify signature and platform trustworthiness

Fig. 6.5 Remote attestation.

6.3.1 Trusted Virtualization Environment

Our solution for protecting consumers' data (and applications) in the cloud is based on the assumption that the attacks against the VM come through the infrastructure and not from targeting vulnerabilities in the consumer VMs themselves.

There are two basic principles for a VM to execute in a trustworthy fashion: (1) a consumer VM is either encrypted or executing in a *Trusted Virtualization Environment (TVE)* and (2) before a VM is decrypted and launched in a TVE, the consumer attests that it can trust the TVE, i.e., that the TVE is indeed a trusted virtualualization environment.

Consumer VMs reside in the cloud in three places: in cloud servers, in the network (e.g., during deployment and migration), and backed up on disks. To prevent data disclosure and modification, we have to limit what a system administrator can do with a VM on a server and force it to be encrypted on the network and when backed up. Therefore, servers must run a TVE.

A *trusted virtualization environment (TVE)* is the infrastructure software that manages a single cloud server. It is basically another way of referring to the trusted computing base of a virtualization cloud server. Figure 6.1 makes it clear that the main components of the TVE are the virtualization layer and the management software. The software configuration of a VM server is said to be a TVE if: (1) the management software does not offer administrator operations that can directly compromise data confidentiality or integrity, and (2) it supports only trusted versions of the critical software components part of the trusted computing base (e.g., hypervisor and management software.). Referring to one TVE is a simplification; cloud consumers can trust several TVE configurations.

Remote Attestation

The basic operation on which a consumer relies to verify a trusted virtualization environment (principle 2) is remote attestation. The process consists in collecting a set of measurements to verify the current configuration of the target cloud server. Since our solution uses DRTM-based attestation, initial trust is established when the consumer receives an integrity measurement for the hypervisor controlling the server.

Figure 6.5 provides an abstraction of the remote attestation process between a consumer (left) and a cloud server (right). Consider that the challenging consumer uses his own computer controlled in some manner, the security of this computer is out of the scope of our work.

The remote attestation process is as follows: First, the consumer requests the server where the hypervisor is running to send an integrity measurement of the hypervisor (1). Second, a software agent in the server requests a quote operation from the TPM (2). Third, the TPM replies with signed values of the platform configuration register that contains the integrity measurement of the hypervisor (3). The consumer may provide a nonce to be included in the signed data in order to assure freshness. Fourth, a message containing the signed values is send to the consumer (4). Fifth, the consumer verifies the signature and if the integrity measurement matches a trustworthy hypervisor version (5). This process allows consumers to verify if the hypervisor running in a cloud server is trustworthy.

The behaviour of a consumer software agent on step five depends on the result of verifying the validity of the receive integrity measurement(s):

- If the integrity measurement matches the consumer implicitly trusts the software executing in the cloud infrastructure.
- In the event of receiving an integrity measurement that matches an older golden measurement the consumer might trust the software if it is not associated with malicious activity, but otherwise notifies the cloud provider about the event and chooses to not trust the software.
- In case of a received integrity measurement matching an older golden measurement associated with malicious activity, the consumer notifies the cloud provider to update the platform and terminates the communication.

The remote attestation process just described is the most traditional one in which the physical TPM is used to report integrity measurements on platform components. However, in our architecture, transitive trust has a significant impact. When a consumer agent trusts the hypervisor, this means that the hypervisor is added to the trusted computing base and its *micro-TPM* capabilities are then used to generate and collect future integrity measurements of a cloud platform components. For example, the *micro-TPM* could be used to store a copy of the integrity measurement for the hypervisor in order to speed up its verification process.

6.3.2 Critical Management Operations

A description of virtual machine launch, migration, backup, and termination and how they are performed in the trustworthy architecture follows [72]. This description does not intend to include every critical operation that takes place in a cloud architecture, it simply presents these four operations as examples of how a trustworthy architecture provides more transparency while at the same time it protects a consumer's security sensitive data.

Any other critical operations will be protected in a similar manner. This type of functionality is also an example of software that is going to be measured using the *micro-TPM* capabilities. The TPM itself is going to be used in the boot process of cloud servers to keep an integrity measurement of the executing hypervisor. This measurement is used in the initial trust establishment between consumer and cloud provider. Trusting the hypervisor is paramount because it is the root of trust for obtaining further integrity measurements of software agents like the ones discussed in this section. Establishing trust in a hypervisor executing in a remote cloud server is achieved through a standard remote attestation process as the one represented in Figure 6.5. However, the integrity measured obtained from the physical TPM can then passed to the trusted software *micro-TPM* implementation.

To explain the operations discussed in this section, it is assumed the existence of a software agent on each end of the communicating parties. This software agent will be responsible for performing the steps that take place in that party. The operations are divided in several steps, which are numerically represented in the figures throughout this subsection. The number for each step is listed inside parenthesis after the corresponding written description.

Virtual Machine Launch

In a trustworthy virtual machine launch operation, the final objective is to have the consumer trust the software agents in the server responsible for handling his virtual machine and the data he might offload to it.

Figure 6.6 depicts the complete process. The process starts with the consumer software

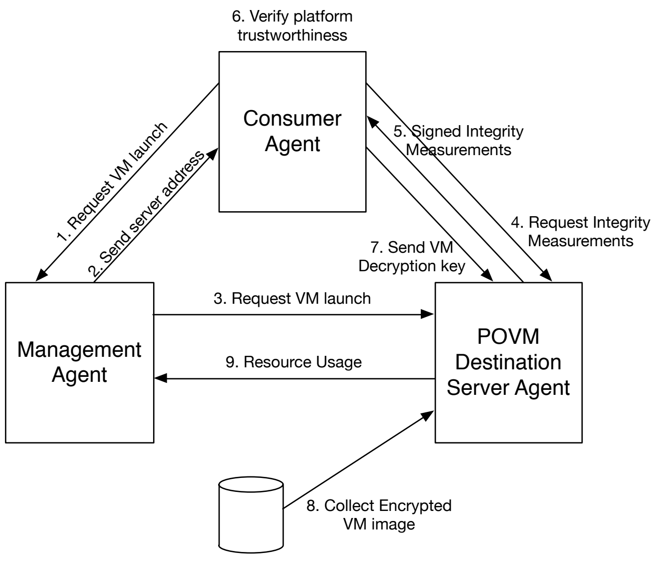

6. Verify platform trustworthiness
Consumer Agent
1. Request VM launch
2. Send server address
5. Signed Integrity Measurements
4. Request Integrity Measurements
7. Send VM Decryption key
Management Agent
3. Request VM launch
9. Resource Usage
POVM Destination Server Agent
8. Collect Encrypted VM image

Fig. 6.6 Trustworthy VM Launch.

agent contacting the software agent resident in the front-end server (1), which is responsible for managing the resources available in the infrastructure. This is a simple interaction where the consumer requests to launch a virtual machine and the front-end agent replies with the physical address for an available cloud server (2).

The second stage takes place between the consumer and the destination cloud server. First, the consumer requests a remote attestation of the platform where his virtual machine is about to be launched (4). Second, after requesting a quote operation from the server's *micro-TPM*, the cloud server agent replies with a signed message containing the required integrity measurements (5). Let us assume in this example scenario that the server sends measurements for the hypervisor and the software responsible for launching the virtual machine. The VM launch software is executed in the privileged operations virtual machine.

After these steps, the consumer verifies the signature and checks if the integrity measurements correspond to the current golden integrity measurements for the hypervisor and VM launch routine (6). If the verification is successful the consumer establishes a secure communication channel with the privileged operations virtual machine and sends the symmetric key necessary to decipher the encrypted virtual machine image (7). An approach for exchanging the key could be using the public portion of an attestation identity key associated with the *micro-TPM* of the destination server. The virtual machine image can be sent by the consumer or be collected from a local repository (8). Finally, the VM image is deciphered using the key the consumer provided (9) and the process is concluded with the launch of the virtual machine in the cloud server (10).

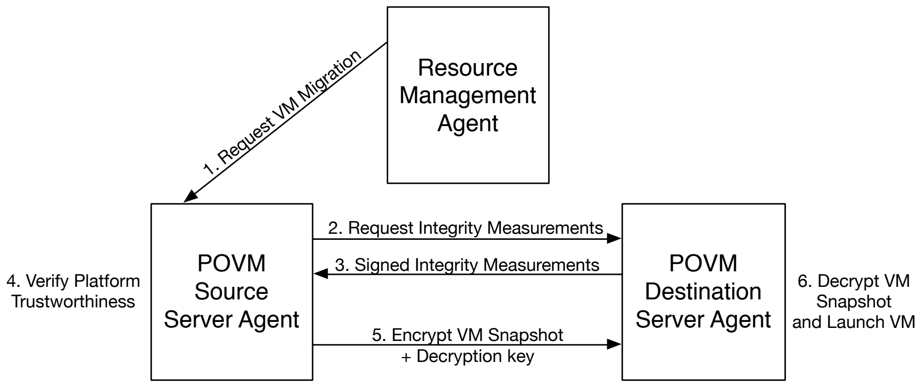

Resource Management Agent
1. Request VM Migration
POVM Source Server Agent
POVM Destination Server Agent
2. Request Integrity Measurements
3. Signed Integrity Measurements
4. Verify Platform Trustworthiness
5. Encrypt VM Snapshot + Decryption key
6. Decrypt VM Snapshot and Launch VM

Fig. 6.7 Trustworthy VM Migration.

The communications between consumer agent and privileged operations virtual machine software agent are handled in the destination server agent which can be software part of the management software of the server. This management software is part of the management virtual machine as seen in Figure 6.1. In step seven of Figure 6.6, the secure communications channel between privileged operations virtual machine can use inter-virtual machine communication solutions to avoid bloating the trusted computing base in the privileged operations VM. The advantages and performance of using such channels to handle communications between virtual machines was already discussed in previous work [107].

Virtual Machine Migration

From the consumer's perspective, verifying if the initial server (where his virtual machine is instantiated) is trustworthy is not enough because virtual machines can be migrated to different cloud servers within the cloud infrastructure. However, this problem can be easily solved by having the source cloud server verify if the destination server is trustworthy. The consumer has already verified that the initial server is trustworthy, so that guarantees that the virtual machine migration operation can be trusted to perform the steps represented in Figure 6.7. The communications between the involved servers are handled in the respective management virtual machine. The data can then be sent to the privileged operations virtual machine using inter-virtual machine communication channels in order to keep the trusted computing base to a minimum.

Consider a scenario where an attacker tries to migrate a virtual machine from a trustworthy server to a platform over which he has total control. This attack would not go through because the source server would reject the migration when the integrity measurements do not match a trustworthy virtualization environment. Furthermore, the privileged operations

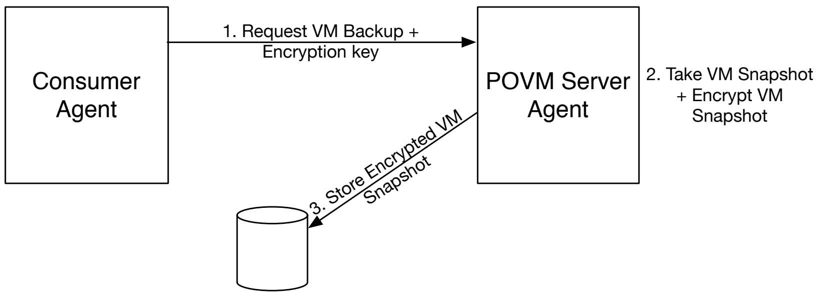

Consumer Agent
1. Request VM Backup + Encryption key
POVM Server Agent
2. Take VM Snapshot + Encrypt VM Snapshot
3. Store Encrypted VM Snapshot

Fig. 6.8 Trustworthy VM Backup.

virtual machine should have software agents responsible for logging and auditing these operations to detect any anomalies.

Under these assumptions the trustworthy virtual machine migration process is a simple remote attestation of the destination server performed by the source cloud server. Figure 6.7 illustrates the required steps. First, the resource management agent requests a migration (1). Second, the source server requests a remote attestation of the destination server's platform software (2). Next, the destination server requests a quote operation from its *micro-TPM*.

On the destination server, a signed list of the requested integrity measurements is sent to source server (3). What follows is the verification of the signature and the platform software (4). If the verification is successful a ciphered snapshot of the virtual machine together with the decryption key is sent to the destination server (5). Finally, the destination server deciphers the virtual machine snapshot and the VM is launched in the new trustworthy server (6).

Virtual Machine Backup

Although some data confirms that cloud storage services might experience outages, one of the major attractive points of the cloud is the always available promise [3]. The notion is that you store data in the cloud and then you can access it from various client platforms as long as it is capable of interfacing with the online storage system. With a service that offers such high availability it is expected that consumers will be drawn to using it as a backup resource. However, backing up a consumers data also presents some security requirements which the approach that follows satisfies.

The assumption for this operation is that the server where the VM is executing is already trusted by the consumer. If the VM snapshot needs to go through a backup server then a process similar to a VM migration takes place before the backup is performed.

In Figure 6.8, it is possible to view the parties and steps involved in backing up a ciphered

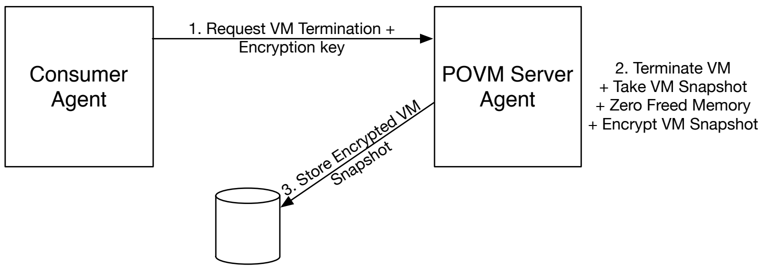

Consumer Agent
1. Request VM Termination + Encryption key
POVM Server Agent
2. Terminate VM + Take VM Snapshot + Zero Freed Memory + Encrypt VM Snapshot
3. Store Encrypted VM Snapshot

Fig. 6.9 Trustworthy VM termination.

instance of a consumer's virtual machine. The operation is quite simple. First, the consumer agent sends a message containing the encryption key and a VM backup request to the cloud server agent (1). Second, the server agent takes a snapshot of the VM to backup, encrypts it using the consumer's key (2). Finally, and still in the server agent, the encrypted VM backup is stored in the appropriate storage media (3).

Virtual Machine Termination

Terminating a virtual machine is fairly simple. However, it is relevant to discuss this operation because it has some security implications. More specifically, due to the memory space allocated to the VM throughout its life cycle.

Consider a scenario where an application (e.g., Apache Web Server) executing in a virtual machine recurs to a private key to sign the values in a Diffie-Hellman key exchange. This means that the private key will be present in volatile memory for the signing operation. Therefore, if after termination this memory space is not zeroed or randomized an attacker can assemble a special purpose Linux image designed to corrupt as little memory space as possible so it can collect security sensitive data from the remainder of the memory.

The termination operation follows the steps represented in Figure 6.9. First, the consumer requests a termination directly to the server where the VM is executing (1). At this stage the consumer also informs the provider if he wishes for his VM image be encrypted and stored in the cloud infrastructure. If this is true the consumer will annex the symmetric encryption key to the request.

Second, the cloud server terminates the VM, zeroes or randomizes the memory space, and encrypts (2) and stores the VM image in a local repository (3). The zeroing of the memory space should be done in the termination process because the operation is associated with the VM in question. Therefore, it is ideal to perform it at this stage, and not when a new VM is launched, so the owner of the VM can be assured that its security sensitive information is secure. The operation is concluded when the cloud server notifies the management agent of the termination and the freed resources.

6.4 Related Approaches

This section discusses work that is closely related to the cloud ecosystem we propose in this Chapter. It presents work that trusted computing technologies as a tool for securing both virtualization and cloud computing.

TrustVisor is a virtual machine monitor built for commodity systems with the objective of assuring code and data integrity for portions of executing applications [54]. This virtual machine monitor implements a software *micro-TPM* which allows challenging entities to verify the trustworthiness of a specific functionality in a remote system.

IBM's *trusted virtual datacenter* (TVDc) approach aims at creating virtual domains of virtual machines within a datacenter [7]. This solutions builds on the sHype hypervisor which enforces mandatory memory access control (MAC) policies to guarantee isolated trusted virtual domains [76]. This approach does not target cloud computing.

In Trusted Cloud Computing Platform (TCCP) the authors propose the use of trusted computing base to improve the security of a cloud computing environment [79]. This solution uses a node known as *trusted coordinator* to keep track of the trusted nodes in a cloud environment. Trusted nodes are cloud server with a trusted platform configuration.

The myTrustedCloud architecture integrates the offering of trustworthy cloud services with the Eucalyptus cloud computing platform [103]. The trustworthy services implemented include remote attestation of both virtual machines and elastic block storage.

The first two approaches discussed in this section were not about cloud computing. This separates them from our contribution in this chapter. The cloud ecosystem we propose, much like TrustVisor, tries to offer fine granularity when verifying the integrity of cloud services. Another point that distinguishes our work from IBM's TVDc is that we propose security by default without any administrator responsible for devising security policies.

The last two works are very closely related to what we propose in this chapter, the main distinction is our focus on achieving integrity verification of building blocks for cloud services. Building blocks refer to smaller components of a complex cloud ecosystem such as VM launch or migration.

The main concern for our approach is to keep the trusted computing base of measured components to a minimum and then combine such components to build a more complex ecosystem. This differs with other approaches which centre on measuring larger entities such as virtual machines or only verifying the presence of a trusted virtual machine monitor. The novelty about our proposal is assuring the integrity of cloud services with more granularity. The cloud operation discussed in Section 6.3 are examples of the granularity we suggest.

6.5 Limitations

In this section we introduce limitations identified in the cloud ecosystem proposed in this chapter. We discuss three limitations that can have a negative impact in the security of the proposed architecture.

The first problem is the challenges of managing of golden integrity measurements. A cloud consumer trusts he/she can use golden integrity measurements to verify the trustworthiness of a remote cloud computing platform. However, golden integrity measurements are also going to need to be updated. The process of revoking old golden integrity measurements and releasing new ones can be a challenging one. This process might end up opening attack windows.

Another problem is the presence of software vulnerabilities in cloud services that are considered trustworthy. The cloud ecosystem we propose here does not address this problem. It is true that we try to keep the trusted computing base of measured entities to a minimum but that still does not assure that those entities are free from vulnerabilities.

The final problem we mention is a problem that transfer from public-key infrastructures where the endpoint systems are left with trusting the entities at the root of the chain of trust. This means that a cloud consumer is left with trusting that the entity responsible for verifying the software and signing golden integrity measurements is trustworthy and does not collude with cloud providers to compromise data confidentiality and integrity.

Chapter 7

Conclusions

The work in this thesis had two practical research stages. Initial work focused on demonstrating how current virtualization layer software is not effective at preventing attacks originating from a malicious insider. Whereas the second stage was dedicated to designing a novel trustworthy cloud architecture capable of preventing the attacks we had identified as possible in current solutions.

The first stage focused on performing sophisticated attacks against the memory space of virtual machines to compromise security sensitive consumer data. Our attacks were successful against three of the major providers of software solutions for the virtualization layer. Identifying the same vulnerability in three different solutions builds a strong argument in favour of the issue being a security design problem and not a simple software development flaw in certain virtualization layer software solutions.

The second and final stage consisted in designing and testing a novel cloud architecture that could prevent the malicious insider attacks previously identified in our work. In this stage we used a particular virtualization software and modified its memory management mechanisms to turn the hypervisor into the policy decision and enforcement point in the architecture. This change is paramount to guarantee a more trustworthy architecture while reducing its trusted computing base.

This chapter discusses our main contributions and is organised in two major sections. In Section 7.1 we discuss our major contributions and how they differ from previous work. Finally, Section 7.2 is dedicated to lines of research we have identified as potential future work.

7.1 Summary of Contributions

This thesis contains three major contributions: a literature review, identified a design flaw in current virtualization layer software solutions, and design and implementation of a novel trustworthy cloud architecture.

Our literature review presents a unique view on current virtualization software solutions. It looks into the security effectiveness of such solutions when an insider threat is considered. Therefore, it offers a different perspective on the security of various state of the art virtualization solutions. More specifically, it analyses if different virtualization software approaches can prevent a malicious insider from compromising security sensitive data and guarantee memory integrity and confidentiality.

The security design flaw we identified is common to the three major virtualization software providers for cloud solutions. Our experiments consisted in devising attacks to compromise security sensitive data while it is resident in random access memory. We performed successful sophisticated attacks capable of compromising data from a virtual machine's memory space in real time. The fact that this permissive memory model is present in the three major virtualization software providers makes it a relevant security issue that needs to be researched. Previous work on this security challenge only identified a security flaw in a specific virtualization solution [73].

The final contributions of this project are connected and include an insider threat prevention mechanism and a trustworthy cloud architecture. We designed, implemented, and tested our security mechanism to prevent a malicious insider from compromising security sensitive data resident in random access memory. The novel contribution is how it reduces the trusted computing base to include only the hypervisor. The hypervisor is turned into the security policy decision and enforcement agent. Our design prevents the insider threat by enforcing the principle of least privilege. This guarantees memory confidentiality and integrity for consumer data resident in random access memory.

Our final contribution uses our insider threat prevention mechanism as a building block in a trustworthy cloud computing architecture. In this architecture, we propose how to build a trustworthy cloud ecosystem that allows remote cloud consumers to verify the trustworthiness of a cloud computing platform. The novelty of this architecture is offering better granularity of integrity verified cloud services.

7.2 Future Work

This section discusses the research challenges we consider the most relevant in a trustworthy cloud computing infrastructure. It is important to mention that some of these challenges are a direct result from the changes imposed to the cloud ecosystem in order to transform the virtualization layer in the policy decision and enforcement entity.

7.2.1 Further Reductions to the Trusted Computing Base

The trustworthy architecture we suggest in Chapter 6 can have further reductions to its trusted computing base. The fact that Xen is a monolithic virtual machine monitor means it has a considerably high number of lines of code when compared with microhypervisor approaches such as the NOVA hypervisor [90].

Further reducing the trusted computing base of the trustworthy cloud computing architecture would mean disaggregating Xen or using a microhypervisor to take advantage of the protection rings in root-mode. These protection rings could be used to isolate virtual machine monitors launched for each individual virtual machine. This means that the critical parts of the hypervisor would execute in root-mode's ring 0 while the less privilege operations could execute in root-mode's ring 3. A virtual machine's guest operating system and its applications would use non-root mode's ring 0 and ring 3, respectively. In such an architecture, the privilege operations currently isolated in a special-purpose virtual machine could be moved to root-mode's ring 3. This change means a mandatory memory access control mechanism can be implemented in a more appropriate way starting from the design stage of a secure microhypervisor architecture.

7.2.2 Uniqueness of Software Agents

This problem originates from one of the assumptions in the current public key infrastructure. The assumption assumes that certificate authorities are trustworthy and will not sign certificates to rogue entities. Such certificates could be used in man in the middle attacks.

This assumption is relevant for our trustworthy architecture with regard to golden integrity measurements. Once again we are left with relying on the trusted authority to only sign measurements for trustworthy software agents. The verification process involves both cloud provider and trusted authority, so a collusion attack might have an undesired impact on the trustworthiness of a cloud platform. It is even more serious if you consider how everything would appear to be certified but at the same time the software controlling the platform would be malicious.

7.2.3 Monitoring Virtual Machines

In previous cloud configurations, management software was granted full access to the memory space assigned to virtual machines executing in a cloud server. We have already shown that this memory access model is too permissive and allows insiders to gain access to security sensitive data. However, this model had the advantage of permitting detailed monitoring of the activities in a consumer's virtual machine.

Our architecture uses the virtualization layer to prevent insiders from obtaining access to the memory space of virtual machines. Therefore, any previous monitoring solutions based on virtual machine introspection are going to fail in such an architecture. The design choices we have made do not hinder the use of monitoring solutions based on virtual machine introspection. However, their method of operation needs to shift to comply with security requirements.

Trustworthy cloud monitoring solutions would have to relocate to the virtual machine allowed to perform privileged operations. Which means that these operations would execute where the critical cloud management operations do. However, the data collected and analysed in random access memory would have to be encrypted before it is stored in any permanent storage media. The ideal encryption scheme would be one in which the cloud provider and consumer need to supply a key in order to access the desired data. This would be required in case this data needs to be accessed to solve any disputes.

7.2.4 Managing Golden Integrity Measurements

A trustworthy cloud architecture (like the one discussed in Subsection 6.2) needs to consider the challenge of managing a set of golden integrity measurements. It will need to at least perform such management tasks for the golden integrity measurements of hypervisor versions available on cloud servers.

The challenge is how to synchronize a list of valid golden integrity measurements between consumer, trusted authority, and cloud provider. Consider a scenario where a hypervisor requires a security fix and a new golden integrity measurement needs to be created and sent to the consumer. This situation creates an attack window during which an attacker can use previously trustworthy hypervisors to mount attacks against consumers.

The problem is aggravated if we consider a cloud provider that offers verification of critical cloud operations [71]. Therefore, the design of an effective method for revoking and releasing new golden integrity measurements is of paramount importance.

7.2.5 Encrypt-on-Save Data

Guaranteeing data confidentiality and integrity while data is loaded and processed in memory is positive but different challenges need to be overcome when the same data is moved to permanent storage devices. The problem of assuring data confidentiality and integrity for permanently stored data was partially discussed in Subsection 7.2.3 when we introduced the challenges faced by data generated from monitoring virtual machines.

Consider a scenario where a consumer is using a text editor to modify a confidential file. This type of software usually has auto save options which we envision as an advanced feature when considering the support for encrypt-on-save functionality. It is ideal that a consumer does not need to worry about the privacy of the information present in his documents while storing or editing them in the cloud. Assuring that data is encrypted throughout its complete life cycle is the final objective to achieving data confidentiality.

References

[1] Alliance, C. S. (2013). The Notorious Nine: Cloud Computing Top Threats in 2013.

[2] AMD (2014). AMD Virtualization (AMD-V) Technology.

[3] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., and Zaharia, M. (2010). A view of cloud computing. *Commun. ACM*, 53(4):50–58.

[4] Azab, A. M., Ning, P., and Zhang, X. (2011). SICE: A Hardware-level Strongly Isolated Computing Environment for x86 Multi-core Platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 375–388, New York, NY, USA. ACM.

[5] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. (2003). Xen and the art of virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 164–177, New York, NY, USA. ACM.

[6] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. (2006). vTPM: virtualizing the trusted platform module. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA. USENIX Association.

[7] Berger, S., Cáceres, R., Pendarakis, D., Sailer, R., Valdez, E., Perez, R., Schildhauer, W., and Srinivasan, D. (2008). TVDc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47.

[8] Bethencourt, J., Sahai, A., and Waters, B. (2007). Ciphertext-policy attribute-based encryption. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 321–334, Washington, DC, USA. IEEE Computer Society.

[9] Bhatia, N. (2009a). Performance Evaluation of AMD RVI Hardware Assist.

[10] Bhatia, N. (2009b). Performance Evaluation of Intel EPT Hardware Assist.

[11] Bishop, M. (2004). *Introduction to Computer Security*. Addison-Wesley Professional.

[12] Bowers, K. D., Juels, A., and Oprea, A. (2009). HAIL: a high-availability and integrity layer for cloud storage. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 187–198, New York, NY, USA. ACM.

[13] Bryant, R. E. and O'Hallaron, D. R. (2010). *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition.

[14] Capelli, D., Moore, A., Trzeciak, R., and Shimeall, T. J. (2009). *Common Sense Guide to Prevention and Detection of Insider Threats*. Software Engineering Institute, 3rd edition edition.

[15] Chisnall, D. (2007). *The Definitive Guide to the Xen Hypervisor*. Prentice Hall Press, Upper Saddle River, NJ, USA, first edition.

[16] Colp, P., Nanavati, M., Zhu, J., Aiello, W., Coker, G., Deegan, T., Loscocco, P., and Warfield, A. (2011). Breaking up is hard to do: security and functionality in a commodity hypervisor. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 189–202. ACM.

[17] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, W. (2008). RFC 5280 - Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. Technical report, The Internet Engineering Task Force.

[18] Denning, P. J. (1971). Third generation computer systems. *ACM Comput. Surv.*, 3(4):175–216.

[19] Department of Defense (1985). *Department of Defense Trusted Computer System Evaluation Criteria*. Department of Defense. DOD 5200.28-STD (supersedes CSC-STD-001-83).

[20] Dick Csaplar (2012). Is the Hypervisor Market Expanding or Contracting? http://goo.gl/X16XN5. The Aberdeen Group [Online; access 13-April-2015].

[21] Diffie, W. and Hellman, M. (2006). New directions in cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654.

[22] European Space Agency - Inquiry Board (1996). Ariane 5 - flight 501 failure.

[23] Fabrice Bellard (2014). QEMU: Open Source Processor Emulator.

[24] Fang, H., Zhao, Y., Zang, H., Huang, H., Song, Y., Sun, Y., and Liu, Z. (2010). Vmguard: An integrity monitoring system for management virtual machines. In *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on*, pages 67–74.

[25] Floyer, D. (2013). Wikibon Hypervisor Study: Additional Analysis of Multi-Hypervisor Environments. http://goo.gl/uaBFFM. Wikibon.org [Online; accessed 13-April-2015].

[26] Fraser, K., H, S., Neugebauer, R., Pratt, I., Warfield, A., and Williamson, M. (2004). Safe hardware access with the xen virtual machine monitor. In *In 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS*.

[27] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. (2003). Terra: a virtual machine-based platform for trusted computing. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 193–206, New York, NY, USA. ACM.

[28] Goldberg, R. P. (1974). Survey of Virtual Machine Research. *IEEE Computer*, 7(9):34–45.

[29] Grawrock, D. (2009). *Dynamics of a Trusted Platform: A Building Block Approach*. Intel Press, 1st edition.

[30] Greene, J. (2010). Intel Trusted Execution Technology: Hardware-based Technology for Enhancing Server Platform Security. Technical report, Intel.

[31] Guo, F. (2011). Understanding memory resource management in vmware vsphere 5.0 - performance study.

[32] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. (2009). Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98.

[33] Heath, N. (2013). Linux trailed Windows in patching zero-days in 2012, report says (ZDNet).

[34] Hoglund, G. and Butler, J. (2005). *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional.

[35] Institute, P. (2014). 2014 cost of cyber crime study: United states.

[36] Intel (2014). Hardware-Assisted Virtualization.

[37] Intel, Inc (2015). Intel R 64 and IA-32 Architectures Software Developer's Manual. *Volume 3a: System Programming Guide*.

[38] International Telecommunication Union (ITU) (2008a). X.680 : Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation.

[39] International Telecommunication Union (ITU) (2008b). X.690 : Information technology - ASN.1 encoding rules: Specification of Basic En- coding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER).

[40] International Telecommunication Union (ITU) (2014). ASN.1 Project.

[41] Juels, A. and Oprea, A. (2013). New approaches to security and availability for cloud data. *Commun. ACM*, 56(2):64–73.

[42] Kauer, B. (2007). Oslo: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 16:1–16:9, Berkeley, CA, USA. USENIX Association.

[43] Keeney, M. (2005). *Insider threat study: Computer system sabotage in critical infrastructure sectors*. US Secret Service and CERT Coordination Center.

[44] Keller, E., Szefer, J., Rexford, J., and Lee, R. B. (2010). NoHype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 350–361, New York, NY, USA. ACM.

[45] Kerckhoffs, A. (1883). La cryptographie militaire. *Journal des Sciences Militaires*, pages 161–191.

[46] Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., and Zolotarov, V. (2014). OSv—Optimizing the Operating System for Virtual Machines. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA. USENIX Association.

[47] Kocher, P. C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK. Springer-Verlag.

[48] Kohnfleder (1978). *Towards a Practical Public Key Cryptosystem.* MIT Department of Electrical Engineering.

[49] Krautheim, F. J. (2009). Private virtual infrastructure for cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA. USENIX Association.

[50] Li, C., Raghunathan, A., and Jha, N. (2010). Secure virtual machine execution under an untrusted management os. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 172 –179.

[51] Lipner, S. (2004). The Trustworthy Computing Security Development Lifecycle. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 2–13, Washington, DC, USA. IEEE Computer Society.

[52] Love, R. (2010). *Linux Kernel Development.* Addison-Wesley Professional, 3rd edition.

[53] Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., and Crowcroft, J. (2013). Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA. ACM.

[54] McCune, J. M., Li, Y., Qu, N., Zhou, Z., Datta, A., Gligor, V. D., and Perrig, A. (2010). Trustvisor: Efficient TCB reduction and attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland 2010).*

[55] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Isozaki, H. (2008). Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference in Computer Systems (EuroSys).*

[56] McVoy, L. and Staelin, C. (1996). Lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA. USENIX Association.

[57] Mell, P. and Grance, T. (2011). The NIST definition of Cloud Computing.

[58] Misra, S. C. and Bhavsar, V. C. (2003). Relationships between selected software measures and latent bug-density: Guidelines for improving quality. In *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI*, ICCSA'03, pages 724–732, Berlin, Heidelberg. Springer-Verlag.

[59] Murray, D., Milos, G., and Hand, S. (2008). Improving xen security through disaggregation. In *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 151–160, New York, NY, USA. ACM.

[60] National Institute of Standards and Technology (1995). An Introduction to Computer Security: The NIST Handbook. Technical report, National Institute of Standards and Technology, Washington.

[61] Nexenta (2012). Server Hypervisor Market Share Survey. http://goo.gl/xPfpTH. UP2V.nl [Online; accessed 13-April-2015].

[62] Nguyen, A., Raj, H., Rayanchu, S., Saroiu, S., and Wolman, A. (2012). Delusional boot: Securing hypervisors without massive re-engineering. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 141–154, New York, NY, USA. ACM.

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

[64] Open HUB (2015). Linux Kernel. http://goo.gl/YYmmyo. Open HUB [Online; accessed 13-April-2015].

[65] Payne, B. D., Carbone, M., and Lee, W. (2007). Secure and flexible monitoring of virtual machines. *Computer Security Applications Conference, Annual*, 0:385–397.

[66] Perrig, A. (2010). A clean-slate design for the next-generation secure internet (lecture notes).

[67] Red Hat (2014a). libvirt: Virtualization API.

[68] Red Hat (2014b). Linux Kernel Based Virtual Machine.

[69] Regenscheid, A. and Scarfone, K. (2011). SP 800-155. BIOS Integrity Measurement Guidelines (draft). Technical report, National Institute of Standards & Technology, Gaithersburg, MD, United States.

[70] Rivest, R. L., Shamir, A., and Adleman, L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126.

[71] Rocha, F., Abreu, S., and Correia, M. (2011). The final frontier: Confidentiality and privacy in the cloud. *Computer*, 44:44–50.

[72] Rocha, F., Abreu, S., and Correia, M. (2013a). *The Next Frontier: Managing Data Confidentiality and Integrity in the Cloud.* IEEE Computer Society Press.

[73] Rocha, F. and Correia, M. (2011). Lucy in the sky without diamonds: Stealing confidential data in the cloud. In *Proceedings of the 1st International Workshop on Dependability of Clouds, Data Centers and Virtual Computing Environments*, DSNW '11, Hong Kong.

[74] Rocha, F., Gross, T., and van Moorsel, A. (2013b). Defense-in-depth against malicious insiders in the cloud. In *IEEE International Conference on Cloud Engineering 2013*, San Francisco, USA.

[75] RSA Laboratories (2002). PKCS no.1 v2.1: RSA Cryptography Standard.

[76] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., and Doorn, Leendert van (2005). Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In *Proceedings of the 21st Annual Computer Security Applications Conference*, ACSAC '05, pages 276–285, Washington, DC, USA. IEEE Computer Society.

[77] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, Leendert (2004). Design and Implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA. USENIX Association.

[78] Saltzer, J. and Schroeder, M. (1975). The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308.

[79] Santos, N., Gummadi, K. P., and Rodrigues, R. (2009). Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, Berkeley, CA, USA. USENIX Association.

[80] Santos, N., Rodrigues, R., Gummadi, K. P., and Saroiu, S. (2012). Policy-sealed data: a new abstraction for building trusted cloud services. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 10–10, Berkeley, CA, USA. USENIX Association.

[81] Scarfone, K. A., Souppaya, M. P., and Hoffman, P. (2011). SP 800-125. Guide to Security for Full Virtualization Technologies. Technical report, National Institute of Standards and Technology, Gaithersburg, MD, United States.

[82] Schneier, B. (1993). *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., New York, NY, USA.

[83] Seshadri, A., Luk, M., Qu, N., and Perrig, A. (2007). SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 335–350, New York, NY, USA. ACM.

[84] Srivastava, A. and Giffin, J. (2008). Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, pages 39–58, Berlin, Heidelberg. Springer-Verlag.

[85] Srivastava, A., Raj, H., Giffin, J., and England, P. (2012). Trusted VM snapshots in untrusted cloud infrastructures. In *Proceedings of the 15th international conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 1–21, Berlin, Heidelberg. Springer-Verlag.

[86] Stallings, W. (2010a). *Cryptography and Network Security: Principles and Practice*. Prentice Hall Press, Upper Saddle River, NJ, USA, 5th edition.

[87] Stallings, W. (2010b). *Network Security Essentials: Applications and Standards*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition.

[88] Stanley R. Ames Jr (1981). Security Kernels: A Solution or a problem? In *Proceedings of the IEEE Symposium on Security and Privacy*.

[89] Stefanov, E., van Dijk, M., Juels, A., and Oprea, A. (2012). Iris: a scalable cloud file system with efficient integrity checks. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 229–238, New York, NY, USA. ACM.

[90] Steinberg, U. and Kauer, B. (2010). NOVA: a microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, EuroSys '10, pages 209–222, New York, NY, USA. ACM.

[91] Strongin, G. (2005). Trusted Computing Using AMD "Pacifica" and "Presidio" Secure Virtual Machine Technology. *Inf. Secur. Tech. Rep.*, 10(2):120–132.

[92] Tal Garfinkel and Mendel Rosenblum (2003). A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *In Proc. Network and Distributed Systems Security Symposium*, pages 191–206.

[93] Tasker, P. S. (1981). Trusted computer systems. In *Proceedings of the IEEE Symposium on Security and Privacy*.

[94] Trusted Computing Group (2007a). TCG Mobile Reference Architecture.

[95] Trusted Computing Group (2007b). Trusted Computing Group - TCG Architecture Overview, Version 1.4.

[96] Trusted Computing Group (2011). Trusted Computing Group - TPM Main Specification Level 2 Version 1.2, Revision 116: Part 3 - Commands.

[97] Trusted Computing Group (2014a). Trusted Computing Group - About TCG.

[98] Trusted Computing Group (2014b). Trusted Computing Group - Developers - FAQ.

[99] Uhlig, R., Neiger, G., Rodgers, D., Santoni, A. L., Martins, F. C. M., Anderson, A. V., Bennett, S. M., Kagi, A., Leung, F. H., and Smith, L. (2005). Intel virtualization technology. *Computer*, 38(5):48–56.

[100] Van Dijk, M. and Juels, A. (2010). On the impossibility of cryptography alone for privacy-preserving cloud computing. In *Proceedings of the 5th USENIX conference on Hot topics in security*, HotSec'10, pages 1–8, Berkeley, CA, USA. USENIX Association.

[101] Vasudevan, A., McCune, J. M., Qu, N., van Doorn, Leendert, and Perrig, A. (2010). Requirements for an Integrity-protected Hypervisor on the x86 Hardware Virtualized Architecture. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, TRUST'10, pages 141–165, Berlin, Heidelberg. Springer-Verlag.

[102] VMWare (2007). Understanding Full Virtualization, Paravirtualization, and Hardware Assist. http://goo.gl/kfREyp. VMWare white paper. [Online; access 13-April-2015].

[103] Wallom, D., Turilli, M., Taylor, G., Hargreaves, N., Martin, A., Raun, A., and McMoran, A. (2011). myTrustedCloud: Trusted Cloud Infrastructure for Security-critical Computation and Data Managment. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 247–254.

[104] Whitaker, A., Shaw, M., and Gribble, S. (2002). Denali: Lightweight Virtual Machines for Distributed and Networked Applications. In *Proceedings of the 2002 USENIX Annual Technical Conference*.

[105] Xen Project (2014). Xen Project Software Overview. http://goo.gl/yTaLaA.

[106] Zhang, F., Chen, J., Chen, H., and Zang, B. (2011). Cloudvisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 203–216, New York, NY, USA. ACM.

[107] Zhang, X., McIntosh, S., Rohatgi, P., and Griffin, J. L. (2007). XenSocket: A High-throughput Interdomain Transport for Virtual Machines. In *Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, Middleware '07, pages 184–203, New York, NY, USA. Springer-Verlag New York, Inc.

[108] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2012). Cross-VM side channels and their use to extract private keys. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 305–316, New York, NY, USA. ACM.