D2.1.1
Technical Requirements and Architecture for Privacy-enhanced and Resilient Trusted Clouds

Project number:	257243
Project acronym:	TClouds
Project title:	Trustworthy Clouds - Privacy and Resilience for Internet-scale Critical Infrastructure
Start date of the project:	1st October, 2010
Duration:	36 months
Programme:	FP7 IP

Deliverable type:	Report
Deliverable reference number:	ICT-257243 / D2.1.1 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.1
Due date:	September 2011 – M12
Actual submission date:	3rd October, 2011

Responsible organisation:	SRX
Editor:	Norbert Schirmer
Dissemination level:	Public
Revision:	1.0

Abstract:	This report describes requirements for a trusted cloud infrastructure, analyses shortcomings of todays cloud offerings and develops new approaches and building blocks for such an infrastructure, focusing on privacy and resilience.

Keywords:	Security Analysis, Gap Analysis, Requirements and Building Blocks for Trusted Cloud Infrastructure

Editor

Norbert Schirmer (SRX)

Contributors

Sören Bleikertz, Zoltán A. Nagy, Anil Kurmus, Matthias Schunter (IBM)

Michael Gröne, Norbert Schirmer (SRX)

Allysson Bessani, Bernd Kauer, Paulo Verissimo (FFCUL)

Imad Abbadi (OXFD)

Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia,

Davide Vernizzi (POL)

Johannes Behl, Klaus Stengel (FAU)

Sven Bugiel, Stefan Nürnberger (TUDA)

Disclaimer

This work was partially supported by the European Commission through the FP7-ICT program under project TClouds, number 257243.

The information in this document is provided as is, and no warranty is given or implied that the information is fit for any particular purpose.

The user thereof uses the information at its sole risk and liability. The opinions expressed in this deliverable are those of the authors. They do not necessarily represent the views of all TClouds partners.

Executive Summary

This report describes requirements for a trusted cloud infrastructure, analyses shortcomings of todays cloud offerings and develops new approaches and building blocks for such an infrastructure, focusing on privacy and resilience.

The report starts by collecting the security requirements on Cloud Computing by reviewing surveys on the risks of this new paradigm of computing. It identifies security shortcomings of today's cloud offerings and analyses a state of the art Cloud Computing framework for security features as well as gaps.

Based on these results the core building blocks for a trustworthy cloud architecture are discussed and developed. This covers hardening of existing cloud platforms, new services enhancing the interface, improving resiliency and building a cloud infrastructure based on Trusted Computing technology.

Using Cloud Computing means to share responsibility for IT security with the cloud provider. Trust in the cloud services as well as in the provider has to be established to mitigate the risks of malicious insiders. In this report several aspects and approaches to gain trust are discussed. A secure logging mechanism is described that is the basis for auditing of the cloud services. A middleware layer of infrastructure services is proposed which automatically matches user requirements onto cloud properties. An role based access control model is described that tracks maintenance tasks of cloud administrators to ensure integrity and confidentiality of the customers payload in the virtual machines. A fundamental approach to establish trust in the cloud infrastructure is achieved by employing Trusted Computing technology. This approach provides technical means to measure the integrity of a remote system and to build a verifyable trusted computing base spanning over the customer's own infrastructure as well as the cloud resources of the provider. In this report we describe an architecture for such an trusted infrastructure.

To improve resilience of the cloud infrastructure replication techniques can be used. However, one has to keep the costs induced by replication in mind, as reducing costs is one of the motivations to move into the cloud. In this report an approach for a Byzantine Fault Tolerant architecture is described, which reduces the number of replicas needed.

A core technology enabling Cloud Computing is virtualization. Improving the state of the art in recursive virtualization is discussed focusing on performance aspects as well as in security improvements.

Contents

1 Introduction **1**

I Requirements **4**

2 Surveys on Cloud Computing **5**
2.1 Overview of Surveys on the Risks / Threats of Cloud Computing . . . 5
2.1.1 Gartner 2008 . . . 5
2.1.2 ENISA 2009 . . . 6
2.1.3 CSA 2010 . . . 7
2.1.4 TCG 2010 . . . 7
2.1.5 TUDA 2010 . . . 8

3 Security Threats Towards Commodity Clouds **9**
3.1 Introduction . . . 9
3.2 Related Work . . . 12
3.3 Background on AWS . . . 13
3.3.1 Amazon's Elastic Compute Cloud (EC2) . . . 13
3.3.2 Authentication in AWS . . . 15
3.4 AMI Privacy Analysis . . . 15
3.4.1 Our Findings . . . 15
3.4.2 Tool for AMI Privacy Analysis . . . 17
3.4.3 Costs for AMI Privacy Analysis . . . 19
3.4.4 Discussion . . . 20
3.5 Cloud Specific SSH Threats . . . 21
3.5.1 SSH Authentication . . . 21
3.5.2 AMIs with SSH Backdoor . . . 22
3.5.3 AMI Identification via SSH . . . 23
3.6 Countermeasures . . . 26

4 Security Analysis of OpenStack **29**
4.1 Introduction . . . 29
4.2 OpenStack Architecture . . . 30
4.2.1 Nova: Compute Cloud . . . 30
4.2.2 Swift: Storage Cloud . . . 31
4.2.3 Glance: An Image Repository . . . 33
4.3 Methodology for Security Analysis . . . 34
4.4 Existing Security Mechanisms . . . 34
4.4.1 Nova . . . 34
4.4.2 Swift . . . 36
4.4.3 Glance . . . 37

4.5 Security Shortcomings 37
4.5.1 Nova 37
4.5.2 Swift 40
4.5.3 Glance 40

5 Clouds' Infrastructure Taxonomy, Properties, and Security Challenges 41
5.1 Introduction 41
5.1.1 Cloud Evolution 42
5.1.2 Related Work 42
5.2 Taxonomy of the Cloud 43
5.2.1 Cloud Infrastructure Taxonomy 43
5.2.2 Virtual Control Center 47
5.2.3 Factors Affecting Management Services 47
5.3 Deriving Self-Managed Services 48
5.3.1 Multi-Tier Application Scenario at the Cloud 48
5.3.2 Identifying Management Services 51
5.4 Security Challenges 52
5.4.1 Scenario 52
5.4.2 Security Concerns 53

II Building Blocks for Trusted Cloud Architecture 56

6 Log Service 57
6.1 Background 57
6.1.1 Schneier-Kelsey scheme. 57
6.1.2 Trusted Computing 58
6.1.3 Trusted Computing and log: related works 59
6.2 Log Service Model 59
6.2.1 Terminology, Actors and Main Components 59
6.2.2 Overview 60
6.2.3 Functionality 60
6.2.4 Functional and Security Requirements 63
6.3 Architecture 64
6.3.1 Applying SK scheme to the cloud environment. 64
6.3.2 Logging process phases. 65
6.3.3 Relationship with Other Components of the Cloud. 66
6.4 Enhancing trust in the logging process with TC 66
6.5 Possible future research directions. 67

7 Secure Cloud Maintenance - Protecting workloads against insider attacks 70
7.1 Introduction 70
7.1.1 Protecting Cloud Users during Server Maintenance 71
7.1.2 Outline 72
7.2 Requirements and Threat Model 72
7.2.1 Maintainability Requirements of Cloud Administrators 72
7.2.2 Security Objectives of Cloud Users 73
7.2.3 Threat & Trust Model 73

7.3 Extending an Infrastructure Cloud Architecture . . . 73
7.3.1 Current Architectures of Infrastructure Clouds . . . 74
7.3.2 Architecture Extensions for Minimizing Administrator Privileges . . . 74
7.4 Minimizing Privileges of Administrators during Maintenance . . . 76
7.4.1 Initial Trusted State . . . 76
7.4.2 Privilege Levels . . . 76
7.4.3 Policy Enforcement for Privilege Levels . . . 77
7.4.4 Privilege Elevation . . . 78
7.4.5 Privilege Revocation and Recovery . . . 80
7.5 Security Discussion . . . 82
7.5.1 Integrity . . . 82
7.5.2 Confidentiality . . . 83
7.5.3 Availability . . . 83
7.6 Implementation & Evaluation . . . 84
7.7 Related Work . . . 84
7.8 Conclusions and Future Work . . . 85

8 Self Managed Services **86**
8.1 Introduction . . . 86
8.2 Self-Managed Services at Virtual Layer . . . 87
8.2.1 Factors Affecting Management Services . . . 87
8.2.2 Functions of Self-Managed Services . . . 88
8.2.3 Services Interdependency . . . 93
8.3 Main Challenges and Requirements . . . 93
8.4 Conclusion . . . 95

9 Resource Efficient BFT **96**
9.1 Introduction . . . 96
9.2 Background . . . 97
9.2.1 Trusted Subsystems . . . 97
9.2.2 Preventing Equivocation . . . 98
9.2.3 Trusted Signed Counter . . . 98
9.3 System Model and Architecture . . . 99
9.4 CheapBFT Protocol Suite . . . 100
9.4.1 Clients . . . 100
9.4.2 Active and Passive Replicas . . . 101
9.4.3 When to Abort? . . . 101
9.5 PrimaryBackup . . . 101
9.5.1 Agreement Stage . . . 102
9.5.2 Execution Stage . . . 103
9.5.3 Snapshots and Garbage Collection . . . 103
9.5.4 Switching Protocols . . . 104
9.5.5 Validating an Abort History . . . 104
9.5.6 Processing an Abort History . . . 105
9.6 BackupRotation . . . 106
9.7 PlainBFT . . . 107
9.8 Related Work . . . 108
9.9 Conclusion . . . 108

10 Tailored Cloud Services 109
10.1 Introduction . . . 109
10.2 General concepts . . . 109
10.3 Operating System design . . . 110
10.3.1 Virtual hardware support . . . 110
10.3.2 Tailoring process . . . 111
10.4 Choice of Programming Languages . . . 111
10.4.1 Formal verification . . . 111
10.4.2 Language candidates . . . 112
10.4.3 Java . . . 113
10.5 Summary . . . 115

11 Trusted Platform Agent 116
11.1 Introduction . . . 116
11.2 Software Development with TC . . . 117
11.3 Trusted Platform Agent . . . 117
11.4 Features of the TPA . . . 119
11.4.1 Foundation of Protection . . . 119
11.4.2 Local Protection . . . 119
11.4.3 Vouching for Protection . . . 120
11.5 Concluding Remarks . . . 120

12 Trusted Infrastructures 122
12.1 Introduction . . . 122
12.2 Infrastructure Overview . . . 123
12.3 Trust Model . . . 123
12.4 Central Management . . . 125
12.4.1 Functionality . . . 125
12.4.2 Management . . . 126
12.5 Trusted Virtual Domains . . . 128
12.5.1 TVD Features Overview . . . 128
12.5.2 TVD Architecture Overview . . . 129
12.5.3 Functionality and Management . . . 131
12.6 Inter-TVD Information Flow Control . . . 133
12.6.1 State of the Art . . . 133
12.6.2 Requirement Analysis . . . 134
12.7 Conclusion . . . 137

13 Recursive Virtual Machines 138
13.1 Motivation . . . 138
13.2 Related Work . . . 139
13.3 Nested Virtualization . . . 140
13.4 A Novel Design . . . 142
13.4.1 CPU Virtualization . . . 142
13.4.2 I/O Virtualization . . . 144
13.4.3 Discussion . . . 146
13.5 Advanced Security Mechanisms . . . 146
13.6 Conclusions . . . 147

Bibliography 147

List of Acronyms 163

Glossary 165

List of Figures

3.1 Basic System Model of Cloud App Store 10
3.2 VM instantiation in Amazon AWS. The Consumer chooses the image (AMI-ID), resources (Type), and availability zone (Region) for her VM on the Web Interface of the AWS Cloud App Store. Depending on the type of the AMI, the VM is instantiated (Instance-ID$_{\text{AMI-ID}}$) either as (A) EBS-backed or (B) S3-backed. 14
3.3 Approach for Analysis of AMIs 18
3.4 Linux Based Public AMIs per Region 20
3.5 SSH First Connection 22
3.6 Matching Between AMIs and Instances 25
3.7 Improved Bundling Tool 27

4.1 OpenStack Nova Architecture (Source: `http://nova.openstack.org/service.architecture.html`). 30
4.2 OpenStack Swift Architecture. 32
4.3 The Glance Architecture. 34
4.4 The User authenticates itself to the *Auth Component*. On success, the actual OpenStack service can be reached, which – if necessary – asks the *Auth Component* to check permissions. 37

5.1 Cloud Taxonomy: 3-D View 43
5.2 Conceptual Models of Cloud Layers 46
5.3 Cloud Taxonomy — Multi-Tier architecture at Community Cloud Provider . . 49

6.1 Schneier and Kelsey's log entry creation scheme. 58
6.2 Use case diagram for cloud ontology. 61
6.3 Use case diagram for log service. Compare it to *each layer* in Figure 6.2 62
6.4 High level architecture of the Log Service. 65
6.5 Schneier and Kelsey's log entry scheme extended with `TPM_Quote`. 68

7.1 State Diagram for a Compute Node per Customer 71
7.2 74
7.3 Extended Architecture Overview 75
7.4 Overview of Privilege Elevation Process 79
7.5 Overview of Revocation and Recovery Process 80

8.1 Factors Affecting Self-Managed Services Behavior 87
8.2 Adaptability Function 89
8.3 System Architect and Resilience Functions 90
8.4 Scalability Function 91
8.5 Availability Function 92
8.6 Service Reliability Function 92

8.7 Self-Managed Services Interdependency 93

9.1 Implementation of a trusted signed counter 99
9.2 A minimal configuration of CheapBFT consisting of two active and one passive replica each equipped with a trusted subsystem (TSS). 100
9.3 PrimaryBackup (for $f = 1$) message exchange between two active and one passive replica 102
9.4 Protocol PrimaryBackup for active nodes 102
9.5 Protocol PrimaryBackup passive nodes 103
9.6 Abort history log 105
9.7 Respond to a PANIC message 105
9.8 Checking an abort history 105
9.9 Protocol BackupRotation balances load over all nodes by means of multiple configurations of PrimaryBackup 107

12.1 Schematic Trusted Infrastructure - TrustedServers managed by the TrustedObjects Manager. 124
12.2 Overview of TVD architecture with logical view of three TVDs distributed over two physical platforms with four compartments and the physical deployment of the TVD components. 130
12.3 Example of an inter-TVD information flow between two TVDs on different platforms and supporting security services such as TVD Manager and Information Flow Manager. 134
12.4 Overview of a TVD-based trusted infrastructure use-case. 137

13.1 Recursive virtualization with three hypervisors. The root HV multiplexes the hardware, whereas the other hypervisors wrap the OS to implement security functions. 139
13.2 Virtualization instructions usually executed by a hypervisor to handle a single trap on AMD and Intel CPUs. A nested hypervisor will need six virtualization instructions to emulate one virtualization instruction from its child. 141
13.3 Maximum number of nested VMs for different branching factors that can sustain a given interrupt frequency. Calculated for a 3 GHz machine where a single trap can be handled within 1000 cycles. 142
13.4 Event-flow within three levels of VMs. With nested virtualization (left) an event causes an exponential number of traps to be forwarded through the hierarchy. In our design either the root hypervisor can handle the event itself (middle) or it forwards it directly to the upper layer (right). 142
13.5 Bridging cascade (left) vs. direct assignment of virtual NICs (right). Traditionally every virtualization layer implements a bridge to multiplex its network card with his child VMs. This forces packets through a deep bridging cascade. In our design the root layer can emulate multiple virtual NICs and recursively assign them to upper layer VMs. Thus only a single bridge is involved in network packet handling. 144

List of Tables

3.1 Summary of the Findings of our Privacy Analysis of Public EBS-backed AMIs (April 2011) 16
3.2 Example of Search Patterns ($*$ denotes the wildcard character and | alternatives) 19
3.3 Attacks based on identical SSH host keys. 23

7.1 Overview of Privilege Levels 76

11.1 Libraries related to TC. The table does not list TPM/J and TPM4JAVA since they are no longer developed. 118

Chapter 1

Introduction

Cloud Computing promises on-demand provisioning of scalable IT resources, delivered via standard interfaces over the Internet. Only the capacity used is charged, and the resources scale with the demands of the users. Fixed investments into an on-premise IT is turned into variable costs, a static IT-infrastructure becomes a flexible and dynamic service. These characteristics fit perfectly into todays fast paced business world, however there are problems that inhibit coarse acceptance of Cloud Computing in todays businesses. Putting resources into the Cloud also results into a shared responsibility between cloud provider and customer. In particular the responsibility for all security aspects is now shared. Moreover as all cloud customers use the same resources, the infrastructure is shared among multiple clients, usually called tenants in this context, which may be competitors.

The goal of the TClouds project is to tackle the security challenges of Cloud Computing, and build a trusted and resilient cloud infrastructure.

In TClouds we focus on the most flexible service model of Cloud Computing, Infrastructure-as-a-Service (IaaS). Customers can build entire virtual infrastructures by renting resources like storage, network, and computing platforms in form of virtual machines with administrative access to the whole operating system.

This report describes the work of the first year within Work Package 2.1, which is about building a single trusted cloud. Whereas report D.2.2.1 describes the work on the cloud-of-clouds middleware of Work Package 2.2 and Work Package 2.3 and its report D2.3.1 is concerned with the management aspects of both the single cloud as well as the cloud-of-clouds scenario. The interconnection of the topics is reflected in the different reports as they describe different aspects of our overall approach towards a trusted Cloud Computing platform. For example the work on ensuring integrity and confidentiality of virtual machine images is described in D2.3.1 as it turned out that the major challenge is the key management aspect. Nevertheless, the resulting component is an important building block of a trusted cloud infrastructure as we describe it in this report.

Structure

The report is structured into two main parts. The first part focuses on the requirements of a privacy enhanced and resilient trusted cloud, analyses cloud infrastructures in general and identifies security deficiencies. The second part describes the core building blocks of a trusted cloud infrastructure which are needed to overcome the deficiencies identified in the first part.

Chapter 2 summarizes important Surveys on Cloud Computing focusing on the risks and security challenges. The surveys are complemented by an own survey focusing on the research challenges. The main risks and security challenges concentrate around the core features of cloud offerings. First using Cloud Computing means outsourcing. Resources and responsibility

are split between the customer and the cloud provider. Trust between the cloud provider and the customer has to be established and compliance to legal requirements of a customer has to be fulfilled by the provider. Second, resources at the cloud provider are shared between customers which might be competitors. So isolation of customers becomes a main requirement for a trusted cloud infrastructure. Third, a public cloud offers standardized interfaces to literally anybody. So protection of these public api's against attacks is of utmost importance.

Chapter 3 analyses the security threats of today's commodity clouds, focusing on the market leader Amazon Web Services. The chapter investigates the security and privacy threats caused by the unawareness of cloud users with respect to the image management in the cloud. Virtual machine images are published (e.g. containing ready to use web services or database systems) and can be used by other customers to rapidly deploy functionality in the cloud. Severe shortcomings were found in the images, like backdoors, private keys, passwords and sensitive data. The chapter also covers the techniques used to analyse the images and discusses possible countermeasures.

Chapter 4 is dedicated to a security analysis of OpenStack. OpenStack is an open source Cloud Computing framework and serves two purposes within the TClouds project. First, as a representative example of a Cloud Computing framework, it is used to identify the gaps which have to be closed in order to obtain a trusted cloud infrastructure. Second, as it is an open source framework we use it as a platform to prototype the TClouds security and resilience extensions within a realistic setup.

Chapter 5 suggests a taxonomie of infrastructure clouds that is used to identify and describe important requirements for Cloud Computing. This covers both functional requirements and privacy and resilience requirements. This chapter lies the foundation for Chapter 8 were the taxonomie is used to design a cloud architecture.

In Chapter 6 a logging mechanism for the trusted cloud infrastructure is discussed. The focus is the secure logging of infrastructure events such as creation, destruction or migration of a VM or allocation and deletion of a bucket of storage. A trusted logging facility is a core requirement for a cloud infrastructure to reach compliance with the legal requirements on auditability of the cloud infrastructure. Trusted Computing Technology is used to implement a hardware anchor to ensure code integrity of the logging facility.

Chapter 7 builds on the analysis of OpenStack of Chapter 4 and proposes a solution to limit the threat of malicious cloud administrators on the confidentiality and integrity of the customers' virtual macchines. A role based access control model is suggested that controls the maintenance tasks of a remote administrator. During normal operation the cloud middleware itself has full control and there is no need for an administrator to access the servers. Only during maintenance operations the access rights of the administrator are elevated for the dedicated task by fine grained privilege levels. The system always keeps track on the possible effects on confidentiality and integrity.

Chapter 8 continues the requirements analysis of Chapter5 by presenting a high level design of a cloud middleware of self-managed services. User requirements are matched with the infrastructure properties by the services which take care of compliance issues (territorial restrictions of the serves), security obligations (e.g. isolotaion, confidentiality) as well as resilience requirements.

Chapter 9 introduces a resource efficient Byzantine Fault Tolerance (BFT) framework for the cloud infrastructure. Availability and resilience of the cloud infrastructure is a core requirement for a trusted cloud. A Byzantine Fault Tolerant architecture is not only able to handle crash-stop failures by introducing replication, but can handle arbitrary faults, ranging from software bugs, intrusions, viruses to spurious hardware errors. However, the drawback of a Byzantine

Fault Tolerance architecture is the high degree of replication that is needed, which increases the costs of Cloud Computing. In this chapter a variant of BFT is discussed which is more resource efficient by relying on a trusted counter.

Chapter 10 describes a mechanism to automatically taylor virtual machines to act as an auxiliary service within the cloud. With this approach cloud customers can deploy resource efficient, minimized services within the cloud for which they have assessed trustworthiness. As an example a key/value storage service is used.

Chapter 11 introduces a Trusted Platform Agent, which supplies a neat interface to Trusted Computing technology by integrating them into a common library together with other commonly needed functions (e.g. cryptographic or network-related functions). Trusted Computing technology is employed as a basic technology in Chapters 6 and 12.

Chapter 12 describes how to build a trusted cloud infrastructure from the ground up based on Trusted Computing technology providing a hardware anchor as a root of trust. Starting from a trusted boot procedure a security kernel controls the hypervisor and all server machines to ensure confidentiality and integrity of the platform. A central management component is used to control security policies which are deployed and enforced on all servers. The concept of Trusted Virtual Domains is supported for seamless integration of the cloud offering into the infrastructure of a customer without sacrificing security requirements.

Hardware virtualization is the core technology to enable Cloud Computing by allowing to share hardware resources by offering virtual machines to the customer. In Chapter 13 the problem of recursive virtual machines is discussed and a novel solution is proposed. The need for recursive virtual machines has two sources. First, customers may use virtualisation within their infrastructure and want to provision this setup into the cloud, which also uses virtualisation. Moreover, recursive virtual machines can be used to add additional layers of security. In this chapter the efficiency problems are addressed that arise when naively stacking virtual machines.

Part I

Requirements

Chapter 2

Surveys on Cloud Computing

Chapter Authors:
Sven Bugiel, Stefan Nürnberger (TUDA), Norbert Schirmer (SRX)

2.1 Overview of Surveys on the Risks / Threats of Cloud Computing

In this section we give an overview on surveys that evaluate the risks of Cloud Computing. By using these risks as a starting basis we evaluate which risks we can counter by technical means within a trusted cloud infrastructure.

2.1.1 Gartner 2008

This is a summary of Gartner's assessment of the security risks of Cloud Computing [HN08].

Privileged User Access As sensitive data is processed outside of the enterprise, by non-employees, special care has to be taken to control who has access to the data, and how to ensure the trustworthiness of these people.

Compliance The user himself is ultimately responsible for the security and integrity of the data stored in the cloud. A service provider should submit to external audits and security certifications and provide the customer with the necessary information as a basis for a trust relationship between the customer and the service provider.

Data Location When storing data in the cloud it is unclear where it is physically stored, it might even be stored in different countries with different national privacy regulations. If one needs to stay within a specific jurisdiction the provider has to give contractual commitment to obey the law.

Data Segregation Data in the cloud is stored in a shared environment together with the data of other customers. To segregate the stored data encryption may be offered by the providers. How can trust into the key-management and encryption implementations used by the provider be assessed by the customer?

Availability While the scalability and reliability of today's cloud services seems to be high, there is no guaranteed quality of service with a contractual commitment by the provider and which can be technically enforced.

Recovery How can the cloud offering recover from total disaster? Even if the provider isn't willing to tell where he stores the data he should tell the customers about backup and recovery strategies.

Investigative support Cloud services are especially difficult to investigate, in case of illegal activities because logging and data of multiple customers are colocated on the same physical machine and may spread dynamically over different hosts and data centers. Moreover, the provider should give contractual commitment to allow such investigations at all.

Long-term viability Will the customer be able to get all his data back in case the provider gets broke?

2.1.2 ENISA 2009

This is a summary of ENISA's assesment on the risks of Cloud Computing [ENI08].

Loss of Governance The customer cedes control to the cloud provider on security issues, which may not be covered by a commitment in the service level agreement.

Lock-In It is hard to migrate from one provider to another or back to an in-house solution, as tools and standards for services are rare. This can introduce a dependency on a particular cloud provider.

Isolation Failure Multi-tenancy and shared resources pose new risks for separating storage, memory, routing etc. However attacks against hypervisors are still rare compared to attacks against traditional OSs.

Compliance Risks Achieving certain levels of certifications may be hard or impossible for a customer: The cloud provider may have to give evidence of his own compliance, the cloud provider may not permit audit actions by the customer, or compliance cannot be achieved in a public cloud infrastructure at all (e.g. PCI DSS).

Management Interface Compromise Customer management interfaces are publicly accessible and control a large set of resources and hence pose an increased risk by being compromised, especially via remote access and web browser vulnerabilities.

Data Protection It is hard for a customer to effectively check how data is handled by the provider and to ensure that data is handled in a lawful way. The situation gets even worse between federated clouds.

Insecure or Incomplete Data Deletion Data deletion may not result in true wiping of the data. Redundant copies may not be deleted, or hard disks cannot be destroyed as they also store data of other tenants.

Malicious Insider The damage that may be caused by a malicious insider is great, especially by system administrators or security service providers.

2.1.3 CSA 2010

This is a summary of the top threats of Cloud Computing identified by the cloud security alliance [All10].

Abuse and Nefarious Use of Cloud Computing Cloud resources and the anonymity in the cloud may attract criminal activity within the cloud. A cloud provider may be attacked by his own users with his own infrastructure.

Insecure Interfaces and APIs Clouds offer rich interfaces and APIs for cloud management and control which can be attacked and should be secured.

Malicious Insiders Due to convergence of IT services and customers on a single management domain the threat of malicious insiders is amplified. Moreover the processes and procedures within the providers may not be transparent to the customers.

Shared Technology Issues Strong isolation of different compartments (of multiple tenants) on the shared hardware resources have to be maintained by the hypervisor, which can be target of attacks.

Data Loss or Leakage The threat of data loss or leakage is increased in the cloud, due to the number of risks and challenges which are either unique to the cloud, or more dangerous because of architectural or operational characteristics of the cloud.

Account or Service Hijacking In a cloud scenario attacks like phishing, fraud and exploitation of software vulnerabilities are even more dangerous as an attacker can gain access or manipulate all the operations that are performed within the cloud.

Unknown Risk Profile By outsourcing the hardware/software ownership to the cloud, the customer looses the control/information about the current state of security relevant issues: Software versions, code updates, intrusion attempts, etc.

2.1.4 TCG 2010

This document [Gro] is partly based on the CSA report. They propose six challenges for Cloud Computing:

Securing Data At Rest The data stored at the cloud provider should be encrypted to preserve privacy and confidentiality

Securing Data in Transit Data in transit should be protected by encryption to prevent eavesdropping. Furthermore, authentication is necessary to be certain about the two (or more) communication partners.

Authentication In the cloud environment, authentication and access control are more important than before, since the data would otherwise be accessible to *everybody*. Additionally, the SSO or similar means should support to e.g. hinder fired employees in accessing data.

Separation between Costumers is important, as physical proximity (e.g. same machine) allows for covert channels, for which no absolute countermeasure exists.

Cloud Legal and Regulatory Issues must be verifiable by the client, who uses the cloud service to guarantee that they are compliant with his/her expectations.

Incident Response A plan must exist for security breaches or misbehaviour. At least an automated notification (or even automated response) is needed.

2.1.5 TUDA 2010

The TClouds partners of the TU-Darmstadt started their own survey to summarize the challenges of security in Cloud Computing. Many are solved today, others have questionable solutions and yet some of them are unfortunately left open for further research or even not solvable.

Economical View. From an economical point of view, security features are usually regarded as utility-function. That means, that the costs for implementing security features are only paid, if the risk of unsatisfied costumers or data theft is higher than a certain threshold. This threshold is usually the money it would cost to handle such an incident multiplied by its probability ($M \cdot p$). If the cost for implementing security counter measures are higher than this product ($C > M \cdot p$) the countermeasures are usually *not* taken.

Storage. Encryption is not the holy grail for storage. Also necessary are means to proof data possession (so called Proof of Retrievability (POR) or Provable Data Posession (PDP)). Another great demand for encrypted data is the fact, that it should still be indexable and sortable. This can be done using Order Preserving Encryption (OPE).

Computation. Outsourcing computation to the cloud means to trust the cloud, that these computations were indeed performed. Several means exist to verify execution. Some approaches silently compute an undetectable hash during execution (that can later be verified), others operate on a different computation model, e.g. Fully Homomorphic Encryption (FHE) or Garbled Circuits (GC).

Policies may further enhance the security and trust of the cloud's client(s). They serve several functions. Some of them are: Separation of competitors, local law regulations etc. The negotiation of policies has sometimes to be done in private, as neither party would like to reveal their extent, to which they would be willing to 'bend' their demands. Automatic, private policy negotiation does this job and provably gathers the best result for both parties (unbiased).

Attack Scenarios and their Mitigation. Detection and avoidance of Denial Of Service (DoS) attacks.

Covert Channels. Covert Channels are not easy to handle, as basically *every* part involved in the design of a computer or software could be used to communicate covertly with another party. This poses an attack scenario for competitors, as [RTSS09] have shown that it is possible to guess the proximity of two virtual machines in the Amazon EC2 cloud.

Chapter 3

Security Threats Towards Commodity Clouds

Chapter Authors:
Sven Bugiel, Stefan Nürnberger (TUDA)

3.1 Introduction

Cloud computing offers fine-grained IT resources, including storage, networking, and computing platforms, on an on-demand and pay-per-use basis. The high usability of today's Cloud Computing platforms makes this rapidly emerging paradigm very attractive for customers who want to instantly and easily provide web-services that are highly available and scalable to the current demands [NIS11b].

In the most flexible service model of Cloud Computing, *Infrastructure-as-a-Service* (IaaS), customers can build entire *virtual infrastructures* by renting resources like storage, network, and computing platforms in form of virtual machines with administrative access to the whole operating system. other users, similar to an app store for the cloud. An important feature of modern commodity IaaS cloud providers is *VM image management*. Cloud customers are able to import, create, share, and publish the images (or *snapshots*) of their Virtual Machines. For example, the open source project OpenStack recently started the subproject *Glance* for *"discovering, registering, and retrieving virtual machines images."*. The well-known provider Rackspace offers its customers the possibility to *"create an image of any cloud server"*. However, the by far most flexible and powerful image management is provided by the IaaS market-leader Amazon Web Services (AWS). Amazon offers not only pre-configured, templated images which enable the AWS customers to get their virtual infrastructure up and running immediately, but also provides very powerful and easy to use tools (e.g., a web-service) to import existing images (e.g., from VMWare based infrastructures), create new images from existing ones or running VMs, and to publish the images to make them available to other AWS customers. At the time of writing, there are more than 15.000 public images available[1].

In its essence this usage model resembles the one of the very-well established mobile app stores like Apple's App Store or the Android Market. Cloud customers are able to rapidly deploy new functionality in their virtual infrastructures by using publicly available images that implement the required functionality. Although the bulk of the contemporary available VM images are for free, a new market is opened by providing images in return for payment. For

[1]A very nice overview over the public images on AWS is provided by the website http://www.thecloudmarket.com/

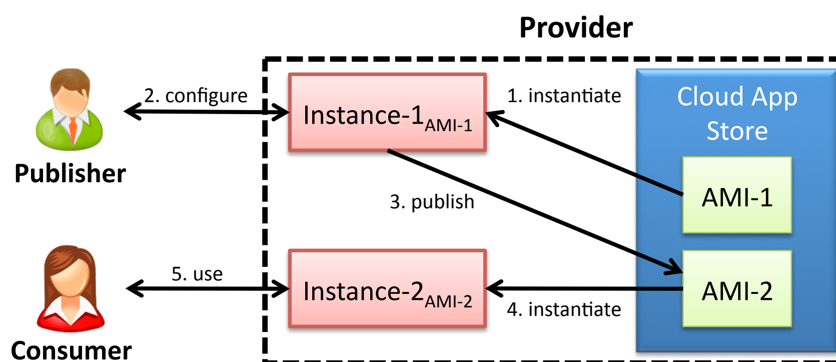

Figure 3.1: Basic System Model of Cloud App Store

instance, on Amazon images can be published as so called *paid AMI* for which users have to pay an extra fee to the creator of the image in return for using the VM. Even third parties like VMWare recognized the possibilities of VM images and since recently provide a *virtual appliance store*[2] for VMWare technology based clouds.

In a recently published paper [BPN+11], we investigated and evaluated the security and privacy threats caused by the unawareness of users in the cloud with respect to the above mentioned image management. Although the methods and techniques described in the following are applicable to arbitrary IaaS providers, we focus on one of the major cloud providers, Amazon's Elastic Compute Cloud (EC2) [EC2] and adapt our terminology accordingly. In the following, we describe the players involved in the (Amazon) *Cloud App Store* and the resulting security challenges.

The Cloud App Store. As shown in Figure 3.1, the Cloud App Store involves a Provider and typically two kinds of users, Publisher and Consumer. The *Provider*, Amazon in our case, operates the IaaS cloud infrastructure, authenticates users and bills them for the resources they consumed.

The *Publisher* creates and publicly offers cloud apps, called *Amazon Machine Images* (AMIs). For this, he selects an existing AMI (AMI-1 in Figure 3.1), instantiates it (Instance-$1_{\text{AMI-1}}$), logs into the running instance to configure it, and finally publishes a snapshot as a new AMI (AMI-2).

The *Consumer* selects this AMI from a list of available AMIs, instantiates it (Instance-$2_{\text{AMI-2}}$), and uses it for her purposes. Optionally, a Publisher can declare an AMI as *paid AMI* to earn money from Consumers invoking it.

Security Challenges. In this deliverable, we focos on specific security challenges for both, Consumers and Publishers that arise when sharing images using the model described above (see also [WZA+09, ASM09]).

Security of Consumer. The Consumer must trust the Publisher not to include any malware into the AMI. Such a malicious AMI could contain a Trojan horse that spies on or modifies the Consumer's data, or a backdoor for malicious remote login. Even though full protection against such malicious AMIs is almost impossible, filters, virus scanners, and rootkit detectors could provide at least some level of protection [WZA+09].[3]

[2] http://www.vmware.com/appliances/

[3] In principle, this is similar to mobile app stores where downloaded apps must be trusted as well. Recently, Google's mobile app store withdrew 25 Android apps that were infected with malware [Goo11]. As such attacks

Security of Publisher. The Publisher on the other hand might accidentally publish AMIs that contain highly sensitive information. Examples include keys, credentials, passwords, command history/log files, or source code.

Although Amazon's user guide recommends to ensure that all confidential information is removed before publishing an AMI [Ama11a, Sharing AMIs Safely], many users seem to be unaware of the crucial consequences of ignoring these recommendations, do not have the appropriate tools at hand, or simply forgot private data in their AMIs.

The Gap between Theory and Practice. The Provider could filter AMIs for Trojans, backdoors, or confidential information to reduce the chance of malicious or sensitive data within AMIs. This was proposed in [WZA+09], but although the automated filtering system presented in that paper seems to be used already within the IBM SmartCloud [IBM], the explicit filtering rules are not available to the public.

In contrast, Amazon currently does not provide automated scanning of public AMIs as they are not responsible/liable for what users do with their own data. Though Amazon quickly reacts on incidents reported to their security hotline and informs affected customers, e.g., those running an AMI in which a backdoor was found [Ama11c].[4]

In this chapter we show that these previously reported incidents are only the tip of the iceberg and many of the publicly available AMIs have severe security vulnerabilities leaking highly sensitive data.

Our Contribution and Outline After summarizing related work in Section 3.2 and giving background information on the Amazon Web Services (AWS) in Section 3.3 we present the following contributions.

Extraction of Sensitive Information from Public AMIs (cf. Section 3.4). Through an extensive analysis we were able to extract highly sensitive information from several publicly available EC2 AMIs. To make the analysis cost and time effective we developed an automated tool that uses different search strategies and exploits technology specific aspects of the Amazon cloud. The costs for running our attack were less than $\$20$ while the information we extracted from the AMIs would allow an attacker to cause financial damage of several $\$10,000$ per day and could severely harm the reputation of several companies that operate services in the cloud. After testing overall 1225 AMIs we got hold of the source code repositories, administrator passwords and other types of credentials of various web service providers.

SSH Vulnerabilities in AMIs (cf. Section 3.5). We discovered several vulnerabilities in AMIs that are introduced by incorrect usage and configuration of SSH. About one third of the tested 1100 public AMIs in Europe and the US-East region contain an SSH backdoor, i.e., a (forgotten) public key that allows remote login for the Publisher. We identified multiple instances that use the same SSH host key which allows an external attacker to correlate these instances running the same or a similar AMI, identify candidates for corresponding public AMIs, and mount several attacks, e.g., host impersonation.

Countermeasures (cf. Section 3.6). We provide several mechanisms to protect against our attacks on public AMIs. Besides organizational measures we propose to use our tools to enhance the security of the interfaces for publishing AMIs and also extensions to the interface of the Cloud App Store.

also harm the reputation of the mobile app store provider, some providers already review new apps submitted to the store to ensure that they perform as expected [App, Hea06].

[4] "For security reasons, we (Amazon) recommend that any instance based on a publicly available AMI that is distributed with an included SSH public key should be considered compromised and immediately terminated." [Ama11b]

3.2 Related Work

In this section we briefly revisit previous work on the security challenges of publicly sharing Virtual Machine (VM) images (AMIs in our terminology) on which we build our practical attacks. Afterwards we review the main related work on general cloud security, security aspects specific to the Amazon cloud, and methods for searching private data.

VM Image Analysis As summarized in Section 3.1, security and privacy risks for the Consumer and Publisher when sharing VM images have been identified in [WZA+09]. Shared VM images may contain either malware that was intentionally or unintentionally included by the Publisher. To protect against these threats, the authors propose filtering of VM images by the Provider which has been implemented in the Mirage image management system [RTA+08]. The IBM SmartCloud [IBM] deploys a system that is presumably based on (a mechanism like) Mirage for automated patching [ZNZ+10] and periodical malware scans [SRA+10] of public VM images.

We show that the risks pointed out in [WZA+09] ubiquitously occur in Amazon's Cloud App Store and have more severe consequences than previously thought. As our results show, Amazon does not apply any centralized filtering as proposed in [WZA+09], and considers this as the responsibility of cloud users. As countermeasure against these threats we propose tools and extensions to the user interface of the Cloud App Store that allow even technically less skilled users to protect their published AMIs from containing sensitive data.

General Cloud Security Challenges The risks and threats for Cloud Computing as analyzed in [Clo10b, Eur09, CPK10, TJA10] concern mainly the vulnerabilities inherent to cloud infrastructures, e.g., protection of the outsourced data and computations against eavesdropping and illegitimate modifications; new threats induced by sharing physical resources with other users' VMs (*multi-tenancy*); non-availability of the users' outsourced data and cloud-based services; or vendor lock-in to a single provider.

The security threats induced specifically by the usage model and capabilities of VMs, e.g., massive scalability and VM snapshots, i.e., copies of the current state of a VM, were discussed in [GR05a, RY10, CYC08b]. These threats include, e.g., the rollback of a VM to an already compromised state, the conflict between traditional security management systems and the VMs' flexibility, or the loss of entropy upon state rollback affecting the freshness needed for cryptographic mechanisms and protocols.

Recent guidances and best practices aim at mitigating these threats and securing Cloud Computing [NIS11a, Clo09b]. However, we show that many Publishers and Consumers do not adhere to these recommendations.

Amazon Specific Cloud Security Challenges The documentation of the Amazon Web Services (AWS) [AWSb] contains several relevant security guidelines. Moreover, the Elastic Compute Cloud (EC2) [EC2], an integral part of AWS, has recently been subject to scientific and industrial security research as summarized next.

AWS Security Recommendations. An overview of the security processes in the AWS cloud infrastructure is given in [Var11]. Further, Amazon provides security guidelines and best practices on how to use AWS [AWS10], including advises and references [Swi09] on how customers can secure their AWS credentials in EC2 instances with respect to the risk of unintentionally embedding them in public AMIs. However, the proposed solutions require technical

knowledge of the AWS mechanisms which conflicts with the high usability of the Cloud App Store. Thus, Cloud App Store users are either unaware of the security risks/guidelines or are only capable of using the store but not of implementing the security guidelines. This assumption is underlined by the high number and severity of our findings.

VM correlation. The possibility to gain insight into the EC2 network topology, and how to exploit this knowledge for placing a malicious VM A purposefully on the same physical host as a specific foreign VM B, and finally to eavesdrop on VM B via side-channels, was demonstrated in [RTSS09]. Our SSH correlation attack (cf. Section 3.5) provides additional information about running VMs that can be used to enhance their approach by narrowing down the search space for the co-location.

Malicious VMs. The attacks presented in [ASM09] exploit design flaws of the EC2 Cloud App Store such as phishing by publishing a malicious AMI that appears high in the list of available AMIs. Our countermeasures proposed in Section 3.6 can be used to mitigate these attacks.

Private Data Search Several methods and sources exist to search for unintentionally leaked private data in public resources. Examples include recovery of sensitive data from second-hand hard-drives [GS03], or "Google hacking" [AT07, Tat06] which allows to query the Google search engine to find private information in its indices, e.g., private keys, hashed passwords, or private information about a person.

3.3 Background on AWS

In this section we recall the main aspects of the Amazon Web Services (AWS) [AWSb]: the Amazon Elastic Compute Cloud (EC2) [EC2] in Section 3.3.1 and authentication to AWS in Section 3.3.2. For detailed information we refer to Amazon's documentation on AWS [AWSc]. Readers already familiar with AWS can skip this section.

3.3.1 Amazon's Elastic Compute Cloud (EC2)

In 2006, Amazon introduced the Elastic Compute Cloud (EC2) [EC2] as part of the Amazon Web Services (AWS). It allows users to run Virtual Machines (VMs) on-demand on the infrastructure provided by AWS. The VMs behave similar to a physical server and contain a full-blown operating system (currently supported are various Linux distributions, FreeBSD, OpenSolaris, and Windows). Running VMs are called *instances* and isolation between instances is enforced by the XEN hypervisor [AWS10]. Figure 3.2 illustrates the instantiation of VMs from Amazon Machine Images (AMIs) via the AWS Cloud App Store. When a Consumer starts a VM, she has to specify 1) an AMI from which the instance is derived (AMI-ID), 2) a type defining the resources available to the instance (Type), and 3) the geographical region in which the VM is deployed (Region) as described next.

Region and Type The Region defines one out of five EC2 data centers (two in the US, one in Europe, and two in Asia). To guarantee failsafe operation, each data center of a region is currently split into two or more independent *availability zones*.

The Type specifies how many resources are allocated for the instance (CPU, RAM, temporary disk space, speed of the network connection) and determines the costs ranging from

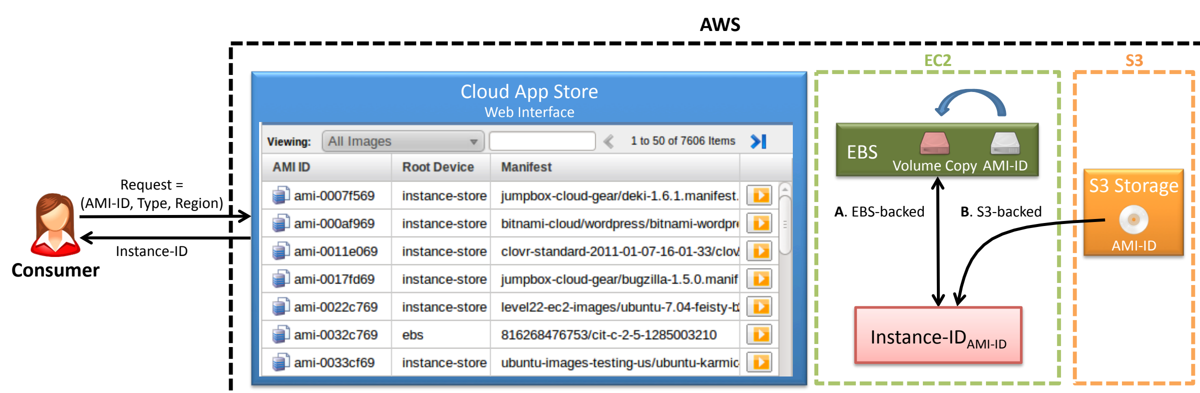

Figure 3.2: VM instantiation in Amazon AWS. The Consumer chooses the image (AMI-ID), resources (Type), and availability zone (Region) for her VM on the Web Interface of the AWS Cloud App Store. Depending on the type of the AMI, the VM is instantiated (Instance-ID$_{AMI-ID}$) either as (A) EBS-backed or (B) S3-backed.

$0.02 to $2.10 per instance operation hour. Besides costs for running instances, the user is also monthly charged for I/O usage and consumed network bandwidth.

Amazon Machine Images (AMIs) Amazon introduced the concept of pre-built VM image templates, called Amazon Machine Images (AMIs) for rapid deployment of instances. An AMI contains a whole operating system together with applications and data. When an instance is started, a copy of the selected AMI is booted and control over the instance is handed over to the user. AMIs are managed in the *Cloud App Store* and are either provided directly by Amazon or by third party publishers. Users can take these public AMIs to create their own AMIs which are either kept for themselves (private AMIs), made accessible to a group of users (shared AMIs), or made publicly available for every user of EC2 (public AMIs) as shown in Figure 3.1. AMIs are further distinguished by the storage type they are based on – either S3 or EBS – as described next.

S3-backed AMIs. S3-backed AMIs are stored on the highly available Simple Storage Service (S3) [S3]. As shown in Figure 3.2, S3-backed AMIs are instantiated by first copying the image onto the hard drive of a physical EC2 node which then boots the image. A new S3-backed AMI can be created from within a running S3-backed instance by a process called *bundling*, which stores the current state of the instance's file system on S3 and registers this state with the Cloud App Store as new AMI. The data in an S3-backed instance is only persistent for the life of the instance and lost upon instance termination or failure. This resembles the usage model of a live CD.

EBS-backed AMIs. EBS-backed AMIs reside on the Elastic Block Storage (EBS) [EBS]. EBS offers persistent, attachable block devices, called *volumes*, which can hold an arbitrary file system. When a user instantiates an EBS-backed AMI, a *bitwise* copy of the image is created on EBS and the VM is started from this new image as shown in Figure 3.2. Storing the instance's data persistently on an EBS volume enables the user to stop the execution of an EBS-backed VM. He then has only ongoing costs for the storage occupied by the block device. New EBS-backed AMIs are created by storing a bitwise copy of the current state of a volume, called *snapshot*, on EBS and registering this snapshot with the Cloud App Store as new AMI. EBS volumes cannot only be used to hold bootable AMIs, but are also a general way to store

persistent data. They can be attached on-the-fly to running instances (comparable to a USB flash drive in the cloud).

Networking All instances are executed in an environment which provides logging, monitoring, and security capabilities such as a simple firewall for inbound traffic (cf. [BSP+10]). On startup, the instance is assigned an external IPv4 address for Internet connectivity and an internal address for communication with other EC2 instances. The user is only charged for data traffic with the Internet over the external address.

3.3.2 Authentication in AWS

AWS uses different authentication mechanisms to provide authenticated access to the AWS account and to running instances as described next.

Authentication to the AWS Account. During the initial account creation and verification, an AWS Customer associates her email address with an AWS account, selects a password, and provides credit card and personal information for the monthly billing. After successful verification by Amazon she obtains a password to log into a web management system where she can inspect log files and the current account activity, change personal information, and manage instances. In this web management system, the Customer can register credentials for an Application Programming Interface (API) to control the life cycle of an instance by means of tools provided by 3rd parties or Amazon [Mur08]. We refer to those credentials as *API keys*.[5] The API keys can be used for example to start or terminate instances, create EBS volumes, or to register public images. Moreover, API keys are required within an S3-backed instance during the bundling of the same instances in order to access the S3 storage. However, API keys do not provide direct access to personal or credit card information.

Authentication to Instances. After an instance has been started, the control needs to be securely transferred to the Customer by providing her with a secure and authenticated login channel. For this, Linux/Unix-based instances commonly use Secure Shell (SSH) [YL06]. Here, a customer generates an SSH key pair $(\mathsf{pk}, \mathsf{sk})$ and registers the public key pk with AWS. Upon instantiation of an AMI, this key is automatically made available to and imported by the SSH server running in the newly created instance and can be used for secure logins of the user who holds the corresponding secret key sk.

3.4 AMI Privacy Analysis

By systematically examining only little more than 10% of all public Linux-based AMIs we were able to extract a lot of private information. The results of our analysis are summarized in Section 3.4.1. To automate the analysis we implemented a tool as described in Section 3.4.2. We calculate the costs for analyzing all AMIs in Section 3.4.3 and discuss the reasons for and implications of our analysis in Section 3.4.4.

3.4.1 Our Findings

We analyzed a total of 1225 AMIs in the European and US-East region of the Amazon EC2 cloud. In our privacy analysis of these AMIs we discovered AWS API keys, private keys and

[5] Technically, API keys provide authorized access to a SOAP, Query, or REST-based web service.

credentials, and private data, including source code. For each of these categories we describe the possible threats and our findings next. A summary of our findings in EBS-backed AMIs is given in Table 3.1.

Finding	US-East-1		EU-West-1	
Analyzed AMIs	550	(100%)	550	(100%)
AWS API Keys	12	(2%)	2	(0.35%)
SVN Credentials	4	(0.7%)	3	(0.55%)
SSH/SSL User Keys	14	(2.5%)	5	(0.9%)
SSH Host Keys	205	(37%)	122	(22%)
SSH Backdoor	253	(46%)	93	(16%)

Table 3.1: Summary of the Findings of our Privacy Analysis of Public EBS-backed AMIs (April 2011)

AWS API keys

Our most significant findings are API keys (cf. Section 3.3.2) which could be used to abuse the AWS account of the AMI Publisher. Those keys had been used by the Publisher of the examined AMIs to query the AWS API, in most cases for authorization before publishing and registering the running instance as new AMI.

Threats. The presence of AWS API keys in public AMIs is a very serious threat to the owners of those keys. It allows the adversary to destroy the victim's virtual infrastructure or to create his own infrastructure at the expense of the victim [Clo10b]. For example, an attacker could publish an AMI used for a DDoS attack, and then instantiate his AMI multiple times at the expenses of other cloud customers using the API keys he discovered in the other customers' public AMIs. This is in particular a problem as it is currently not possible to set a maximum limit of costs to prevent excessive usage in case of an accident (e.g., misconfiguration) or security breach [AWSa]. With one AWS API key an attacker can cause costs of more than $3000 per day for running instances and additional charges for traffic and storage[6]. Moreover, before the introduction of improved key management with the *Identity and Access Management (IAM)* [IAM] service in September 2010, customers had to use a single API key to control all their AWS services like EC2, the S3 storage or the SimpleDB database [SDB] engine. It seems that many EC2 customers have not migrated to this new service yet, e.g., because of unawareness or of being technically overwhelmed, and thus a discovered API key can be used to extract private information also from AWS services beyond EC2.

Findings. We retrieved 20 AWS API keys from S3 and EBS-backed AMIs in the US-East and European region.

Private Keys and Credentials

Private keys and login credentials are quite frequently stored or cached on a computer's hard drive. For instance, they are used to login to other hosts via SSH or to provide secure communication channels using SSL certificates.

Threats. Although most private keys are encrypted with a password when stored on disk, this data can be recovered [YBAG04]. Thus, these unintentionally published keys and credentials can compromise the security of the Publisher's IT infrastructure. For instance, an attacker

[6] For the exact calculation we refer to [BPN+11]

can use private keys of SSH user authentication to log in to other hosts the key is authorized for. These hosts may be located within the virtual infrastructure of the Publisher in the cloud or even in the conventional IT infrastructure of the Publisher. Leaked private keys of valid SSL certificates that are used on commercial websites/services would allow an attacker to carry out man-in-the-middle attacks on those sites or implement ideal phishing attacks.

Findings. We discovered 16 unencrypted private keys for SSH user authentication, 3 unencrypted private keys of valid SSL certificates, and various other private keys. Most of the SSH keys we found were associated with an account with administrative privileges.

Private Data

Most AMIs contain information about their Publisher, resulting from the configuration and usage of the instance from which the AMI was created. This information can be obtained by analyzing, e.g., log, cache, or configuration files.

Threats. Private data can be used to profile Publishers, e.g., to identify their name, email address, other hosts they connected to, browser history, or their affiliation. The value of private information found by a random attacker is hard to estimate. However, leakage of private information might harm the reputation of a person or company, cause financial harm, or even entail legal consequences.

Findings. In one particular AMI we found a picture of the owner of the AMI, holding a badge stating his name and employing company. In other cases we were able to find out the name of the AMI publisher by using information in the bash history and the name of the SSH key used for login. However, we cannot give concrete numbers as privacy violations are hard to categorize.

Source Code

The Amazon cloud is well suited for software development and testing of new software products and especially start-up companies make use of it.

Threats. Obtaining and analyzing source code of new proprietary applications or websites allows an attacker to steal unpublished ideas and concepts. Moreover, it gives him enough insight to mount further attacks on these products (or their users) and even the ability to insert malicious code if he gains access to the source code repository.

Findings. We obtained credentials for 7 different private source code repositories. Most of these repositories were operated by businesses and the repository's copy stored in the AMI contained highly valuable intellectual property and hard coded passwords for administrative access.

3.4.2 Tool for AMI Privacy Analysis

Next, we explain our tool for privacy analysis of AMIs and give details on its technical background and our strategy for cost- and time-efficient attacks. We have implemented our tool in the Python programming language and made use of the Boto library [Mur08] in order to access the AWS API. We concentrated our analysis on Linux based AMIs, which form the majority of the provided AMIs (see [Clo] for detailed statistics), and thus designed our tools specifically for the Linux directory structure and key formats.

Figure 3.3 depicts our approach of iteratively inspecting public AMIs for private information. It consists of (1) the instantiation of target AMIs, (2) the search for fingerprints, private

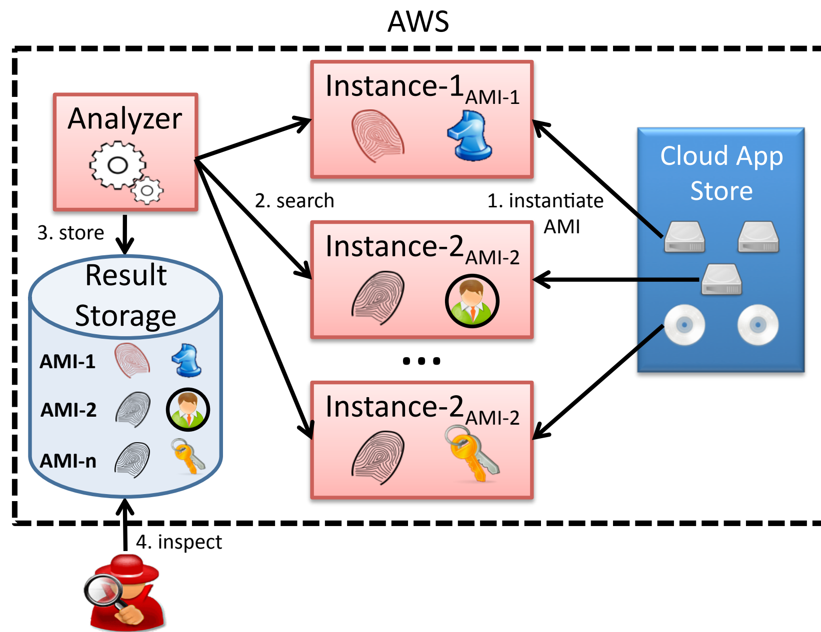

Figure 3.3: Approach for Analysis of AMIs

information, or keys, and (3) the storing of the results in a data store. The last step (4) is a semi-automated inspection and analysis of all findings.

We describe how our tool accesses the data inside the AMIs (Section 3.4.2) and extracts private content from it (Section 3.4.2).

AMI Access

In order to analyze an AMI we mount its volume directly into the file system of our Analyzer instance (cf. Figure 3.3). By running it in the cloud, we avoid expensive network traffic (e.g., downloading the image) and are able to profit from the scalability of Cloud Computing. Further by mounting target AMIs as volumes or via the SSH File System (SSHFS) [Sze] we minimize assumptions on the environment our analysis tools runs in (e.g., software dependencies). We describe the two methodologies in more detail next.

SSHFS. The SSHFS allows to mount remote volumes into the local file system by means of the SSH protocol. As SSH is the standard tool to perform system administration on EC2 Linux based instances and thus is supported by most AMIs, this approach is widely applicable. Although this method does not make any restrictions on where the Analyzer is run (e.g., it could be executed on a local host), deploying it on a dedicated EC2 instance reduces the network overhead significantly. Moreover, Amazon does not charge for cloud-internal data traffic.

EBS-Mount. Analyzing EBS-backed images is substantially faster than using SSHFS as EBS volumes provide raw disk access. As described in Section 3.3.1, EBS-backed images are stored on and booted from EBS volumes. Therefore, our analyzer tool first starts a public EBS-backed AMI to instantiate it on a new EBS volume. By dissolving the mapping between this instance and the new volume its file system is stored on, the tool obtains the EBS volume that contains the AMI to be tested. This EBS volume is then mounted by the Analyzer instance. This method allows also to analyze EBS-backed AMIs that are explicitly configured without SSH access (e.g., to protect the source code of an appliance).

Pattern	Explanation (Common Usage)
*.pem	Specifies a file containing a private key, public key or a certificate
*.priv	File extension for *priv*ate keys
*.pub	Used to store *pub*lic keys
*.crt	*Cert*ificates
id_rsa	Default file name of private SSH keys
*.gpg | *.pgp	Files related to use with the encryption and signing application GPG/PGP
*.jks	Acronym for Java key-store [PK00]
secret | *key* | *private*	Potentially interesting files
.bash_history	User's history of executed commands
.svn/ | .git/ | .hg/	Common source code repositories

Table 3.2: Example of Search Patterns ($*$ denotes the wildcard character and | alternatives)

Extraction of Private AMI Content

Once an AMI is mounted into the file system of the Analyzer instance, the Analyzer searches for keys and private information in the AMI and stores its findings locally for later evaluation. Some of the patterns for file or directory names that our tool is searching for are listed in Table 3.2. These patterns include common names for key files/stores, shell history files, and source code repository directories.

The primary targets of the automated search are high-value findings, i.e., private keys and credentials, usually to be discovered in the home directories of users, the home directory of the superuser (root), and common locations for (custom) programs and configuration files.

If a search result indicates the presence of further, not automatically discovered private data, e.g., a source code repository directory was found or the shell history is non-empty, we manually investigate the corresponding AMI for further findings such as source code or private information.

Forensic Analysis of EBS Volumes. As described in Section 3.3.1, EBS-backed AMI instances are created as a *bitwise* copy of the original AMI volume they are derived from and hence provide access to raw blocks on the newly created volume. This enables our analyzer to apply forensic methods [GS03, Gar09] in order to recover and scrutinize deleted files which may contain private information.

3.4.3 Costs for AMI Privacy Analysis

Our analysis tool described in Section 3.4.2 currently takes approximately 10 minutes to start and completely analyze an AMI. However, as Amazon charges for each started instance hour [Amaa], the costs for starting an AMI are one instance hour. For EBS-backed AMIs the smallest available type is the "Micro" instance for $\$0.03/\mathrm{h}$. The costs for S3-backed AMIs depend on the AMI's architecture: a 32-bit S3-backed AMI can be started as "Small" instance for $\$0.10/\mathrm{h}$ while 64-bit AMIs only support the "Large" instance type for $\$0.40/\mathrm{h}$. A medium scale analysis can even be done "for free" as Amazon offers free services to new customers in the first year (currently 750 "Micro" instance hours per month plus some storage and traffic volume).

Costs of Our Analysis. During our analysis, we examined 1225 AMIs (100 S3-backed 32-bit AMIs, 25 S3-backed 64-bit AMIs and 1100 EBS-backed AMIs). As we have split our analysis into two months, the costs for starting the EBS-backed AMIs were covered by the free offer and for the S3-backed AMIs we paid in total $20.

Costs for Analyzing All Public AMIs. Scanning *all* free and Linux based 9864 public AMIs that are currently available[7] (cf. Figure 3.4) would cost approximately $1550: Starting all AMIs would cost around 3058 "Micro" instance hours, 4449 "Small" instance hours, and 2357 "Large" instance hours. As the Analyzer component of our tool takes on average 10 minutes per AMI this adds $\frac{9864*10}{60} = 1644$ "Micro" instance hours. We note that such an exhaustive search can be highly parallelized and therefore carried out very fast, e.g., in less than one day with 70 analyzing instances, without increasing the costs.

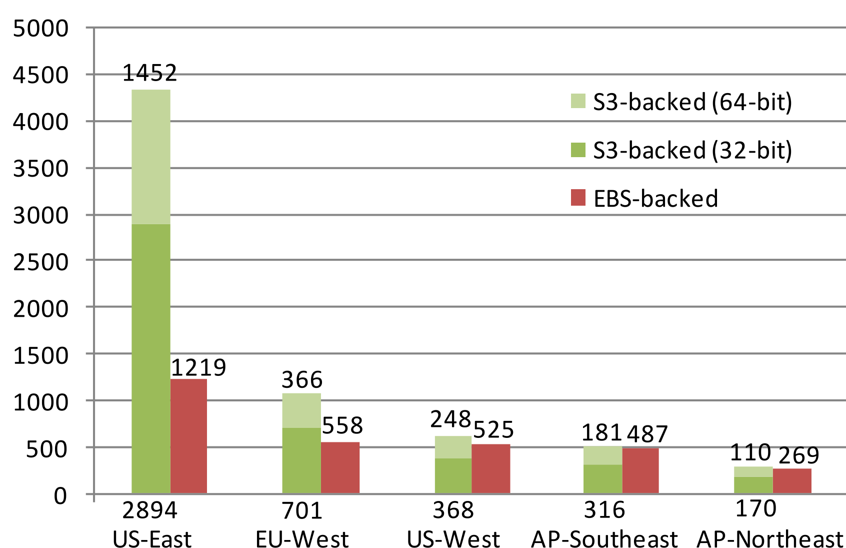

Figure 3.4: Linux Based Public AMIs per Region

Attack Optimization. One could even reduce the costs for the attack to almost zero by first scanning public AMIs that are likely to contain AWS API keys and then using these keys to start further analyzer instances at the expense of these initial victims. Candidates for the initial attacks are AMIs from inexperienced or unaware users, i.e., those who published only few AMIs or give suspicious names to their AMIs such as "backup" or "test".

3.4.4 Discussion

We discuss the reasons for and implications of our successful extraction of private information from public AMIs.

Reasons for Forgotten AWS API Keys We discovered AWS API keys in both S3- and EBS-backed AMIs. Our inspection of S3-backed AMIs revealed that those keys were unintentionally included during the bundling process when the AMI was created. Since S3-backed instances require an API key within the instance for bundling, the success rate to find API keys within this kind of AMIs is high. Our inspection of EBS-backed AMIs showed, that the Publishers

[7] Note that the number of public AMIs changes frequently.

used their API keys within their instances in order to access further AWS services such as S3 or EBS storage, and that the Publishers simply forgot to delete the key before publishing.

EBS Specific Problems In general our analysis shows that EBS-backed images contain more private information than S3-backed AMIs. We attribute this to the fact that data in EBS-backed instances is persistent (cf. Section 3.3.1). In fact, EBS-backed AMIs are much more dangerous than S3-backed AMIs for various reasons:

Loss of intellectual property. Publishers who want to protect their intellectual property by limiting access to the actual VM should not distribute it as an EBS-backed AMI. Indeed, in our analysis we found appliances which were configured explicitly without SSH access but could easily be inspected by mounting them as EBS volumes (cf. Section 3.4.2).

Forensic attacks. EBS backed AMIs or EBS volumes that once contained *any* valuable information, should not be made public. By applying forensic methods [Gar09, GS03] on a raw EBS volume we were able to reconstruct and inspect several deleted files, containing private information.

Snapshots. Snapshots are non-bootable EBS volumes with the purpose to ease sharing (of large volumes) of data. Snapshots can be created as new volume and filled with data, but they can also be created from instances. Thus, they also bare the risk of unintentionally publishing (forensically retrievable) private information. For instance, a first analysis of various snapshots revealed a private picture of the Publisher.

3.5 Cloud Specific SSH Threats

In this section we discuss vulnerabilities of the Secure Shell (SSH) protocol [Hat04], which result from the cloud's dynamic nature and the usage model of AMIs. In particular, we evaluate SSH-based backdoors of instances (Section 3.5.2) and present new attacks due to the SSH-based identification of the AMI from which an instance is derived (Section 3.5.3).

SSH provides confidentiality and integrity, supports asymmetric key pairs for user authentication ($\mathsf{sk}_u, \mathsf{pk}_u$) and for host authentication (sk_h,pk_h). The SSH protocol is well-established and is the primary method for remote administration of Linux based instances in EC2 (cf. Section 3.3.2). We first provide a very brief explanation of the SSH protocol in Section 3.5.1 to lay the foundation for the better understanding of the remainder of this section.

3.5.1 SSH Authentication

As described in Section 3.3.2, SSH [Hat04] is the primary method for remote administration in EC2. The SSH protocol supports user authentication based on passwords or asymmetric key pairs and has built in protection against man-in-the-middle attacks as described next.

In the cloud environment mainly the asymmetric key based method is used. It is supported by the EC2 infrastructure as well as most public AMIs, and is considered to be more secure than password based authentication. As mentioned in Section 3.3.2, the AMI user's public key pk_u is imported by a newly created instance (called host in SSH context) from EC2 and the user then authenticates herself by proving possession of the associated secret key sk_u. In addition to that, SSH also supports a check of the host's identity to prevent an attacker from impersonating a remote host or launching a man-in-the-middle attack. This is realized with an unique and unforgeable asymmetric key pair ($\mathsf{sk}_h, \mathsf{pk}_h$), called host key which is stored on the remote host. When a user connects to a host he has to verify that the fingerprint, i.e., a hash, $\mathsf{f}(\mathsf{pk}_h)$ of the host

key belongs to the machine he wants to connect to (cf. Figure 3.5). As no Certificate Authority is involved, it is the responsibility of the user to ensure the authenticity of the fingerprint when connecting to a new or unknown host, e.g., over an outbound channel. After the key has been accepted, the SSH client application caches the mapping between the key and the host's IP address or hostname.

```
john:~$ ssh ec2-user@47.127.0.233
The authenticity of host '47.127.0.233 (47.127.0.233)' can't be established
    .
RSA key fingerprint is f3:5a:f4:a2:f3:d1:e5:52:2c:e3:87:ff:07:13:01:11.
Are you sure you want to continue connecting (yes/no)?
```

Figure 3.5: SSH First Connection

3.5.2 AMIs with SSH Backdoor

Recently, Amazon informed several of its customers that a public AMI contained an SSH user authentication key pk_u and thus a backdoor allowing the publisher of the AMI who holds the corresponding sk_u to log into instances derived from that particular AMI [Ama11c]. Amazon strongly advises customers to terminate such instances and regards them as compromised [Ama11b]. However, this problem is not an isolated incident and, as we will show in this section, many of the publicly available AMIs contain SSH backdoors.

Threats. An SSH user authentication key deployed in a public AMI poses a severe threat to the AMI Consumer's privacy, as the key owner is able to log in to the affected Consumer's instances. A potentially malicious Publisher is thus able to deliberately eavesdrop or modify the Consumer's data and services in the instance.

Findings. Our analysis revealed that 30% of the 1100 analyzed EBS-backed AMIs in the European and US-East region at the time of the analysis contained pk_u and thus a backdoor for the AMI Publisher (detailed numbers are given in Table 3.1). By examining the affected AMIs we discovered that the backdoor problem is not limited to AMIs created by individuals, but also affects appliances of well-known open-source projects and even of companies that offer IT-security products. Moreover, our investigation yields that an overwhelming number of AMIs allows SSH login and that most users have administrative privileges[8].

Approach

Our approach to identify AMIs with a SSH backdoor is based on our analysis tool presented in Section 3.4.2. For this, we analyzed the *.ssh/authorized_keys* file. It is located in the users' home directories and contains the public keys pk_{u_i} of users that are allowed to log in.

Discussion

Reasons for Forgotten Public Keys. The reasons why such a huge number of AMIs contain backdoors for the Publisher are similar to those for unintended inclusion of private information as discussed in Section 3.4. However, the Publisher has no strong incentives to remove his public key before publishing an AMI, as the only negative consequence of not doing this is that he can be traced among AMIs.

[8] Many Linux distributions for EC2 allow the default user to execute commands as superuser via "sudo".

Countermeasures against SSH Backdoors. In case a Consumer decides that she wants to run a particular AMI (e.g., because of the unique features offered by this AMI), she should check the *authorized_keys* file in every home directory and delete all public keys pk_u (except for her own). In addition to that she should examine the configuration of the SSH server in case other authentication methods (like insecure password authentication) are enabled or other locations for authorized keys are specified. Another countermeasure is to use the Amazon provided inbound firewall to generally restrict the IP range from which users are allowed to login via SSH. It also supports features for the implementation of a multi-tier infrastructure with a central gateway that validates access attempts to other instances [BSP+10].

3.5.3 AMI Identification via SSH

Some public AMIs contain an SSH host key pair $(\mathsf{sk}_h, \mathsf{pk}_h)$ which is generally used to prove the identity of the remote host to the client during login (cf. Section 3.5.1 for details). Usually, a host key pair is generated from fresh entropy when installing SSH or the operating system. However, in cloud apps, where instances of an AMI are only a copy of the hard drive's state of an already configured machine, the SSH host key pair is not regenerated. Hence, all instances of an AMI are using the same SSH host key pair.

The severity of this vulnerability can be amplified by combining it with the technique for extracting information from public AMIs described in Section 3.4.2. The attacker can examine several (or all) public AMIs and extract the SSH host key pairs $(\mathsf{sk}_h, \mathsf{pk}_h)$ from it. Afterwards, he can use the host key fingerprint $\mathsf{f}(\mathsf{pk}_h)$ of an instance to look up the corresponding public AMI and the secret key sk_h.

In the following, we describe the consequences of our attacks, our findings, our approach, and the underlying causes.

Threats

The possibility to identify instances that are using the same SSH host key or potentially even identifying the corresponding public AMI from which these instances are derived allows a variety of attacks as summarized in Table 3.3 and described in the following.

Attack	Prerequisites
Correlation of System Configurations	-
Whitebox Attacks	public AMI identified
Impersonation Attacks	
Man-in-the-Middle Attacks	
Co-Location Attacks	
Phishing Attacks	owner of public AMI

Table 3.3: Attacks based on identical SSH host keys.

Correlation of System Configurations. Even if the attacker only detects instances with an identical fingerprint $\mathsf{f}(\mathsf{pk}_h)$, but does not manage to identify the corresponding AMI (e.g., because it is not public), he knows that these instances are likely to be instances of the same AMI (or a derived AMI). This enables him to discover test systems or forgotten servers which are not as secure as the productive system.

If the attacker identifies the corresponding public AMI he has more options:

Whitebox Attacks. The attacker can examine the contents of the AMI for potential vulnerabilities and misconfigurations like usage of default passwords or insecure services to launch attacks on the victim instance.

If the attacker extracts the host key pair $(\mathsf{sk}_h, \mathsf{pk}_h)$ from the AMI and has control over the network between the Consumer and the Provider (e.g., [DY81]) he can launch impersonation and man-in-the-middle attacks on SSH:

Impersonation Attacks. The attacker can redirect the SSH connection attempt of the Consumer to an instance under his control. In this case, verification of the host key fingerprint $\mathsf{f}(\mathsf{pk}_h)$ does not protect the Consumer as the attacker is able to equip his fake instance with the host key pair obtained from the public AMI. The Consumer may get suspicious that the impersonating instance is only similar to the expected environment, but there is a chance that the Consumer reveals private information or credentials before recognizing this, e.g., passwords upon login to other services. The attack especially works when automated scripts are used, e.g., a script that automatically uploads a backup copy onto an instance in the cloud. By impersonating this instance the attacker can obtain such a backup copy.

Man-in-the-Middle Attacks. If SSH is configured for password-based user authentication, an attacker can use sk_h to play as man-in-the-middle and eavesdrop or modify the communication between the Consumer and the VM.

Co-Location Attacks. The AMI type narrows down the possible instance type as 32-bit AMIs do not support the more powerful instance types because they cannot make use of the offered resources. This information can be exploited by an attacker to narrow down the search space in co-location attacks [RTSS09].

Phishing Attacks. If the attacker himself is the Publisher of the executed malicious AMI he can deniably identify victims of his own VM phishing attacks. The malicious AMI might contain intentionally embedded vulnerabilities [ASM09, WZA+09] or SSH backdoors (cf. Section 3.5.2).

Findings

Our analysis shows that approximately 29% of the 1100 analyzed AMIs (in US-East and EU-West) are misconfigured such that their derived instances do not recreate the SSH host key pair $(\mathsf{sk}_h, \mathsf{pk}_h)$ on first boot (cf. Table 3.1). We discovered that 62 distinct SSH host keys in the EC2 Tokyo region were used by more than one instance. We identified the public AMI of 11 of those host keys, which were contained in 278 instances running in this region.

Our experiment also revealed that $\frac{604}{2533} \approx 23\%$ of the instances in the Tokyo region with SSH access are using a non-unique host key pair $(\mathsf{sk}_h, \mathsf{pk}_h)$. However, we were not able to map all duplicate SSH host keys to public AMIs and believe that these misconfigured instances are derived from private AMIs owned by one user or a small group and are thus potential production or test systems with similar vulnerabilities (cf. *Correlation of System Configurations* described above).

Approach

Our approach for finding public AMIs using not freshly generated SSH host keys is illustrated in Figure 3.6. Based on our analysis tool for public AMIs (cf. Section 3.4.2), we first extracted the SSH public key fingerprints $\mathsf{f}(\mathsf{pk}_h)$ of all AMIs in the Tokyo region and used this to obtain a mapping from 615 fingerprints to their corresponding AMI ID. Further, by making SSH connection attempts to all IP addresses in the Tokyo region's IP range (175.41.192.0/18), we gathered

all mappings from IP addresses to host key fingerprints for the region. By matching the SSH fingerprints in both results, we found a mapping between the IP addresses of running instances to their corresponding AMI ID. Moreover, the diversity of SSH fingerprints in the region's IP range gave us the statistic about duplicated SSH host keys as explained in Section 3.5.3.

AMI	SSH ID
ami-124	EE:CS:DS
ami-257	B2:D9:24
ami-374	B2:D9:24
ami-424	D4:25:13
...	...

1) Scanned AMIs

3) Matching

SSH ID	IP Address
A1:CS:DS	10.122.230.214
B2:D9:24	10.122.230.215
C3:9A:42	10.122.230.216
D4:25:13	10.122.230.217
...	...

2) Scanned IP addresses

Figure 3.6: Matching Between AMIs and Instances

Discussion

In summary, our findings in Section 3.5.3 show that AMIs which do not recreate their SSH host key occur frequently.

The main reason for SSH host keys being included in public AMIs is presumably the same as for private information (cf. Section 3.4.4) and SSH user authentication keys (cf. Section 3.5.2) and, thus, countermeasures are similar. We therefore extend this discussion to the whole process of secure authentication and administration in the cloud, as it is negatively affected by the flexibility and usage model of computing clouds.

Alternatively, an attacker could use other publicly available information than the SSH host key to identify the AMI of an instance, e.g., the software version and configuration of a web-server running in the instance. However, the SSH host key is a much more distinguishing characteristic than the software configuration. Nevertheless, this is supplementary information to identify the AMI of an instance if more than one candidate image exists (e.g., *ami-257* and *ami-374* in Figure 3.6).

Static vs. Dynamic Environments So far, the SSH protocol has been used mostly in and is designed for relatively static networks. Hence, manual verification of the host keys is only necessary in case of relatively infrequent changes to the underlying infrastructure (e.g., change of the server's IP address or hostname, or newly generated host key). In contrast, the cloud provides a highly dynamic environment with much shorter life time of instances and dynamically assigned IP addresses. Additionally, the use of preconfigured VMs instead of installation from scratch opens new vulnerabilities as shown in Section 3.5.3.

Trust on First Use This leads to the general challenge of secure authentication to started EC2 instances which is not sufficiently addressed in the current implementation of EC2 and even worse for AMIs with non-unique host keys. As described in Section 3.3 and shown in Figure 3.2, the Consumer issues a request to start a public AMI and is returned the Instance-ID which can be used to look up the IP address of the instance (over the AWS API or Web

Interface). When she connects to the IP address of this newly created instance, she has no prior knowledge on the contents of the AMI and its host key ("Trust-on-first-use", cf. [WAP08]). The only option is to use an out-of-band method, provided by Amazon, to obtain the console output of the started instance to which the SSH server writes the host key fingerprint $\mathsf{f}(\mathsf{pk}_h)$ after the host key pair has been freshly created. However, as it takes a few minutes until the output of the console becomes available to the user, we assume that most users do not make use of this technique when establishing the first connection to a newly started instance. The situation is even worse when the instance's SSH server does not create a fresh host key pair where the user has no means to verify that he really connects to the machine he started.

This problem is aggravated when the instance is stopped and restarted with a newly assigned external IP address. In this case, the console output does not contain the host key fingerprint as no new host key was generated, and the user is forced to store the correct fingerprint upon the very first connect in order to be able to securely authenticate his instance after reboot or change of the external IP address.

3.6 Countermeasures

Finally, we present and discuss several countermeasures that create awareness, reduce the impact of lost credentials, or assist users in realizing that they accidentally published sensitive information. Some general countermeasures have been proposed and discussed in [WZA+09], however, we believe that the high popularity and deep integration of Amazon specific services, e.g., S3 and EBS, into the EC2 Cloud App Store prevents adoption of these countermeasures as it would require fundamental changes to the architecture. We therefore present mechanisms that specifically address the environment provided by Amazon and its limitations.

Organizational Measures Most problems that led to our attacks could have been prevented by the Publishers themselves. Especially as most public AMIs do not provide any additional value to other consumers, e.g., due to lacking functionality and documentation, they should not have been made public in the first place. Therefore, the first and most effective step should be to provide better information to users on the risks they are facing when publishing an AMI and to create problem awareness. In addition to that, despite the novelty of Cloud Computing and AMI publishing, companies and individuals should develop and follow guidelines, best practices, and processes for releasing their AMIs to the public. In this case AMIs do not differ from documents, software, or mediums containing confidential information (e.g., discharging of old hard drives, cf. Section 3.2). Another effective way to reduce the impact of an accidentally lost AWS API key is the recently introduced AWS Identity and Access Management (IAM) [IAM]. It allows managing of multiple security credentials for an AWS account with a different set of permissions which could be used to limit the damage an attacker can cause.

Tool Assistance for the Publisher Additionally, Publishers can be protected by enhancing Amazon's toolchain.

S3-backed AMIs. As discussed in Section 3.4.4 some Publishers of S3-backed AMIs have exposed their AWS API key used for authorizing the bundling operation [Bun]. We propose to extend the bundling command in a way that a warning is issued in case the AWS API key is included into the new bundle (cf. Figure 3.7 for an example) or information about the key is stored in the command line history. Moreover, the extension can provide a *safe bundling* option

to warn Publishers about data which is potentially private (e.g., by applying search patterns similar to those in Table 3.2).

```
$ ec2-bundle-vol -k /root/sk-HKZYCLO.pem
WARNING: The key sk-HKZYCLO.pem used to authorize the bundling
    operation will be included in the image file. Publishing the AMI
    may leak it to the public!
Do you want to proceed (y/n): n
```

Figure 3.7: Improved Bundling Tool

EBS-backed AMIs. Protection for EBS-backed AMIs has to be implemented differently than for S3-backed AMIs. EBS-backed AMIs can be almost instantly generated out of running instances and made public entirely over the web interface. As the underlying storage engine just instructs the storage layer to create a bitwise copy of a volume, we have to apply different countermeasures. Here, Amazon could extend their interface to inform the user on potential risks of private data being included and also offer tools to deal with them. This can be realized by a service (e.g., a public AMI) that is given an EBS volume as input, creates a report on potential privacy problems, and outputs a low-level sanitized volume to prevent forensic analysis.

Regular Scanning As described in [MS10], Providers have to find a trade-off between the benefits of providing security measures and the costs for implementing and operating them. However, we believe that it is possible for a provider like AWS to regularly scan the provided AMIs (cf. cost analysis in Section 3.4.3) while still ensuring that the Provider does not take any liability on undiscovered problems. As hackers are likely to re-implement the techniques described in our report (especially given the high damage that can be caused), public cloud service providers should take immediate action.

For this, Amazon and other providers could adopt our tools, eventually increase their efficiency by integrating them into their infrastructure, and scan the Cloud App Store regularly, or when a new AMI is published. Although the scanning may result in more time and effort for manually verification of the results and informing affected customers of a discovered vulnerability, this approach would underline a proactive approach to security problems.

Improving the Cloud App Store As we have described in Section 3.5 a great deal of AMIs, deployed by a large number of consumers, does not satisfy even lowest quality standards (e.g., contains an SSH backdoor). This situation can be improved by combing an automated rating system that checks AMIs for predefined properties together with a reputation system that allows users to evaluate the usefulness and quality of an AMI. Such features are commonly available in today's mobile app stores as well as for online auction and shopping platforms to rate vendors and products, but still missing in Amazon's Cloud App Store and its web console. Moreover, the AWS web console should be extended to provide more information on particular AMIs like the date of publishing or more detailed descriptions about the AMI's purpose and its publisher. A process for submitting AMIs and extended documentation already exists, but this information [Amab] is not available in the web console. Additionally, well known and trusted AMI Publishers could be rewarded with a special status visible to the consumer (star/gold Publisher). Before earning such a status, a Publisher would have to earn good reviews or pass an analysis of their AMIs by Amazon.

Infrastructural Measure In contrast to the above mentioned countermeasures, which do not require any crucial modifications to the cloud infrastructure, also more fundamental approaches are possible (e.g., in the context of the TClouds project). For example, the solution proposed in [WZA+09] aims at optimizing the efficiency and effectiveness of scanning virtual machine images such as AMIs. Their solution is based on "deduplication" of the images' storage[9] and thus requires crucial changes to cloud infrastructure.

[9] Deduplication means, that redundancy in stored data is removed by storing each distinct data only once (e.g., at the granularity of files) and using pointers from each actual duplicate to the unique data.

Chapter 4

Security Analysis of OpenStack

Chapter Authors:
Sören Bleikertz (IBM), Sven Bugiel (TUDA), Zoltán A. Nagy (IBM), Stefan Nürnberger (TUDA)
Anil Kurmus (IBM), Matthias Schunter (IBM)

4.1 Introduction

OpenStack is an open source Infrastructure as a Service (IaaS) cloud computing platform initiated by RackSpace and NASA. The initiative is joined by other major vendors and startups in the field of cloud computing, such as Citrix, Dell, Cloudkick, AMD, and Intel. The code base of the platform is written in Python (about 70.000 lines of code) and is licensed under the Apache open source license. The mantra of this initiative is that it "strives to become the open source standard for building cloud infrastructures everywhere".

Community

The OpenStack project started in July 2010 and already consists of an active and diverse community as represented in the OpenStack Design Summit 2010 where 250 people from 12 countries participated. The development cycles of the project are very short and new releases are planned every 3 to 6 month. In February 2010 a stable version was released that is suitable for mid-sized deployments with regard to compute resources and production ready for storage resources. The April 2010 release is planned to be production ready for large scale service providers.

The community is very diverse and consists of contributors from a variety of organizations. There seems to be no single organization behind the project, although it was initiated by RackSpace and NASA that are still major drivers in the project. Over 25 companies are supporting the project and they adopted an open integration process for changes to the code base, i.e., improvements to the code base from new contributors are welcomed and accepted.

The project incorporates professional development practices in order to ensure good quality of the software. Among these practices are unit tests, code reviews, code documentation, and continuous integration. Furthermore, the development and planning of the project are transparently conducted on the open source software collaboration platform *launchpad.net*, which is also used for the development of the popular Ubuntu Linux distribution.

4.2 OpenStack Architecture

4.2.1 Nova: Compute Cloud

Nova contains all the management components that are required to build a compute cloud. It is similar to Amazon EC2 and is based on NASA's cloud project.

Architecture Overview: Figure 4.1 gives an overview of the architecture of Nova. Three controllers manage the resources such as compute, network, and storage: compute worker, network controller, and volume worker. API endpoints collect requests from cloud consumers and a C

scheduler dispatches requests to the appropriate controller or worker. A queue is used for all message-based communication between the services.

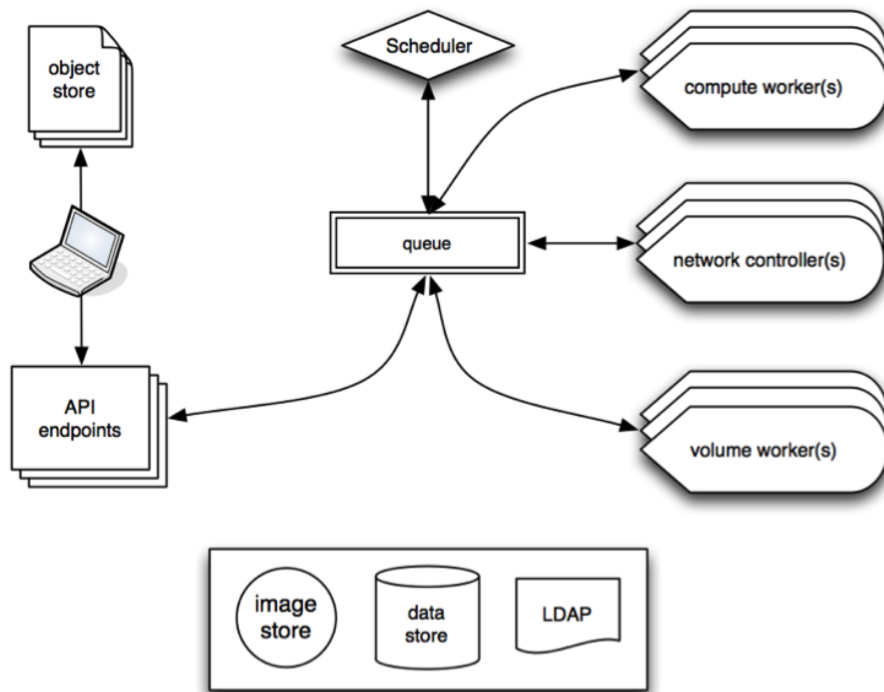

Figure 4.1: OpenStack Nova Architecture (Source: http://nova.openstack.org/service.architecture.html).

Service Communication: The communication of the services in the Nova architecture is realized using message queues. Running all the services separated from each other and only allowing communication using queues reflects the design principles of the Nova architecture: A shared-nothing and messaging-based architecture, holding the state in a distributed data store, and asynchronous calls with call-backs. The message queues are based on the *Advanced Message Queuing Protocol (AMQP)* and Nova uses the *RabbitMQ* implementation of these queues, which is written in Erlang and supports high availability and clustering. Within Nova, message queues are used in a Publish-and-subscribe fashion, where services listen on specific channels,

and other services can write to these channels. Furthermore, it is possible to send messages to specific hosts, e.g., in order to start a VM on a specific host X.

Cloud API: Nova currently supports two APIs: Amazon EC2 and OpenStack. For EC2, a subset of the API is implemented and the open source management tools, such as euca2ools, can be re-used for managing a Nova compute cloud. Furthermore, the *Open Cloud Computing Interface (OCCI)* API is planned to be implemented. The web service providing the APIs is also responsible for authentication and authorization of cloud consumers and their requests. The API server dispatches requests via message to the appropriate services in the architecture.

Scheduler: The scheduler is responsible for choosing a host to run instances on when the API server dispatches a message that a VM is requested to be started. After selecting a host, the scheduler will forward the request to the selected host, which then can start the requested VM. Currently, multiple scheduler drivers exist: *Chance*, which randomly selects a host; *Simple*, which chooses the host with the least load. A few problems exist with the current scheduler implementation, which will be addressed by the OpenStack developers in the future. Namely, that only a single scheduler can exist in the Nova architecture and that the state is kept in a central data store, which forms a single point of failure. For the future, a distributed scheduler is planned, which overcomes these scalability and availability problems.

Compute Worker: The compute worker handles the compute resources on a physical machine. The worker builds disk images for VMs, launches or terminates VMs using a virtualization driver (currently LibVirt and Xen are supported), monitors VM states, attaches or detaches persistent storage volumes to VMs, and provides console outputs from VMs.

Network Controller: The network controller manages the network resources on a host. It basically configures networks and VLANs on a physical machine, but it is not able to configure the network infrastructure such as switches, e.g., in order to setup VLANs on the switch. There exist three different modes for fixed IP addresses: *Flat Mode*, which is a bridged setup that statically allocates IP addresses; *Flat DHCP Mode*, which is also bridged but uses DHCP instead of static IP addresses; *VLAN DHCP Mode*, which uses VLANs in order to provide stronger isolation and provides a VPN gateway to the cloud consumer. Besides fixed IP addresses, there also exists the concept of floating IP addresses that can be dynamically assigned to virtual machines, e.g., in order to assign a static public IP address to a VM.

Volume Worker: The volume worker manages persistent storage volumes that are exported to other hosts. The worker can create and delete LVM-based volumes on a physical host. These volumes can be exported using either iSCSI or ATA over Ethernet (AoE) to hosts, which are running VMs that requested a volume to be attached. Machines hosting volumes for other machines are a single point of failure, and a distributed block storage is needed for resilience. OpenStack developers are looking into adopting Sheepdog, a distributed storage system for QEMU, for this purpose.

4.2.2 Swift: Storage Cloud

Swift is a distributed data blob storage system similar to Amazon S3. This component of OpenStack originates from RackSpace's production cloud storage system and is also used in produc-

tion. With regard to the CAP theorem, Swift only provides eventual consistency.

Architecture Overview: The architecture of Swift is illustrated in Figure 4.2. We can identify three components related to storage: *Account*, *Container* and *Object* servers with their associated rings. The servers store the actual content and the rings are acting as an address book in order to locate the server hosting specific content. For the servers we observe a hierarchy in the stored content, namely that accounts index containers and they contain the actual data objects. A *Proxy* mediates all requests and responses between the servers and the users, and furthermore it uses an authentication and authorization service to validate the user's API calls. *Memcache* is used to cache certain responses within the system.

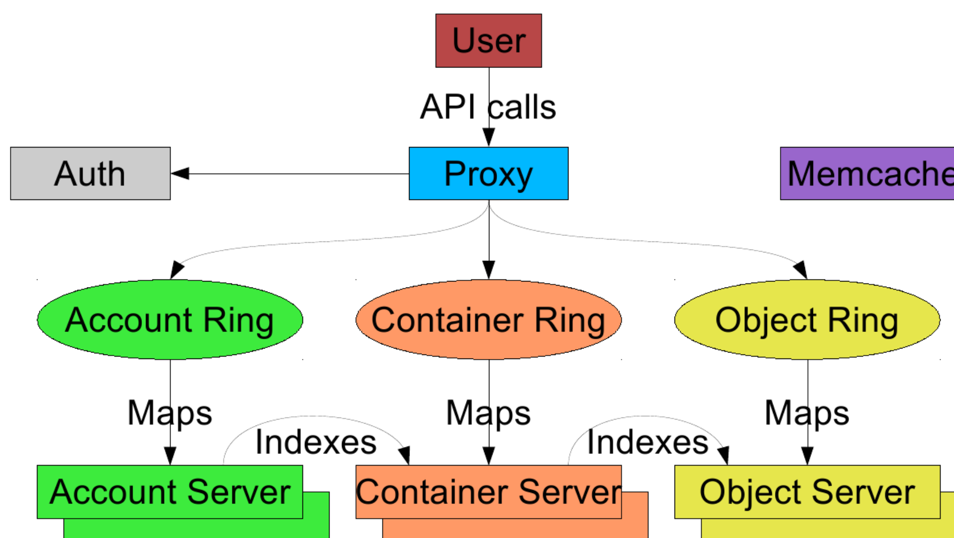

Figure 4.2: OpenStack Swift Architecture.

Proxy & Authentication: The *Proxy* exposes a ReSTful API to the client and dispatches all incoming requests to the corresponding servers. The API is not S3-compatible, although an S3 API middleware is developed that provides such compatibility. Objects, which are requested by the client, are streamed through the proxy from the object server to the client. For authorization and authentication, two services do currently exist: *DevAuth* and *Swauth*. The former allows external auth services to be plugged in, and the later is a scalable auth service based on Swift itself.

Rings: A *Ring* is a mapping of an entity name to its physical location. In the Swift architecture we have separate rings for the different content types, i.e., account, container, and object. Any action on an entity requires to query the ring in order to locate it. The mapping takes into account zones, devices, partitions, and replicas. Each replicate resides in a different zone, which could be a server, rack, or datacenter. Partitions are replicated and balanced across the cluster. Devices are used for handoff in failure scenarios and partitions are assigned to devices. A ring is a statically constructed datastructure (using the ring-builder) and distributed to the servers. In case the configuration of the Swift system changes, rings are recreated and distributed.

Account & Container Server: The functionality of the account and container servers are very similar. Both store an index of the containers or objects respectively in sqlite database

files. Replication of these files to other physical locations is performed by first performing a hash comparison between the source and destination files. If the hashes differ, the records added since a last synchronization point are shared.

Object Server: The object server stores the actual simple data blobs. Objects are stored as binary files with metadata in the filesystem's extended attributes (xattr). The path of the object file is a combination of the object name's hash and a timestamp. In case an object gets deleted, a special "tombstone" file is placed instead of the file, which is also replicated to the other servers, therefore ensuring that other replicas do not serve the deleted file. In case a new version of an object is stored, the older version will be deleted. Large objects are supported with basically infinite size (although depending on the storage cloud capacity) by using client-side chunking that splits the large object into smaller (up to 5 GB) chunks. Replication is done by using *rsync* and pushing data to replica servers. For efficiency reasons, a partial rsync based on hash invalidation is performed.

Updaters & Auditors: There exist two processes which are performed periodically: *Updater* and *Auditor*. The updater updates the index of account or container servers, in case a new container or object is added respectively. Such an update might fail and the update task is queued for later processing, which leads to an eventual consistency window. For example, during this window, a newly added object can be retrieved, but will not be listed in the container. The auditor is responsible for checking the integrity of objects, containers, and accounts. The integrity check is based on a hash comparison for objects, and trying to obtain database information from the sqlite files in case of container and account servers. In case of corruption, the corrupted entity is quarantined and replaced with one from a replica.

4.2.3 Glance: An Image Repository

*Glance*provides services for discovering, registering, and retrieving virtual machine images. Glance also uses a REST API for communication. Glance supports different types of storage, from filesystems, over Amazon S3 to Swift as an object store for virtual machine images (see Figure 4.3[1]).

Glances manages images by assigning a unique ID to each of them and by assigning a status each image can be in.

Image State Glance manages images by one of four possible states assigned to each image[2]:

queued Denotes an image identifier has been reserved for an image in Glance (or more specifically, reserved in the registries Glance uses) and that no actual image data has yet to be uploaded to Glance

saving Denotes that an image's raw image data is currently being uploaded to Glance

active Denotes an image that is fully available in Glance

killed Denotes that an error occurred during the uploading of an image's data, and that the image is not readable

[1]Source `http://glance.openstack.org/architecture.html`
[2]Source: `http://glance.openstack.org/statuses.html`

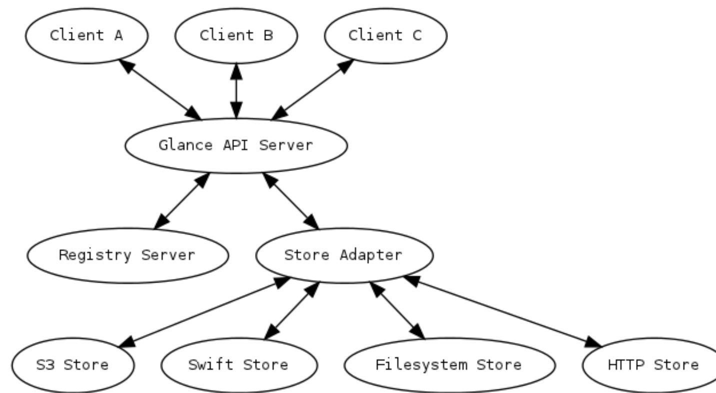

Figure 4.3: The Glance Architecture.

Pubic and Private Images Glance supports storing images that either belong to a certain user account (see *Authentication* in subsection 4.4.3) or they can be public/shared with other users.

4.3 Methodology for Security Analysis

The methodology for the security analysis of OpenStack is to analyze the individual components of Nova and Swift with regard to the following security objectives:

- Confidentiality
- Integrity
- Availability
- Authenticity
- Accountability
- Authorization

In the case that a component fulfills these objectives, it will be listed and explained in Section 4.4, in particular how these objectives are met. Otherwise, we will explain the security shortcomings and their implications in Section 4.5.

Since OpenStack is a fast moving target, we are focusing the security analysis on the *Cactus* release, which was released at 15th of April, 2011.

4.4 Existing Security Mechanisms

4.4.1 Nova

Scalable and Fault-tolerant Architecture Since each component uses the database to store data, and uses the message queue to communicate with each other, high availability is a must for these two. RabbitMQ and both supported databases (MySQL and PostgreSQL) supports clustering and replication to achieve this goal. Apart from this, each OpenStack service must

be made highly available. Currently, failover must be handled outside the provided services' scope. For example, if the network service fails, the running instances will not be able to access anything besides their own VLAN. There are plans to make this less centralized in the upcoming OpenStack releases. The other services (API, scheduler) can be deployed fully redundantly, since they only provide middleware functionality for other services, and only need to access the database and the messaging queue.

Security Groups A security group is a named collection of network access rules, like firewall policies. A VM can belong to any number of security groups. By default, every traffic not explicitly allowed is discarded. However, the default security group allows traffic between VMs of the same group. These groups can be thought of as a security role for a group of VMs, for example, we could create a security group named "webservers", and allow any incoming traffic to port 80 on TCP. The chaining of these groups allows a VM to have multiple service roles. Each rule also has a source, which is either a CIDR address, or an other group's name. Upcoming in OpenStack *Diablo* release, it will be possible to do global blacklisting for IPs.

VLAN Isolation of Cloud Customers The compute service supports different network modes, the default being VLAN DHCP Mode, which provides the most features. For each project, a VLAN ID is assigned, and an interface and bridge is created. The VLAN will get a range of private IPs. The VMs can then only be accessed on this VLAN. However, without assigning public IPs to the VMs, they become inaccessible from the outside. To solve this, special purpose VPN instances can be created (called CloudPipe) that connects, e.g., the enterprise network with the cloud. After starting up, they run OpenVPN, which can be connected to through a public port on the network host for the project. In an IP subnet associated to the project, the second IP address is always reserved for a CloudPipe instance.

Authentication Framework Upon user creation, a pair of keys (the secret and the access key) is generated for the user. The access key is included in web service requests to the API node, and the request are signed using the secret key. A key-pair with a X.509 certificate is also generated that is used for uploading VM images. Currently, authentication credentials can be configured to be either stored in the central database, or to use an existing LDAP server for this purpose.

For authentication on the instances, at least one other key-pair must be generated for SSH. Upon instance creation it is possible to specify which SSH public key should be injected into the image to enable shell access. Upcoming in the *Diablo* release, authentication will be managed by Keystone, a new identity service. Initially using token-based authentication, but eventually supporting plug-in modules for identity storage (LDAP, DB, file, PAM, Active Directory, etc...), protocols (SAML, OAUTH, OpenID, etc...), and necessary middleware to support integration with OpenStack core, affiliated, and compatible services.

Role-based Access Control OpenStack uses Role-based Access Control (RBAC) to make authorization decisions upon receiving an API request. Role assignments can be either global or per-project. The following five roles are provided by default:

- Cloud Administrator (admin): Global role, providing complete system access.
- IT Security administrator (itsec): Global role. It permits modifying the security group configuration.

- Project Manager (projectmanager): Project role. This is the default role assigned to project owners. It allows every operation to be performed, as it happens in the admin role, but in a different context, which is the context of the project.
- Network Administrator (netadmin): Project role. Users with this role are permitted to allocate and assign publicly accessible IP addresses as well as create and modify firewall rules.
- Developer (developer): Project role. This is a general purpose role that is assigned to users by default. This allows only to manage access keys within a project.

Accounting: Quotas For accountability purposes as well as maintaining resources availability for cloud users, the following resources can be limited on a per project basis:

- Total number of CPU cores
- Total amount of memory
- Total number of instances
- Total number of volumes
- Total size of volumes across all instances
- Total number of floating IPs

In the future it is also planned to introduce quotas on network traffic and CPU cycles for instances.

4.4.2 Swift

Authentication and Authorization The swift object store is designed to be used by several costumers. Therefore, its access is secured by ACL authentication mechanisms. The Swift authentication is called *swauth* and is now a separate project[3]. Some parts of swauth are based on different components of the original Rackspace architecture – its ancestor.

The authentication is not only a different project, but can also be a subsystem of Swift or a complete stand-alone system, external to Swift. It communicates with Swift via WSGI. Once the user has been authenticated, he/she holds an *auth token* that he/she can pass along with every request to Swift. Then, these requests internally call the swauth subsystem (via WSGI) to check, whether the access is actually authorized according to a particular ACL (see Figure 4.4). This token does not change over the period of different requests, but it expires after a certain period of time.

Availability As Swift does not use a traditional file system, rather than an object store with buckets similar to Amazon's S3, it can be distributed and replicated accross several nodes and access may occurr simultaneously. This concept enables almost linear scalability with access due to the redundantly distributed data across several nodes. Therefore, the availability profits as well, as there is no longer a single point of failure and built-in replication provides higher data redundancy.

[3] https://github.com/gholt/swauth

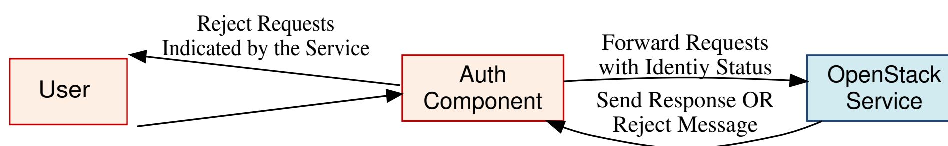

Figure 4.4: The User authenticates itself to the *Auth Component*. On success, the actual OpenStack service can be reached, which – if necessary – asks the *Auth Component* to check permissions.

4.4.3 Glance

Authentication A glance VM image always belongs to an owner. The default authentication of glance can be changed to use the new *Keystone* authentication component (see *"Authentication Framework"* in section 4.4).

However, it is possible to share an image with others, either by using groups in Keystone or by making the images public. This imposes the same risks of publicly shared images as described in section 3.1.

4.5 Security Shortcomings

4.5.1 Nova

Incomplete Protection of Messages in Service Communication The message queue that enables services within an OpenStack cloud to communicate with each other is a crucial component. We are assuming a curious or malicious attacker who has access to the network that is used by the message queue for transportation. The attacker can read and modify traffic on the network.

OpenStack uses RabbitMQ as the message queue implementation, which does not support encryption of individual messages. However, RabbitMQ supports secure channels between the message queue server and clients using SSL/TLS. At the time of this writing, OpenStack does not leverage the SSL capabilities of RabbitMQ in their default configuration, probably due to the overhead of setting up the corresponding public-key infrastructure. Therefore, our assumed attacker can read and modify exchanged messages.

A potential attack to exemplify the problem is that in the case a user requests a new virtual machine, the attacker could either tamper the request with lower resource requirements, in order to degrade the performance of that VM, or increase the resource requirements for higher costs on the user's side.

Incomplete Authentication in Service Communication Besides incomplete protection of exchanged messages using the messaging queue, authentication of the communication participants is also incomplete. We are assuming an attacker that can connect to the message queue service.

RabbitMQ provides authentication of clients using *Simple Authentication and Security Layer* (SASL). The analyzed version of OpenStack uses the same username and password for all messaging queue clients, with *guest* as the default username and password. The currently supported authentication method is plaintext only, where the password is sent in plaintext.

Another limitation of this approach is that only access to the message queue is controlled. The authentication of communication participants is not ensured, e.g., by signing exchanged messages. As a possible attack, this would allow an attacker to inject a false user request that terminates all the virtual machines of a particular user.

Potential of XML Signature Wrapping Attacks against Cloud API The Cloud API is a crucial component in an OpenStack cloud, both from an operational point of view as well as a security one. All requests from cloud users are channeled through the API, which also renders it a large external attack surface. In the previous attack scenarios, the attacker required access to the internal network of the cloud provider. In this scenario, we can assume an attacker who is a regular cloud user.

The authors of [SHJ+11] show attacks against the management interfaces of two other common cloud platforms, namely, Amazon EC2 and Eucalyptus. They are able to employ *XML Signature Wrapping attacks*, in order to circumvent the authorization mechanisms of the management interface. Therefore, they are able to issue management commands while impersonated as a victim user and taking full control of a victim's account.

We conjecture that OpenStack is also vulnerable to the same class of attacks. OpenStack provides a web service interface and even implements a compatible API of Amazon EC2.

Virtual Machine Escape Attacks Virtual machines contain the workloads of the cloud users and they are running on top of a complex stack of software that provides the resource virtualization and management. Complex software is prone to software vulnerabilities that an attacker could exploit to gain privileged access. In many cases, infrastructure clouds are offered as a public service, therefore an attacker could easily launch his malicious virtual machines that tries to attack the underlying virtualization layer. In other cases, an attacker could compromise a virtual machine of a legitimate user and continues the attack against the virtualization software.

Escaping a virtual machine means that an attacker gets access beyond the confined execution environment of the VM. In the worst case, the attacker obtains high privileged access to the virtualization platform and can access the virtual machines of other users, e.g., to disclose or tamper with their data. A vulnerability allowing such an attack was disclosed in CVE-2011-1751[4].

The risk of VM escape attacks is amplified in OpenStack, because they do not recommend to use SELinux or similar security mechanisms, that confine the hypervisor and management software. If SELinux is in place, individual virtual machines are confined in minimal privileged domains which limit the possibilities of a successful attacker.

Missing Authorization for Virtual Machine Management The management of virtual machines, e.g., starting a new VM or migrating a VM to another host, is realized using the virtualization management library *libvirt*. In particular for VM migration, the migration between the source and destination host is performed on a peer to peer basis, that means, the migration is initiated and performed between these two hosts. Therefore, the initiation phase requires access to libvirt on the peer which is realized using the network service *libvirtd*. In OpenStack's current implementation, there exists no authorization for issuing libvirt commands on another host in the cloud.

[4] https://bugzilla.redhat.com/show_bug.cgi?id=699773

If we assume an attacker that has access to the management network of the compute nodes, manipulations of virtual machines running on these hosts could easily be done. For example, virtual machines could be terminated or migrated to insecure hosts.

Disclosure and Tampering during Virtual Machine Migration Live migration of virtual machines is an essential feature for highly available cloud services, where virtual machines can be migrated to other servers in case of maintenance or performance problems. However, in the current form, the virtual machine memory and state is not protected while transported over the network during live migration.

We assume an attacker that has access to the network used for migration. The attack allows disclosure and tampering of the VM's memory and state, as demonstrated in [OCJ08]. For example, they demonstrated the injection of a backdoor in ssh while a virtual machine is migrated.

KVM, a virtualization technology used by OpenStack, supports secure migration in combination with external tools such as GPG or SSH [5], but they are not used in OpenStack current implementation.

Attack of Malicious Compute Node Administrators Cloud administrator typically have high privileged access to the cloud servers hosting cloud users' virtual machines. In the case of a malicious inside attacker, i.e., an administrator, their high privileges allow them to access or tamper potential sensitive data stored or processed in a virtual machine. Therefore the confidentiality and integrity of cloud users' data is at risk. Furthermore, by having local high-privileged access, the malicious administrator can terminate virtual machines. Similar attacks of such malicious insiders were demonstrated by [RC11].

In OpenStack, virtual machines are not protected against access from high privileged users on compute nodes. It is the opposite of the previously outlined attack of virtual machine escapes, where malicious cloud users try to attack the cloud platform. In this attack, malicious administrators try to attack the cloud users.

Storage Volume Disclosure and Tampering Cloud users can extend the storage capacities of their virtual machines by attaching external storage volumes, which are similar to Storage Area Networks. The storage volumes are provided by storage nodes over the network to compute nodes.

In the case that an attacker has access to the network that is used for the storage volume traffic, or has access to the storage nodes, the confidentiality and integrity of the cloud users' data on these volumes is at risk. In the current form of OpenStack, the infrastructure does not provide any encryption or integrity protection for these volumes. However, cloud users could use an encryption layer in their virtual machines that encrypts all the data before written to the external storage volume.

Missing Authentication and Authorization for Attaching Storage Volumes As previously outlined, external storage volumes are exported from storage nodes and attached to compute nodes. OpenStack does not provide any form of authentication and authorization for the attachment of storage volumes.

[5] http://www.linux-kvm.org/page/Migration

In the case that an attacker has network access to a storage node, the exported volumes can be attached to the attackers machine. Potentially, the cloud users' data stored on these volumes is at risk if not otherwise protected.

4.5.2 Swift

Accountability and Authentication Swift does not support accountability in the sense of proven authenticity provided by e.g. digital signatures. The only procedure that guarantees authenticity is to trust the authentication mechanism (swauth) that allows a user to write data to a certain bucket and to trust the isolation of Swift's object store.

A bad example of failed authentication mechanism and its consequences was DropBox, which successfully verified *any* password for four hours[6].

Confidentiality and Integrity The Swift architecture provides no means to abstract from classical cryphtographic means to provide confidentiality and integrity. Encryption and message authentication codes, respectively, have to be applied by the user of Swift, if needed. There is no provided client-side authentication and/or encryption module. A server-side encryption (as provided by mozy[7]) does not provide confidentiality against the cloud provider, as the provider has access to those shared keys. However, the current Swift API is also compatible with Amazon's S3 API, which does not support encryption either, but there is a tool to conveniently encrypt S3 buckets locally[8].

Retention and Object Versioning Currently, Swift does not offer data retention (i.e., data object versioning or record management) to meet legal, governmental, or business requirements on data archival. Operations on Swift objects (e.g., write or delete) overwrite the existing object without storing a history of the changes or a new revision such that the applied operations can be tracked.

4.5.3 Glance

The Glance usage model explicitly allows the sharing and publishing of VM images with other users. As such, this model is subject to the same threats that have been identified in Chapter 3 and [WZA+09, ASM09] and require ideally infrastructure supported solutions: users of shared/public images should be able to ensure (or validate) the trustworthiness of the employed images; publishers of images should have the means to remove any confidential or security sensitive data from their images in an efficient manner. Currently, the OpenStack project misses these features.

[6] http://hardware.slashdot.org/story/11/06/21/045228/Dropbox-Password-Goof-Let-Any-Pas

[7] http://docs.mozy.com/docs/en/user-home-mac/guide/tasks/config_encrypt_c.html

[8] http://aws.amazon.com/articles/2850096021478074

Chapter 5

Clouds' Infrastructure Taxonomy, Properties, and Security Challenges

Chapter Authors:
Imad Abbadi (OXFD)

5.1 Introduction

Cloud computing is relatively a new term in IT (started in 2006 with Amazon EC2[Ama10a]), which has emerged from commercial requirements and applications [JNL10]. Cloud supports three main types: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [MG09b]. The Cloud infrastructure is complex and heterogeneous in nature; various Cloud components provided by different vendors need to communicate in an organized and well managed way. Cloud infrastructure management is mainly provided by internal employees and contractors. There are different tools, which help employees to manage Cloud infrastructure, which require human intervention for supporting the infrastructure. One of the main Cloud potential features is the provision of fully automated services (we refer to such services as self-managed services), which provide Cloud infrastructure with exceptional capabilities enabling it to automatically manage the infrastructure and take appropriate actions on emergencies [AFG+09, JNL10]. Achieving self-managed services is not an easy task considering Clouds' infrastructure complexity and heterogeneity. This would require careful understanding of how experts in the domain manage the infrastructure, and also requires analyzing Clouds' infrastructure management tools and components interaction.

Cloud has different models, e.g. private, community, and public Cloud [JNL10]. Public Cloud (e.g. Amazon EC2 [Ama10a]), as its name indicates, could be used by anyone without having a prior relation with the Cloud provider. On the other hand, private Cloud is mainly used by a specific organization, and a community Cloud is used by collaborating organizations who share a common mission, goals, etc. Public Cloud model has much more customers in comparison with other models. Therefore, public Cloud services should be automated to hide the complexity of the infrastructure, and to increase users' service availability and reliability. Fully automated management services are not yet available, at the time of writing, for many Cloud services, which are required by different types of applications including but not limited to critical applications [Abb11c]. Such lack of automated management for many services forces public Cloud provider to mainly support basic functions which can be automated at the virtual layer. These cover the needs of casual users, small businesses, and uncritical applications. Other Cloud models, on the second hand, which have limited number of users support all possible

wide range of services. Such services are customized for the need of a group of organizations, and require much more human intervention in comparison with the ones provided by public Cloud. One of the objectives of this chapter is to identify the services which require automation. Automating such services are important for potential Cloud.

5.1.1 Cloud Evolution

Prior to the virtualization era, customers used to provide their application requirements to enterprise architects. Enterprise architects used to provide an architecture which is typically designed to a specific customer application needs and requirements. This has caused huge waste of resources, e.g. computational resources and power consumption. Virtualization technology, which is the foundation of the Cloud infrastructure, brings tremendous advantages in terms of consolidating resources; however, it is also associated with other problems, e.g. security and privacy problems [Abb11c]. Virtualization era changes the mentality of enterprise architects, as the relation between users and their physical resources are no longer one-to-one. This raises a big challenge of how such a consolidated architecture can satisfy users' dynamic requirements and unique application nature. Enterprise architects addressed this by studying the environment they inherit prior to virtualization era, and they found different architectures have many similarities. Such similarities enable enterprise architects to split the infrastructure into groups. Each set of groups can be architected and associated with certain properties, which enable such a group to address common requirements of certain categories of applications. For example, a group can be allocated for applications: i) that can tolerate single point of failures; ii) that require full resilience with no single point of failure; iii) that are highly computational; iv) a group for archiving systems, etc.

The second challenging question is how such grouping, which is associated with almost static properties, can fit with the dynamic users' requirements and their application nature. Enterprise architects realize that virtualization can be fine-tuned and architected to support the dynamic properties, which are not already provided by the physical group static properties. In other words, the combination of physical properties and the virtual layer dynamism are used to support customers' expectations. Enterprise architects found that using virtual layer can even provide many automated features that cannot be easily provided at the physical layer. Automated management features at the virtual layer are the main key factor behind the evolution of Cloud computing. However, we are still at an early stage for providing fully automated services for many limitations, which we partially discuss in [Abb11b, Abb11c]. The lack of full automation restricts public Cloud providers from supporting many features already provided, manually, by community and private Cloud providers, as discussed in the chapter.

5.1.2 Related Work

In this chapter we continue our previous work in [Abb11c] which discusses the misconceptions about Cloud computing, introduces Cloud layering concept, and derives the main security challenges in the Cloud. In this chapter we start by proposing a Cloud taxonomy, and then derive management services and factors affecting their actions. The factors include both infrastructure properties and user properties. We have previously defined self-managed services and the security challenges for providing such services in an extended abstract [Abb11b]; however, the foundations of our previous work are clarified in this chapter.

There is few related work, which analyzes Cloud environment (see, for example, [Clo10a, YBS08]). These mainly focus on analyzing Cloud properties, benefits, and services from user

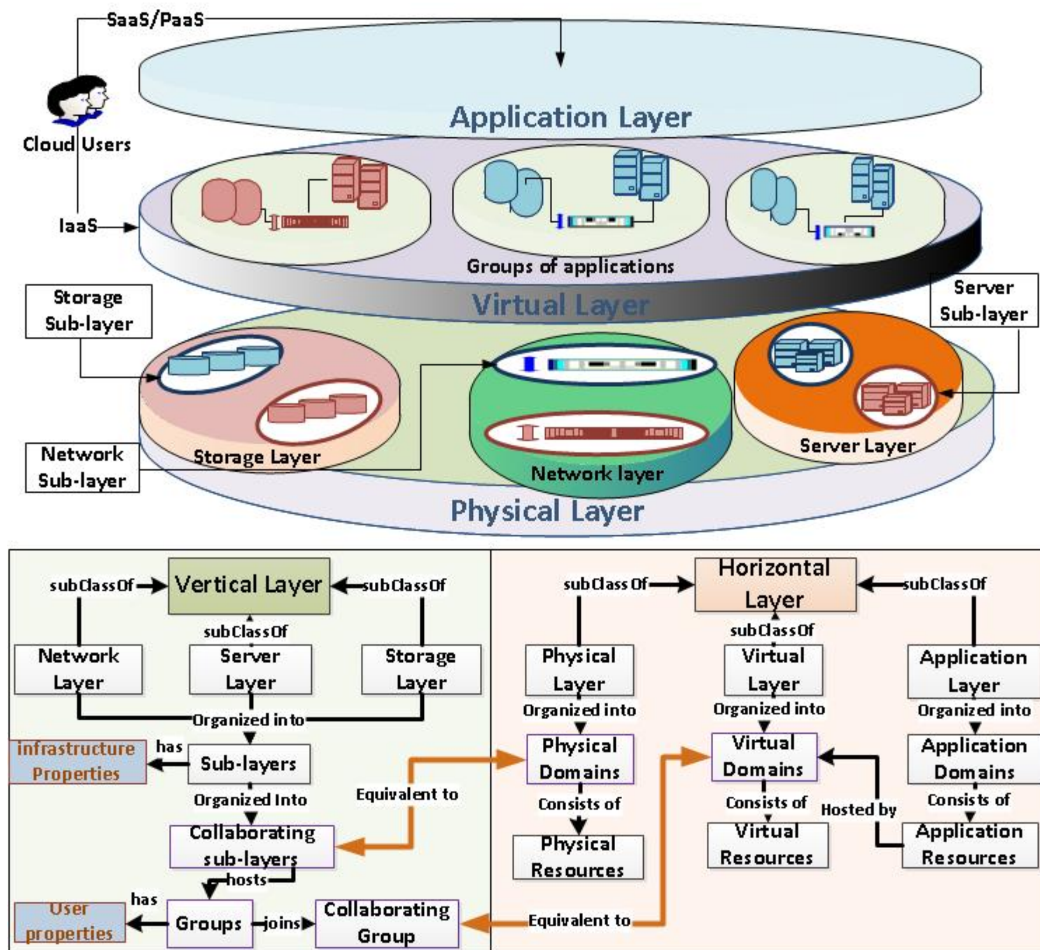

Figure 5.1: Cloud Taxonomy: 3-D View

perspective. However, they do not discuss the Cloud infrastructure taxonomy, do not discuss management services and the properties they require when managing Cloud infrastructure. Our proposed taxonomy does not contradict or even replace previously proposed ones which mainly focus on different angle from ours. It is rather the opposite, as our taxonomy completes the picture of such work which considers the physical layer as a black-box and also does not discuss the management of Cloud infrastructure. Autonomic computing [IBM01] is not related to our work, as it is mainly concerned about management of physical resources.

5.2 Taxonomy of the Cloud

In this section we propose a taxonomy of Cloud focusing on the relationships and interactions amongst Cloud components. In section 5.3.1 we illustrate the taxonomy in the context of a scenario. We use the taxonomy to derive Cloud infrastructural properties, which are one of the key factors when providing automated management services.

5.2.1 Cloud Infrastructure Taxonomy

A Cloud infrastructure is analogous to a 3-D cylinder, which can be sliced horizontally and/or vertically (see Figure 5.1). We refer to each slice using the keyword "layer". A layer represents

Cloud's components that share common characteristics. The layering concept helps in understanding the relations and interactions amongst Cloud components. We use the nature of the component (i.e. physical, virtual, or application) as the key characteristic for horizontal slicing of the Cloud. For vertical slicing, on the other hand, we use the function of the component (i.e. server, network, or storage) as the key characteristic for vertical slicing.

As illustrated in Figure 5.1, the Vertical Layer consists of three layers: Storage Layer, Server Layer, and Network Layer. Each layer is organized into sub-layers; i.e. we have network sub-layer, storage sub-layer, and server sub-layer. Each sub-layer provides specific properties to serve the needs of the wide range of Cloud user requirements. Server, network and storage sub-layers are organized into multiple *collaborating sub-layers*. Sub-layers within each 'collaborating sub-layer' and their components are carefully selected, interconnected, and even physically positioned to support the overall collaborating sub-layer properties.

Virtual resources are then created and grouped based on user's application requirements. Multiple related *groups* join a *collaborating group* based on user requirements and application nature (e.g. dependency amongst application components). Sub-layers and groups are associated with properties and policies, which are of most importance for managing the infrastructure (and especially for the provision of automated self-managed services). Sub-layers' properties and policies are infrastructure related, while group properties and policies are related to user's application requirements. Each group is hosted at a collaborating sub-layer that has physical properties that best match with user properties.

Figure 5.1 also illustrates the relation between Horizontal and Vertical layers. We identify a *Horizontal Layer* to be the parent of physical, virtual and application layers. Each *Horizontal Layer* contains *Domains*, i.e. we have Physical Domains, Virtual Domains, and Application Domains. A *Domain* represents related resources which enforce a *Domain* defined policy. *Physical Domains* are related to Cloud infrastructure and are, naturally, associated with infrastructure properties and policies. A *Physical Domain* in the Horizontal Layer is equivalent to a Collaborating Sub-Layer (in Vertical Layer terms).

An *Application Domain* is composed of the components of a single application. A *Virtual Domain* is then created to serve the needs of an *Application Domain*. Each *Virtual Domain* is composed of groups of virtual resources. A group would typically run and manage a specific component within an *Application Domain*. Each *Virtual Domain* group is associated with user properties which are related to the application component to be served by the group. Such properties help in directing the management services when providing automated self-managed services of the group. For example, such user properties help management services to decide on i) minimum and maximum resources allocated to a virtual machine within a group (Vertical Scalability), ii) minimum and maximum number of virtual machines that can be allocated and deallocated within each group based on load/incidents (Horizontal Scalability), iii) deciding on the right *Physical Domain* that can serve the needs of the application, and iv) helps management services to react based on user requirements during incidents. A *Virtual Domain* in Horizontal Layer is equivalent to Collaborating Group (in Vertical Layer terms).

In this chapter we mainly focus on the vertical slicing of the physical and the virtual layers, as our interest is in IaaS Cloud type. Also, our discussion next follow the vertical slicing as we are mainly interested in deriving infrastructure properties.

Network Layer

A network layer is the backbone that provides communication medium between Cloud's components. The communication medium can be either public or private. By public we mean

communication occurs over the Cloud's local or wide area network. Private, on the other hand, means communication occurs in a physically dedicated network, which is isolated from the public network. Such a private network is especially setup between a set of components to perform a specific function; e.g. (a.) connecting a server to dedicated storage, as in the case of Storage Area Network (SAN) [Wik10b], and (b.) software clustering as the case in Real Application Cluster (RAC) requires member servers in RAC to have a private network [Ora11a].

From an abstract level the communication amongst Cloud components is organized within defined boundaries that follow a process workflow. We refer to such communication as horizontal and vertical communication, which are described as follows (see Figure 5.2).

Horizontal Communication — In this type Cloud entities communicate as peers either inside a sub-layer or across sub-layers. This type of communication does not span outside layer boundaries. We now discuss what we mean by horizontal communication in the following examples: (a.) horizontal communication can be realized when storage systems are self-replicated in such a way one storage entity regularly copies changes of its physical blocks to a standby storage entity; and (c.) when Virtual Machines (VMs) within a sub-layer collaborate in a RAC [Ora11a] and need to exchange messages to synchronize shared memory (e.g. memory fusion [Ora11a]) is also a form of horizontal communication between VMs.

Vertical Communication — In this type Cloud entities communicate with other Cloud entities in the same or different layer following a process workflow in either up-down or down-up directions. This would typically work as follows: an upper sub-layer component runs a process which generates sub-processes that should run at a lower sub-layer, following a process workflow. The lower sub-layer could be in the same or different layer of the upper sub-layer. The lower sub-layer executes the sub-processes and then sends the result back to the upper sub-layer. We provide the following example: a multi-tier application in which the front-end in the Cloud represents a load balancing component that receives users' requests and distributes them across the middle-tier sub-layer. The middle-tier sub-layer, which runs the application logic, processes the request and generates sub-requests that send them to the backend sub-layer. The backend sub-layer, which runs a DB instance, processes the sub-request and then generates a sub-sub-request and sends them to the storage sub-layer. These steps represent an up-down communication channel. Each layer in turn sends its response back in the opposite direction, which represents the down-up communication channel.

There are many other important network properties, which we do not discuss in this section for space limitations, e.g. network speed between components, network nature, any restrictions affecting information flow as in the case of a firewall stopping certain type of traffic, network topology, etc.

Storage Layer

A storage layer is composed of sub-layers, which consist of storage components. A storage component is the basic component[1] that stores Cloud data and/or provides file system services. Storage could be either local storage or network storage. Local storage is connected directly to server(s) via a private network (e.g. SAN), while network storage means servers are connected

[1] We mean by basic component an integrated component (e.g. EMC storage products [EMC11]) and not a simple hard-disk or physical block

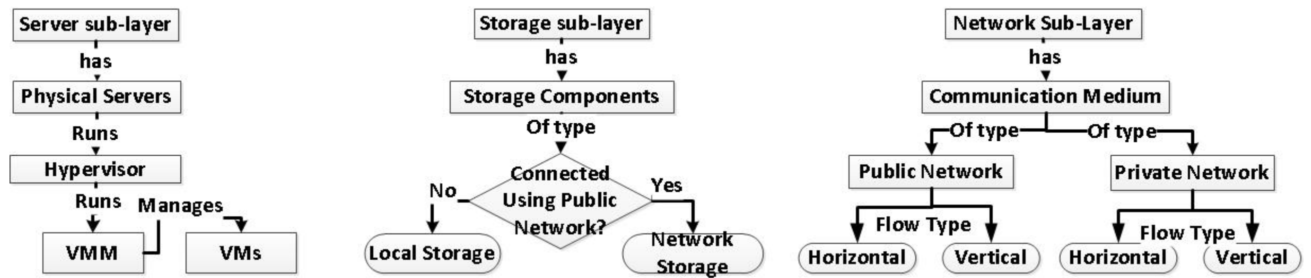

Figure 5.2: Conceptual Models of Cloud Layers

to the storage over public network (e.g. Network-Attached Storage (NAS) [Wik10a]). Network storage can provide Cloud users with storage as a service (e.g. Amazon S3 [Ama10b]); however, local storage does not communicate directly with users and does require a *Server Layer*. Each type of storage (i.e. local and network storage) has many different categories, which is outside the scope of this chapter to discuss. An Enterprise architect decides on storage specifications by considering many factors (e.g. purpose of storage — file system or block storage —, storage usage — e.g. cluster of servers). Storage components communicate horizontally, for example, when replicating data at physical level, i.e. storage-to-storage. The storage layer/sub-layer communicates vertically with other layers. Part of Figure 5.2 provides a conceptual model of Storage Layer.

There are many important properties about storage layer, which include: size, speed, protection measures (e.g. hardware raid), reliability, connectivity with the servers and its speed (private or public network), physical distance from attached servers (for private network), etc. Again we do not aim to discuss these for space limitations (we outline some of these in section 5.3.1).

Server Layer

The server layer is composed of multiple sub-layers. Each sub-layer is composed of a set of physical servers. Physical Servers provide computational resources to Cloud users (e.g. CPU, memory, network, and storage). Each server hardware resources are managed by a hypervisor (a minimized operating system providing the minimum components, which enable the hypervisor to virtualize hardware resources to guest operating systems[MMH08]). Part of Figure 5.2 provides a conceptual model of Server Layer.

The hypervisor runs (or sometimes the same as) the Virtual Machine Manager (VMM). The VMM manages Virtual Machines (VMs) running on the physical server [MLQ+10a, MMH08]. A VM provides an abstraction of CPU, memory, network and storage resources to Cloud users in such a way a VM appears to a user as an independent physical machine. Each VM runs its own Operating System (OS), which is referred to as guest OS. The guest OS runs the VM specific applications. VMs running in the same physical platform are independent, share the platform resources in a controlled manner, and they should not be aware of each other; i.e. a VM can be shutdown, restarted, cloned, and migrated without affecting other VMs running on the same physical platform.

5.2.2 Virtual Control Center

In this section we identify the component that can take the role of providing automated management services. Cloud infrastructure is composed of enormous components, which are not easy to be managed manually. There are different tools, which help Cloud employees to manage a Cloud infrastructure. These cover virtual resource management, physical resource management, network management, server management, etc. In this chapter we are mainly concerned about virtual resource management tools, which manage virtual resources and their interaction with physical resources. There are many tools for managing virtual resources, which are provided by different manufacturers (e.g. VMWare tool is referred to as vCenter [VMw10], Microsoft tool is referred to as System Center [Mic10]). Many open source tools have also been recently developed (e.g. OpenStack [Ope10b] and OpenNebula [Ope10a]), which support additional services. In this chapter, for convenience, we refer to such tools, which are used to manage virtual resources, as Virtual Control Centre (VCC). In our previous work ([Abb11c]) we have outlined VCC which helped us to derive cloud unique security challenges. In this chapter we discuss it considering the provided taxonomy and identify the factors that affect its operation.

VCC establishes communication channels with physical servers to manage Cloud's Virtual Machines (VMs). VCC establishes such channels by communicating with the VMM running on each server. VCC and VMM regularly exchange heartbeat signals ensuring they are up and running. VMM regularly communicates VMs related status (failure, shutdown, etc) to VCC enabling the latter to communicate the status to system administrators. Such management helps in maintaining the agreed Service Level Agreement (SLAs) and Quality of Service (QoS) with customers. In addition, and probably most importantly, VCC provides system administrators with easy to use tools to manage virtual resources across the Cloud infrastructure. This is very important considering the Cloud complex and heterogeneous nature. For example, if a physical machine fails (e.g. due to hardware failure) then to where should VMs running on top of the failed physical machine move. Also, once the failed physical machine is recovered should VMs return back to their original hosting server or should they stay at the guest hosting server. Such examples are managed by VCC based on policies predefined by enterprise architects but managed by system administrators using VCC.

5.2.3 Factors Affecting Management Services

We believe that VCC will play a major role in managing self-managed services. We now identify the factors, which would affect decisions made by self-managed services.

Infrastructure Properties (Static Properties) — As we discussed earlier, Clouds' physical infrastructures are very well organized and managed by multiple parties, e.g. enterprise architects, system administrators, security administrators. These parties build the infrastructure to provide certain services and are aware of Cloud taxonomy, which we describe it early in this section. Therefore, they define the physical infrastructure properties for each infrastructural component, sub-layers, and layer. Providing such properties to VCC is a foundation step for supporting automated management services.

User Properties (Dynamic Properties) — A Cloud user interacts with the Cloud provider via a Cloud webpage and supplied APIs. This enables users to define *user properties*, which should cover the following for potential Cloud:

a.) Technical Requirements — IaaS Cloud users would typically be organizations, which have expertise to provide enough information about their technical requirements in terms

of VMs, storage, and network requirements. For example, they provide the properties of applications to be hosted on VMs, e.g. DBMS instances that require high availability with no single point of failure, middle-tier web servers that can tolerate failures, highly computational application, etc. This enables the Cloud provider to identify the best infrastructural resources that are fit for user requirements.

b.) Service Level Agreement (SLA) Requirements — These specify quality control factors and other legal and operational issues on user services; for example, define system availability, reliability, scalability (in upper/lower bound limits), and performance metrics.

c.) User-Centric Security and Privacy Requirements — Examples of these include (i.) users need stringent assurance that their data is not being abused or leaked; (ii.) users need to be assured that Cloud provider properly isolate VMs that runs in the same physical platform from each other (i.e. multi-tenant architecture[RTSS09]); and (iii.) users need to identify the location of data distribution and processing (which could be for legal reasons). Current Cloud providers have full control over all hosted services in their infrastructure; e.g. the Cloud provider controls who can access VMs (e.g. internal Cloud employees, contractors, etc) and where user data can be hosted (e.g. server type and location). The user has very limited control over the deployment of his services, has no control over the exact location of the provided services, and has no option but to trust the Cloud provider to uphold the guarantees provided in SLA.

Infrastructure Policy — Policies should be defined by Cloud authorized employees and associated with layers and sub-layers to control the behaviours of self-managed services.

Changes and Incidents — These represent changes in: user properties (e.g. security/privacy settings), infrastructure properties (e.g. components reliability, components distribution across the infrastructure, redundancy type), infrastructure policy, and other changes (increase/decrease system load, component failure, network failure, etc).

Management services should automatically manage the Cloud environment by finding the best match of user properties with infrastructure properties that considers infrastructure policy. For example, a sub-layer would be associated with a set of infrastructure properties defining many important factors related to the sub-layer itself, and how it is related to other sub-layers. Also, groups hosted at each sub-layer are associated with users' properties. These enable automated management services to take proper actions on emergencies; as such, services would be provided with architectural factors and users requirements.

5.3 Deriving Self-Managed Services

The first subsection provides a real life scenario for an application that is currently deployed at a community Cloud provider. In this we map the scenario at the provided taxonomy. In the second subsection we use the scenario to identify and motivate the need for automated self-managed services. Chapter 8 provides detailed definition and discussion of the services identified in this section.

5.3.1 Multi-Tier Application Scenario at the Cloud

We have architected and deployed the scenario which is provided in this section for a production environment supporting an editorial workflow. The editorial workflow depends on a weather

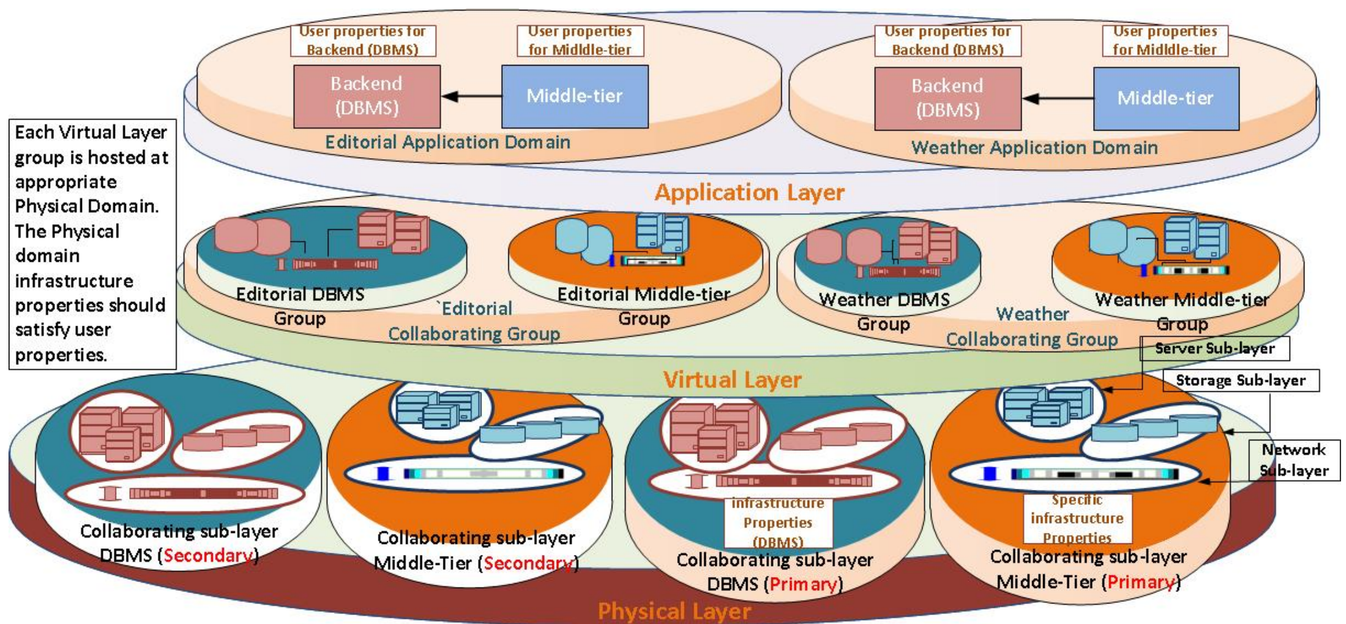

Figure 5.3: Cloud Taxonomy — Multi-Tier architecture at Community Cloud Provider

forecast application. For simplicity we assume both editorial and weather applications have similar architectural requirements.

The system is architected as a multi-tier application, which is deployed across community Cloud infrastructure (primary and secondary locations), to achieve the following user properties: high system availability and reliability, disaster recovery (DR) to support business continuity, high resilience with no single point of failure, transactions type are more write than read, system scalability (i.e. minimum/maximum resources that can be allocated/deallocated when load increase/decrease), and security properties. We next provide a simplified architecture based on user properties and the discussed Cloud taxonomy (Figure 5.3 illustrates the overall architecture).

Application Middle-tier Groups — These virtual layer groups run application business logic functions, which provide services to end-users. We require two groups as illustrated in Figure 5.3: the first we refer to as weather middle-tier group which runs weather middle-tier application component, and the second we refer to as editorial middle-tier group which runs editorial middle-tier application component. Both groups should be hosted using appropriate collaborating sub-layer, as discussed latter. Also, the number of VMs and their specifications within each group would depend on expected load and user requirements, which we do not discuss in this example for simplicity. But each group should at least have two VMs, as the user requires no single point of failure. Having one VM means if it fails the system will be down while the VM gets restarted.

Database Management System (DBMS) Groups — The DBMS groups at virtual layer manage the application data (e.g. storage, retrieval, and indexing). We require two groups as illustrated in Figure 5.3: the first we refer to as weather DBMS group which hosts weather DBMS and the second we refer to as editorial DBMS group which hosts editorial DBMS. Both groups should be hosted using appropriate collaborating sub-layer, as discussed latter. Also, the number of VMs and their specifications within each group would depend on expected load and user requirements, which we do not provide in this example for simplicity. But, as in the case of

application middle-tier groups, each DBMS group should at least have two VMs to support no single point of failure.

Collaborating Sub-Layers — Each collaborating sub-layer at the physical layer is composed of three sub-layers: storage sub-layer, network sub-layer, and server sub-layer. The server sub-layer has special properties enabling it to host the indicated type of application. The storage and network sub-layers are associated with special properties enabling them to collaborate to support server sub-layer properties, which can address wide range of common user requirements which are related to a specific category of application (e.g. DBMS with no single point of failure). The system architect should provide a resilient architecture based on both user supplied requirements and Cloud's infrastructure properties. Figure 5.3 provides four collaborating sub-layers: (a.) collaborating sub-layer middle-tier (primary) which has properties enabling it to host middle-tier application groups and is physically located at Cloud primary location; (b.) collaborating sub-layer middle-tier (secondary) which has properties enabling it to host middle-tier application groups and is physically located at Cloud secondary location; (c.) collaborating sub-layer DBMS (primary) which has properties enabling it to host DBMS groups and is physically located at Cloud primary location; and (d.) collaborating sub-layer DBMS (secondary) which has properties enabling it to host DBMS groups and is physically located at Cloud secondary location. Collaborating groups at the primary location host groups at the primary location, and act as a backup (i.e. DR) for groups located at the secondary location. Similarly, Collaborating groups at the secondary location host groups at the secondary location, and act as a backup for groups located at the primary location. We now discuss some of the properties at individual sub-layers.

Storage layer — The system architect should use storage sub-layers that satisfy user requirements. For example, one of the user requirements indicates that the system activity is more write than read. For performance reasons this would require RAID 1+0 for DBMS sub-layers rather than RAID 5. In addition, the user requires no single point of failure, which implies integrated storage component should be fully redundant from inside and outside (e.g. dual communication channels, multiple processor cards). It also implies replicating data from the community Cloud primary location to its secondary location. Replicating data can be done at different levels: (a.) storage sub-layers or (b.) DBMS server sub-layers.

Server layer — The scenario requires four groups, as discussed above. The system architect should decide on server sub-layers that can host each group. The systems architect should also associate with each group a set of properties enabling it to satisfy consumer requirements. Understanding the nature of the hosted application enables the system architect to even provide enhanced features in terms of using the right hardware configuration (i.e. server sub-layer) that best suits the generic nature of the application; e.g. DBMS application, highly computation systems, etc.

In our scenario, the system architect should: (a.) associate with all groups a dependency property that requires all groups to always run in the same physical location at one time and on emergencies all groups should fail-over to predefined sub-layers located at DR location (such a condition ensures that all dependent components run in the same location; e.g. avoid the case where a DBMS group is hosted in a different location than its corresponding middle-tier group); (b.) host editorial and weather DBMS groups at server sub-layer which has properties of hosting DBMS with no single point of failure; and (c.) host editorial and weather middle-tier groups at server sub-layer which has properties of hosting middle-tier applications. It is beyond the scope of this chapter to discuss architectural reasons beyond that, but of course all groups could be hosted using a single server sub-layer or multiple sub-layers. This is based on user properties and infrastructure properties.

Network Layer — The above sub-layer components (i.e. server sub-layer's components and storage sub-layer's components) must be connected using at least two network channels. Also, related server sub-layers and storage layers should be connected using redundant channel. For example, a DBMS server sub-layer should be connected using multiple channels to related storage sub-layer. In addition, the storage sub-layer itself should provide full resilience, which is outside the scope of this chapter to discuss.

5.3.2 Identifying Management Services

Current public Cloud providers do not support the kind of architecture provided in the scenario above, as it requires human intervention. In this section we aim to derive the main services, which are mostly (at the time of writing) provided by private and community Cloud internal employees. We also aim to show the importance of automating such services. Chapter 8 provides detailed definition and discussion of the services identified in this section. Public Cloud potential future, which is expected to host critical applications should be capable of automatically and without human intervention manage Cloud environment [MG09b].

The first two services we identify are **system architect** and **resilient** design. A Cloud provider should provide an automated application architecture (what we refer to as *system architect as a service*), which should result in a resilient design. It should also automatically deploy the resilient design (what we refer to as *resilience as a service*). As we described earlier the deployment of the architecture should consider infrastructure properties and user requirements. In our scenario a fundamental user requirement, which is especially required by critical applications, is providing a resilient architectural design with no single point of failure. Current Public Cloud providers only support very limited features in this direction in comparison with the ones supported by private and community Cloud providers. This is because fully automated management services do not exist and public Cloud providers can only support limited features that can be managed automatically.

The other important user expectation from a Cloud provider is to automatically adapt to failures, changes in user properties, and infrastructure properties and policies, without affecting user applications. This is what we refer to as **adaptability as a service**. This requirement is critical for potential Cloud infrastructure. For example, when users change their requirements, the virtual layer resources should automatically adapt to such changes, and when infrastructure physical resources get changed the virtual layer resources should also automatically adapt to such changes without compromising users requirements. All these changes should not compromise user requirements, security and privacy properties.

Elasticity is one of the Cloud essential properties. In peak periods the virtual layer resources should automatically scale up, and in off-peaks the resources should automatically scale down. Such scaling is based on the demand and customer pre-agreed SLA, and it should not compromise user requirements, security and privacy properties. We refer to this as **scalability as a service**. Public Cloud providers at the time of writing support vertical scalability, but do not provide horizontal scalability. The Cloud provider should also provide **availability as a service** which is related to utilizing all redundant resources. Also, a Cloud provider should provide **reliability as a service** which assures end-to-end service integrity.

The combination of above services would result in higher availability and reliability as properties. Full reliance on human beings require longer time to architect and deploy solutions, longer time to discover and resolve problems, also it is error prone, the provider is subject to insider threats by Cloud employees, and does not provide a reliable way for measuring the level of trust in Cloud operations. This raises the need of self-managed services that can automatically

and with minimal human intervention manage a Cloud infrastructure. Automated self-managed services provide Cloud computing with exceptional capabilities and new features. For example, scale per use, hiding the complexity of infrastructure, automated higher reliability, availability, scalability, dependability, and resilience that considers users' security and privacy requirements by design. Automated self-managed services should help in providing a trustworthy resilient Cloud computing, and should result in cost reduction. More details about these services can be found in our extended abstract [Abb11b].

5.4 Security Challenges

Current implementation of cloud infrastructure provides users with elastic virtual resources, which are built on physical resources. Cloud providers supply users with APIs enabling them automatically (i.e. without cloud system administrator intervention) acquire and deploy their own virtual resources [Ama10a]. Organizations are the main entities that are expected to be the main customer for IaaS cloud type, by either outsourcing their whole IT or part of their IT infrastructure on the cloud. In this section we provide a scenario that discusses the typical steps an organization would follow to outsource part of their IT infrastructure on the cloud, and then we use this scenario to discuss cloud security concerns.

5.4.1 Scenario

An organization would typically do the following when deciding to outsource a service on the cloud.

1. The organization must first decide on the application it wants to host at the cloud infrastructure. The application nature, organization policy, and legislation factors would play an important part in the decision. For example, organizations' policy makers might reject hosting applications, which process financial data for legislative reasons.

2. In IaaS the enterprise architecture team still need to architect the outsourced infrastructure based on the requirements of the application. This includes drafting the specifications of virtual machines, which might cover interconnection between virtual resources (e.g. a cluster of VMs on the back-end with multiple middle tear VMs), and initial expectations on lower and upper bound on each VM (processor, memory, storage, and networking requirements).

3. The organization establishes a Service Level Agreement (SLA) with the APIs that are supplied by the cloud provider to maintain the identified requirements and to provide an adequate Quality of Service (QoS).

4. The organization would need to communicate with the cloud provider supplied APIs to create virtual resources considering the system architect.

5. Cloud provider's system administrators manage the organizational outsourced infrastructure using VCC, which should be based on the agreed SLA. In turn the organization would pay cloud provider on a pay-per-use model.

5.4.2 Security Concerns

In this subsection we list some of the security concerns that are related to the cloud infrastructure and of interest to us (it is outside the scope of this chapter to provide an exhaustive security analysis of cloud infrastructure).

1. *Trustworthy Clouds Provenance* — Logging, auditing and historical data are of tremendous importance in a Cloud. This is especially the case as a Cloud is expected to support Internet-scale critical applications. Logging, auditing and historical data have different usage, e.g. pro-active service delivery (incidents monitoring and security monitoring), error investigation, billing, and forensic investigation. Almost all of a Cloud's resources generate this data in some way. At present, the only way that provenance is provided on a Cloud is through linking together log and audit data, collected from multiple resources, to provide the complete history of an event or result. As discussed, Cloud systems are composed of dynamically interlinked resources. This means that building a logical sequence of events to investigate an incident for any one application requires data from many sources. These include the application itself, all logs for possible virtual resources that the application could have used, and logs of all physical resources that virtual resources could have used. Administrators must then combine this data correctly by identifying all time intervals in which an application used a specific virtual resource, all possible time intervals in which these virtual resources used physical resources and then all relevant log files from all related resources. Collecting and combining data from these resources is not easy or practical considering the potential scale of Cloud systems. We discussed challenges of trustworthy clouds provenance in [AL11].

2. *Cloud Virtual Resource Management* — Considering cloud infrastructure complexity, without having reliable and secure resource management tools, which enforce predefined policies, any secure solution might be easily bypassed. As we discussed earlier we are mainly interested in VCC related tools. VCC interaction with VMMs deployed at servers raise the following security concerns.

   (a) How can VCC be assured that VMMs' execution environment is secure, trusted, and reliable to provide timely information about the status of VMs. Also how can VCC be assured that VMMs enforce organization policies.

   (b) How can VMMs be assured of VCC trustworthiness and its running execution environment when communicating messages across.

   (c) How can VMM and VCC be assured that their data is stored securely and only accessed when their execution status is trusted.

   (d) VMM and VCC need assurance about the identity of each other.

   (e) VCC, as a central management service, must provide availability, scalability, resilience, and reliability properties without compromising security and data consistency.

3. Insiders — Insiders are cloud provider related users, who misuse their privileges intentionally or accidentally. Insiders are granted authorized credentials by the cloud provider to access cloud infrastructure to perform their job functions. Examples of insiders include cloud internal employees, contractors, and third party suppliers. Insiders' threats could cause major impact on cloud customers' content confidentiality, integrity and availability. Insiders could also cause a major disruption to the cloud infrastructure, especially

when interfering with cloud self-managed services. We now list few examples of possible insider threats in cloud.

(a) Cloud insiders might cause sever pain and unbound consequences on the cloud infrastructure when attacking self-managed services, as discussed in point (4.). When addressing this point it is important to consider the complexity and diversity of technologies that are associated with self-managed services.

(b) A cloud system administrator can connect to the hypervisor for a server, which hosts sensitive application for a financial institution. From the hypervisor the system administrator can access the memory space of the VM that runs the sensitive application. This enables him to access sensitive data.

(c) A cloud system administrator using VCC can delete all VMs for an organization and invalidate backups.

(d) A Cloud hardware supplier can copy VM images of one organization into a USB stick and sell them to a competitor organization.

These sample examples show the importance of mitigating insider threats to have a successful cloud.

4. User-Centric Security — Current cloud providers have full control over all hosted services in their infrastructure; e.g. a cloud provider controls who can access VMs (e.g. internal cloud employees, contractors, etc) and where user data can be hosted (e.g. server type and location). The user has very limited control over the service deployment, has no control over the exact location of the provided services, and has no option but to trust the cloud provider to uphold the guarantees provided in the SLA. These raise several problems, which include the following.

   (a) Regulation issues, especially when moving and processing data across territorial boundaries.

   (b) Users privacy and security concerns. For example, users need stringent assurance that their data is not being abused or leaked.

   (c) How users can be assured that cloud provider properly isolate VMs that runs in the same physical platform (i.e. multi-tenant architecture[RTSS09]) from each other. For example, two VMs for competing organizations might share the same physical platform. Such organizations need the assurance that their applications and data cannot be intercepted by others.

5. Self-Managed Services — Cloud computing's major potential improvement is the addition of self-managed services that do not require human intervention. This is to provide fully automated services from start to end. As we are interested in IaaS our discussion is limited to cloud infrastructure self-managed services (availability, reliability, scalability, adaptability and resilience, which should consider user security and privacy by default). Providing such services require careful consideration and analysis not only because of their complexity but also for the following reasons.

   (a) As we discussed earlier in this section cloud infrastructure is conceptually based on several layers, each has internal sub-layers. These complex layers need to be considered when providing self-managed services. Securing such services require the

consideration of the heterogeneous and complex layering of Clouds. It should also take into consideration the internal/external horizontal and vertical communication channels.

More specifically, providing self automated services for a component requires: (a.) understanding in which sub-layer this component belongs, (b.) what are the properties of this sub-layer, and (c.) what are the relations between this sub-layer and other layers/sub-layers (i.e. what is the policy that governs this sub-layer interaction with other layers). For example, providing availability service for a component is not only concerning about deciding where to replicate the component or its content (e.g. same sub-layer, same location or at a different location) but it should also consider other factors, e.g. how such decision for a component availability might affect the overall service availability; where a service is composed of all end-to-end components that are required to provide the service. We are not aiming to dig deeper into these for space limitations, and we leave this very important subject for near future work; we just want to stress that this point is not as simple as it might seem to be.

(b) Cloud infrastructure mixed nature consists of different types of hardware/software technologies, which are provided by multiple, and most likely, competing vendors. Self-managed services require complex communications at different stages across various cloud entities (i.e. the horizontal and vertical communication). In turn, this means different technologies from competing vendors should establish standard interfaces for exchanging messages. These are not present at the time of writing, and even providing such services using components from the same vendor is very complex to setup, error prone, and raises unique security challenges in comparison with traditional systems.

(c) Cloud infrastructure is not hosted at a single data center that is located at a specific location; it is rather the opposite, as most likely it is distributed across distant data centers. This factor has a major impact on decisions being made by self-managed services for several reasons; for example, the distance and the communication medium between distant data centers will have an impact on data transfer speed. Automated services must consider this important factor and other related factors (e.g. data volume, data access mode, etc) when providing a service. For example, it is sometimes the right decision to provide redundant active/active resources across distant locations and in other cases its wiser to provide active/passive mode.

(d) Cloud-of-clouds is a term that is used to refer to the collaboration of multiple cloud providers to support having dependable cloud infrastructures; i.e. cloud providers collaborate to help each other in enhancing self-managed services as in the case of higher resilience, reliability, scalability, and dependability. For example, if a cloud provider has an emergency other cloud providers can temporarily provide their unoccupied resources to support customers eliminating service failures. Self-managed services must consider the existence of cloud-of-cloud, and it must also be designed to enforce cloud provider related policies when considering a decision to use other cloud resources, as this would have a major impact on security, practicality and legislation related issues.

Part II

Building Blocks for Trusted Cloud Architecture

Chapter 6

Log Service

Chapter Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

The main focus of this component is to log and track events originating at the *infrastructure* layer, such as creation, destruction or migration of a Virtual Machine (VM) or allocation and deletion of a bucket of storage.

This log service is mainly based on the scheme for secure logging proposed by Schneier and Kelsey in [SK99]. Such a scheme provides most of the security features required by a logging system, but it lacks the protection against particular attacks, namely *truncation attack* and *delayed deletion*. Despite this, it has been used as a foundation by most of the subsequent secure logging systems, which use the same structure with different cryptographic primitives (mainly public key cryptography instead of symmetric key). For this reasons and to keep the discussion and design as simple as possible, we base the log service on this scheme, but we claim it should be possible to employ the other secure logging schemes that were designed using Schneier-Kelsey's scheme as basis.

6.1 Background

In this section we introduce the background necessary to understand the functioning of the log service, namely the scheme for secure logging which will be used as basis for the log service and some details about the Trusted Computing technology that will be used to enhance the trust in the logs created through this log service. Note that for what concerns Trusted Computing, only the necessary background is given here, while a broader discussion can be found in Section 11.

6.1.1 Schneier-Kelsey scheme.

Schneier and Kelsey describe a scheme (SK) to perform secure and remote logging [SK99], which is considered the state of the art in the secure logging context (cf. introduction of [MT09]). In their model, Schneier and Kelsey define three roles. They consider a trusted machine $\mathcal{T}$ (e.g. a server in a secure location), an untrusted machine $\mathcal{U}$ (potentially the victim of an attack) on which the log entries are to be temporarily kept and a moderately-trusted external verifier $\mathcal{V}$ (e.g. an external auditor)[1].

[1]In this component, $\mathcal{T}$ role will be played by the *Log Collector*, $\mathcal{U}$ by the *Cloud Node*s and $\mathcal{V}$ by the *Log Reviewer* (cf. 6.3).

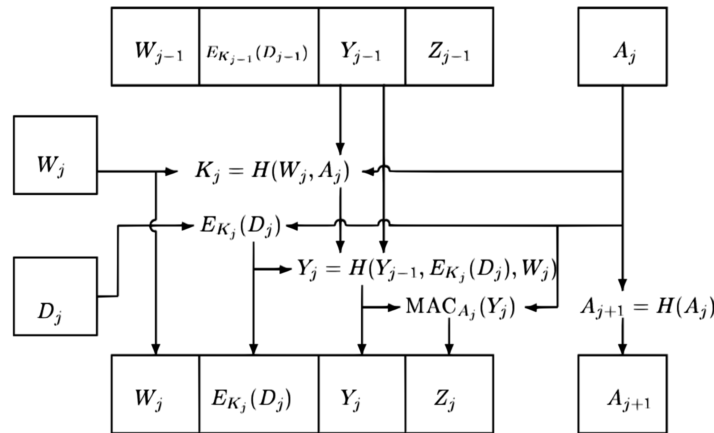

Figure 6.1: Schneier and Kelsey's log entry creation scheme.

In SK a log entry creation scheme is well-defined (Fig. 6.1). With this type of scheme, the immediate identification of log tampering (e.g. deletion of a log entry) becomes possible because the log entries are linked in an *hash-chain* by the element Y_j. Moreover, the integrity of logs is ensured by including an HMAC field (Z_j) in the log entries and the confidentiality of the logged data ($E_{K_j}(D_j)$) is guaranteed thanks to the usage of a symmetric cryptography mechanism. In SK when $\mathcal{V}$ wants to read the logs she requests and obtains the log entries from $\mathcal{U}$ and successively sends them to $\mathcal{T}$ for validation and decryption.

SK guarantees that, even if an attack succeeded, the log entries created before the attack cannot be tampered without leaving notice. However it is not possible to trust the log entries created after the attacker gained control of the system since he could simply log fake events.

6.1.2 Trusted Computing

To enhance the trust in the log service we make use of Trusted Computing (TC). We refer to TC as the technology specified by the Trusted Computing Group (TCG) [TCG], which has defined an architecture based on the Trusted Platform Module (TPM) [Tru07b], a low-cost chip whose main goal is to provide the hardware Root of Trust necessary to build strong security features. The TPM was designed for three main purposes: to securely store keys and data, to collect and store the *code-integrity measurements* of the platform where it is mounted (i.e. digests of running software with the relative configuration files, which represent the *state* of the system) and to report these measurements so that a remote verifier can assess the trustworthiness of the system. The TPM securely stores the measurements within the Platform Configuration Registers. A Platform Configuration Register (PCR) is a shielded location able to accumulate the measurements of the system components. The TPM guarantees that the values of the PCRs can only be updated by adding a new measurement through the "PCR extend" operation:

$$\text{PCR}_{\text{new}} = \text{SHA1}(\text{PCR}_{\text{old}}||\text{Measurement})$$

where Measurement is typically the hash of the component's binary or of a configuration file.

The TPM reliably reports the integrity measurements to an external entity through an operation called Remote Attestation. For this purpose, the TPM provides a primitive called `TPM_Quote` that generates a digital signature over the PCRs. The `TPM_Quote` is computed internally by the TPM with an RSA key whose private part never leaves the chip unencrypted

and can be considered as evidence that the platform equipped with the TPM that computed the quote is in the state represented by the PCRs (see [MSW+04a]).

In addition to the Remote Attestation, the TPM also provides another interesting functionality, the *sealing*. When a data is sealed, it is encrypted with a RSA key which is generated and used by the TPM. The TPM guarantees that the private part of the RSA key will never leave the chip unencrypted and that the unsealing operation (i.e. the decryption of the sealed data) is possible only if the platform is in the desired state specified at the time of sealing (i.e. if no attacks have succeeded or no modification to the configuration of the software was done).

6.1.3 Trusted Computing and log: related works

In this log service the TPM is used to ensure the code-integrity of the logging system, and hence to enhance its trustworthiness. However, the usage of tamper-resistant hardware has also been proposed to enhance the protection of the log entries. Chong et al. [CP03] proposed the usage of tamper-resistant hardware in a secure logging system. In [Acc08], the TPM is specifically employed to store and manage cryptographic keys and, hence, to enhance the security of the logging process. The paper [Aim10] studies the usage of trusted components for secure logging services aiming to minimize the trust assumptions required by such components. On the contrary, Yavuz and Ning [YN09] speculate on the impracticability of using tamper-resistant hardware for creating secure logs, especially in large distributed systems.

6.2 Log Service Model

In this section we model the log service and discuss the actors involved and the functionality provided. First we introduce the terminology, the actors and the main components of the log service. Then, we provide an overview of the system and we state its functionality. We finish this section by discussing its functional and security requirements.

6.2.1 Terminology, Actors and Main Components

Terminology

- *Log entry:* a record containing information about one event. The log entry may give only a partial view of the event. Moreover (part of) the data may be sensitive.
- *Event log:* a (usually small) set of log entries all related to a single event. This is the smallest set that provides the overall view on the event.
- *Log (or registry log):* a set of log entries, usually all related to a single object or actor. Note that a single log entry may be added to several log registries.
- *Log file:* a portion of a log, usually defined to specify a usage/security policy. Log files need to be opened and closed and this may involve cryptographic procedures to guarantee properties of the log entries contained in the file (e.g. forward integrity). A log file is usually a sequence of log entries (i.e. log entries are ordered in a log file), plus it has precise temporal constrains to allow, e.g., deletion of data older than some period of time.

Actors and Main Components

- *Cloud Component*: a component of the cloud infrastructure, intended as a service that can be either provided to the User or internally used by the Cloud itself, i.e. by other components. Here we introduce the following specializations: Log Core, Log Storage and Log Console. For a deeper discussion of other Cloud Components, see D2.4.1, Section 3.1.

- *User*: the end-user of the cloud services.

- *Cloud Admin*: the administrator(s) of the cloud infrastructure.

- *Log Reviewer*: an external authority with enough privileges to read (part of) the logs. The reviewer can be a *User*, or the *Cloud Admin* or an *Auditor* in charge of auditing the cloud system. Depending on its privileges the reviewer will have access to different subsets of the logs.

6.2.2 Overview

We take inspiration from the ontology of Cloud Computing presented in [YBDS08] and we define a new building block in the cloud infrastructure layer which can be used for secure logging by other cloud components as well as by applications running on the cloud. We begin with a simplified use case diagram for the cloud ontology (Figure 6.2), showing the interactions between the actors in the cloud. At each layer, we identify a "core pattern" where the User *uses* a Cloud Component. Because of the dynamic nature of the cloud, we also consider that the User may *react* to some event, for instance demand for additional resources in case of high load. In addition, we capture the relationship between layers: the *Cloud Provider* is responsible for *managing* the Cloud Infrastructure and *developing* the Cloud Platform, whereas the Developer *develops* the Cloud Application. Finally, we consider the role of the *Auditor*, that can *audit* either the Cloud Application or the Cloud Infrastructure.

By applying the core pattern we model the log service and provide an abstract view of its functionality, that is suitable for each layer of the cloud ontology (Figure 6.3).

6.2.3 Functionality

- *Create log:* the Cloud Component creates a log entry and stores it in the log. This is usually triggered when the User *uses* the component.

- *Read logs:* read stands for accessing all logs, possibly subject to a privacy-preserving policy, but usually not restricted to a view on a specific resource (in contrast to *retrieve* described later). This means that the entities entitled to read the log can have an overview of the whole cloud infrastructure where sensitive data may be preserved somehow (e.g., they may be blinded or made k-anonymous). For this reason, logs may be read only by the Cloud Provider, e.g. for monitoring resource usage and billing, or by the Auditor, for auditing purposes.

- *Retrieve logs:* the User accesses the logs related to his resources. This functionality can be seen as the composition of a *read* and a filter. The filter may restrict access to a specific resource or aggregate several entries in a more fruitful form, such as a chart. In addition, retrieving logs may reveal some unexpected behavior to the User, that thus triggers a *react* action. Note that usually the User only retrieves a small portion of the logs, but it may

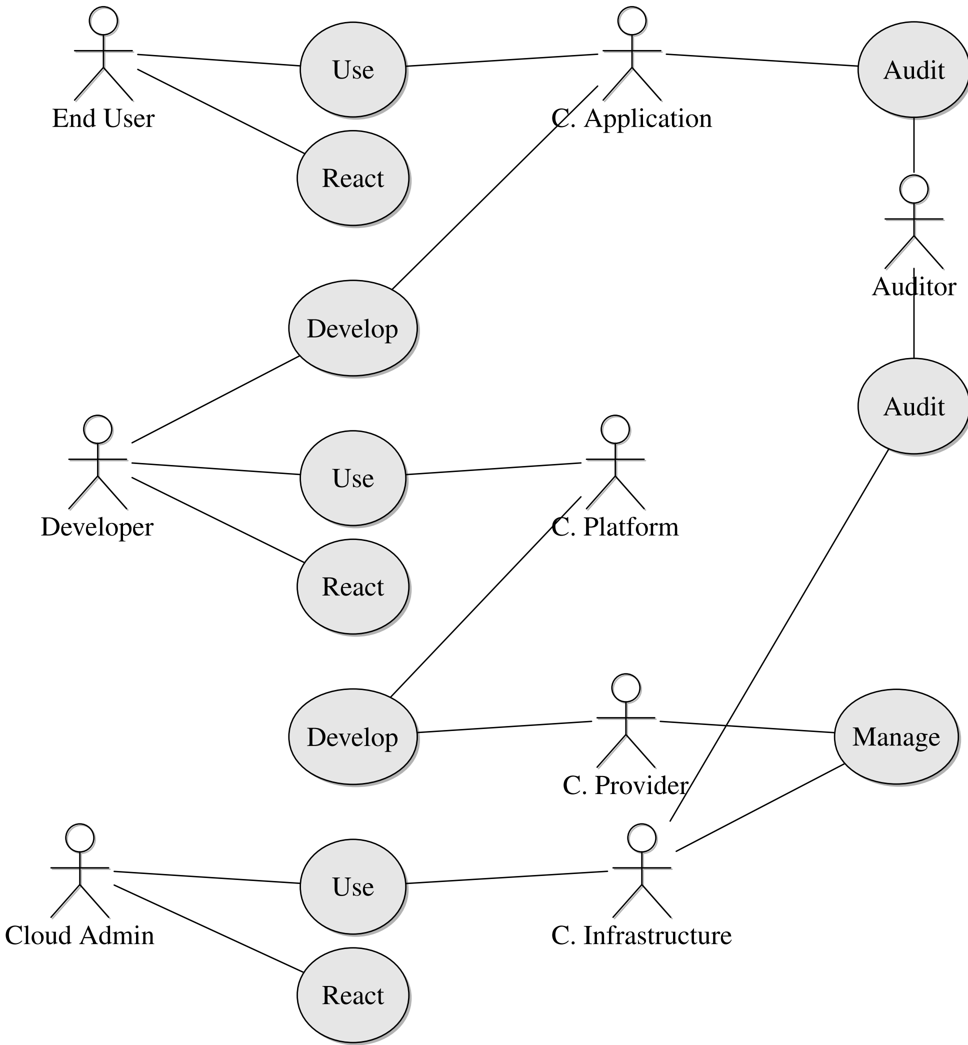

Figure 6.2: Use case diagram for cloud ontology.

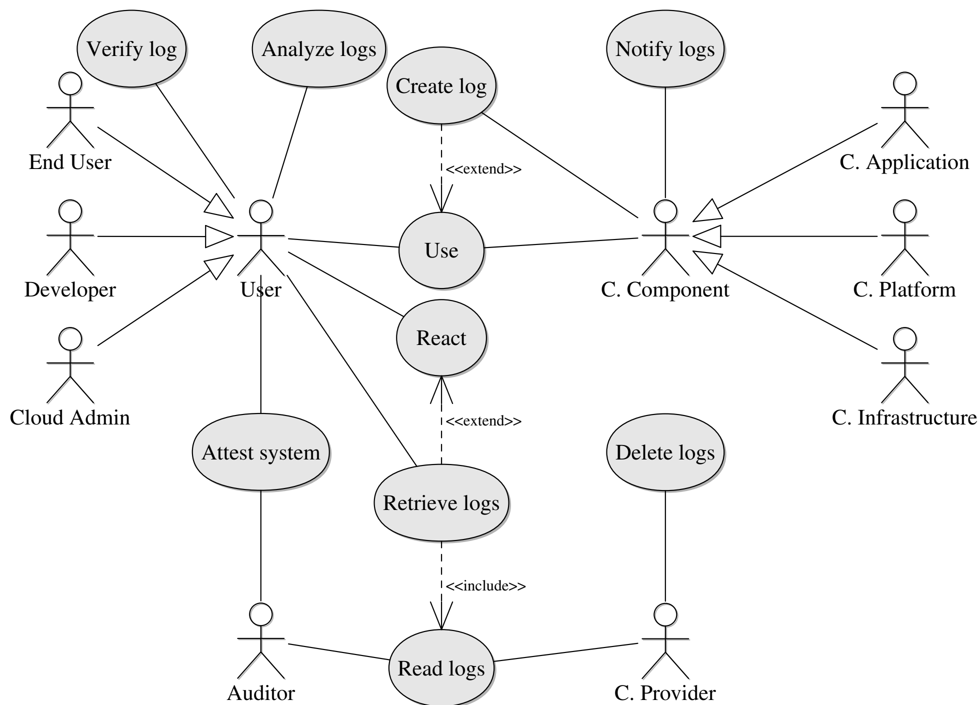

Figure 6.3: Use case diagram for log service. Compare it to *each layer* in Figure 6.2

happen the case where the User wants to dump a large (possibly all) portion of the logs. We indicate this latter case as *dump of logs*.

- *Verify logs:* after retrieving the logs, the User must validate their integrity. In SK scheme (and hence in our system), this is done by interacting with $\mathcal{T}$. During this step, the forward integrity of the log entries is verified and, therefore, attacks to the logs are detected. Upon the successful verification of the log entries, the User may consider the information contained within the log as truthful.

- *Delete logs:* the Cloud Provider should be able to delete logs and log entries. The deletion of logs is usually a way to comply with data retention regulations, for instance logs must be kept for one year, then deleted. On the other hand, the deletion of a log entry is a non-common procedure, usually avoided, that may require a special authorization (possibly also involving the legal framework, e.g. the authorization of a judge).

- *Analyse logs:* in many cases, processing a large amount of log entries can be a computationally intensive operation. Since the log service is part of a cloud system, it can provide a functionality for log analysis to the User, for instance a mechanism based on MapReduce where the User defines his function for processing logs. This is a more advanced feature than the *retrieve*, where how to filter logs is imposed by the log service.

- *Attest system:* attestation of the cloud system is a feature required by User and Auditor. The log service is a key component in this process, that can significantly improve the information necessary for an attestation, for instance by providing evidence of some (both intended or unintended) behavior.

- *Notify logs:* since the cloud infrastructure is largely sparse and the log entry creations depend on external events, it may be useful for the log service to implement a function that the Cloud Components that log events may use to notify a "central" entity that a log entry has been created, without having to transport the actual log entry. This functionality can be especially useful to keep the consistency when complex events that happen on different Cloud Components occur (e.g. the migration of a VM).

6.2.4 Functional and Security Requirements

Here, we discuss the functional and security requirements of the log service, that highlight the differences between log service and other cloud components, specifically the storage. Log entries have particular security requirements which are not usually provided by a normal storage. Moreover, most of these requirements are not necessary for normal buckets of data, therefore we can not simply foresee to extend an existing storage with enhanced security features to implement a log service, as this would probably degrade the functionality of the storage.

Log entries are usually small pieces of data, created at high rate, possibly by many different nodes of the cloud concurrently. Differently from other data, they have a fixed structure which describes an event that happened on the system together with additional metadata such as the time when the event happened or other correlated information. Moreover, they are usually persistently stored for long periods of time.

Since deletion on logs is not usual, the log can reach a very large size which makes its management difficult. In addition, it must be considered that reading one log entry alone has a limited usage (e.g. this can be used to detect that an error occurred), while retrieving a set of

log entries allows for statistical analysis, which brings to deeper investigations. Therefore, the log service must be capable of recording many log entries and of providing a view on a subset of the log entries that satisfies a particular query (for instance, depending on who created the log entry or when the log entry was created).

In order to ensure security of the log entries, a log system must provide the integrity over the stored log entries. In addition to the integrity of each log entry, it is desirable to provide the *forward integrity* security property of the whole log. The forward integrity implies that, if an attacker succeeds in compromising the log system, he can not modify log entries collected before his attack without being noticed [BY97]. This important property guarantees that if the actions of the attacker are logged, it will be possible during a later analysis to spot them and investigate the episode. A possible way to provide forward integrity is to use an *append-only* memory which prevents any modifications of data already inserted (in early works on this topic which do not make extensive use of cryptography, it has been proposed to use printers, write-once read-many disks or CD-ROMs for this purpose).

Logs often contain sensitive information about the usage of a certain system (the cloud in our case), so it is important to mediate every access in order to prevent leakage of information. Therefore, having a strong Access Control mechanism in place is of primary importance. Since it is not possible to guarantee that the log system will never be compromised, this Access Control must be embedded within the log entry itself, in a way that the relevant data is cryptographically protected at the creation time and only the authorized people will have access to the keys required for accessing the data (cf. [SK99]). Moreover, it is desirable to limit the access to logs according to some usage policy. For instance, the user should be capable of retrieving the logs that belong to his resources, while a forenser should be capable of reading all the log entries created during a period of time around a suspicious event.

6.3 Architecture

We base the log service on the scheme proposed by Schneier-Kelsey because it provides most of the security features required by logs and is still considered as the state of the art.

6.3.1 Applying SK scheme to the cloud environment.

The Schneier-Kelsey approach needs to be adapted to the cloud scenario because, in our case $\mathcal{V}$ does not have any information about the cloud physical structure (e.g., the IP address of cloud nodes); therefore the described interactions between $\mathcal{V}$ and $\mathcal{U}$ to review and validate the log entries cannot be realized.

For instance, we consider a cloud customer wishing to access the log entries related to her own Virtual Machine. If the cloud logging process were based on SK, the customer ($\mathcal{V}$) would be unable to make access to the requested log entries because she does not have any information about cloud physical structure and therefore she would be unable to interact with the cloud node on where the Virtual Machine ($\mathcal{U}$) is running. Such an issue can be mitigated by introducing a centralization point of logging information. This way, to make access to the requested log entries the customer has to interact only with the centralization point, which collects all log entries received from the cloud nodes. This approach is the base of our logging model in which we identify three actors.

The Log Service ($\mathcal{L}$), is the recipient of remote and secure log entries. $\mathcal{L}$ may be specialized into Log Core which may be considered as the $\mathcal{T}$ entity defined in SK, Log Storage which acts

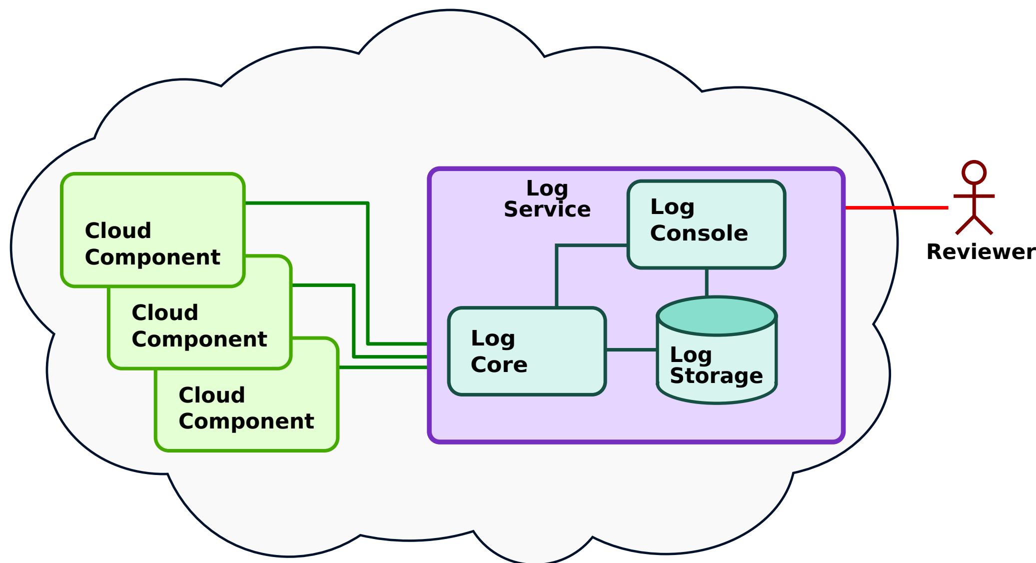

Figure 6.4: High level architecture of the Log Service.

as collectors for all cloud log entries and Log Console which is the primary input port for the User to use the Log Service. Here we mainly focus on enhancing the trustworthiness of the logging process and will refer to the components of the Log Service generically as $\mathcal{L}$, but more information about $\mathcal{L}$ internal architecture can be found in Section 7.7 of D2.4.1

The Cloud Component ($\mathcal{C}$) is an untrusted physical machine that logs its local activity, generates the corresponding log entries and periodically sends them to $\mathcal{L}$. The *Log Reviewer* ($\mathcal{R}$), is an external entity which wants to review the log entries stored by $\mathcal{L}$. With respect to SK, $\mathcal{C}$ and $\mathcal{R}$ can be considered respectively as $\mathcal{U}$ and $\mathcal{V}$. Fig. 6.4 shows the UML deployment diagram describing the interactions between the previously defined actors. In detail, the diagram shows how $\mathcal{R}$ may access the log entries generated by several $\mathcal{C}$s by using a public interface provided by $\mathcal{L}$.

6.3.2 Logging process phases.

The logging process is structured as the composition of four distinct and successive phases. Such phases represent the life-cycle of log entries and are defined as follow:

1) Log entry creation: in this phase $\mathcal{C}$ computes and temporarily stores log entries which are structured as defined in SK. The *correctness* of the log entries directly depends on the *trustworthiness* of the system. We consider a log entry to be correct if it is computed by a good system and its content has not been modified or corrupted *after* the computation time (see *Forward Integrity* [BY97]). Moreover, we consider a system trustworthy if no corrupted or malicious software is running and common security features like *firewall* or Access Control system are present.

2) Transport to the log service: in this phase $\mathcal{L}$ collects the computed log entries from $\mathcal{C}$. In Cloud Computing many entities access the same communication infrastructure. Since this is subject to security risks (e.g. unauthorized access to sensitive data), it is necessary to guarantee the integrity and the confidentiality of log entries during transfers.

3) Log entries storage: in this phase $\mathcal{L}$ aggregates all received log entries in large logs. Since the logs are the only valid evidence of past events it is necessary to securely store them. This entails ensuring the integrity and the confidentiality of the stored data over the time. Such a feature is crucial for forensic activities.

4) Logs review: this phase occurs when $\mathcal{R}$ wants to access the log entries collected by $\mathcal{L}$. Since the information contained in log entries may be sensitive, the presence of Access Control system is mandatory. To address this problem, in SK log entries are encrypted by using a key which is derived locally on $\mathcal{U}$ by using the W_j element (see Fig. 6.1) that serves as permission mask for $\mathcal{V}$. In our model, $\mathcal{R}$ has to authenticate on $\mathcal{L}$ in order to make access to the log entries. Credentials provided by $\mathcal{R}$ during the authentication process are used, similarly to W_j element in SK, to regulate the access to the log entries.

6.3.3 Relationship with Other Components of the Cloud.

Here we highlight the relationship of the Log Service with the other components of the cloud. For this purpose, we recall the cloud ontology introduced in Section 6.2.2.

A first relationship between the log service and the other services at the infrastructure layer appears, as all these services employ the log service to log their relevant events for later retrieval and analysis. Similarly, the log service may rely on other services to accomplish its tasks, e.g. for persistent storage or reliable communication. An example of a log service in this term can be Amazon CloudWatch[2].

The log service is built upon a kernel layer that provides strong mechanisms to enhance the security of the log system and the trustworthiness of the log data. In particular, it is important to ensure that the components can not lie in what they record in the log entries and that the log system is protected against attacks that may happen in the rest of the system.

Moving to the upper layers, the log service may be used as a building block to exploit its security features and provide enhanced services. In Platform as a Service (PaaS) there are at least two. First of all, the log API can be provided to developers to extend the logging possibilities to the applications, similarly as in Google App Engine[3]. In this case the applications will benefit from the security features of the log service for their own purposes. Then, the cloud could provide an analysis framework which can be built by mixing log service and other cloud services, for instance a MapReduce service. In this second case, the users may take advantage of the powerful cloud infrastructure to process large quantities of data (see, e.g., [BPE+10]). Finally, applications running within the Software as a Service (SaaS) layer, may exploit the log service at runtime to log application-events or to notify users about noteworthy events. An example of such a service is Loggly[4].

6.4 Enhancing trust in the logging process with TC

In Cloud Computing computations on data are performed on behalf of the users on cloud nodes which are not under user's direct control. While in possession of user's data, the cloud provider could compromise its confidentiality or integrity (e.g. by revealing sensitive information to a third party or by executing wrong computations on data).

[2] http://aws.amazon.com/cloudwatch
[3] http://code.google.com/appengine/articles/logging.html
[4] http://www.loggly.com

Therefore, the user must *trust* the cloud provider in order to give her data. The trust may be granted for different reasons, for instance the reputation of the cloud provider.

In this context, the availability of a reliable logging system may increase the trust in the cloud since, if any accident happens, the log helps to investigate which problem arose and where it originated from. However, the logging becomes of little importance when the user wants to verify whether the cloud is not complying to certain policy, since the entity that may break the policy is the same that logs the events (and therefore it may decide not to log the suspicious ones).

A possible approach to mitigate this last issue is to employ techniques that allow to verify the behavior of remote entities, such as TC. We have identified a number of possible applications of TC technology to the logging process.

The main problem regards the log entry creation, since the log entries may be tampered with in order to hide a misbehavior. This may happen when an attacker successfully obtains control of a cloud node. In this case the attacker may hide the evidences of his actions. However, as previously mentioned, also the cloud service provider may have interest to tamper with log entries, for instance in the case of a violation of the policy agreed with the user.

The first case is already considered by logging schemes such as SK. This scheme ensures that it is not possible for an attacker to delete or modify log entries created before his attack without being noticed. Unfortunately, SK does not provide any guarantee about the log entries that will be created after the attack.

Also in the second case, it is not possible to provide guarantees about the correctness of the log entries since the entity that wants to tamper with the log entries is the administrator of the platform and hence may decide to log fake events.

A possible solution is to use the quote function provided by TPM, which provides an evidence that a remote platform is behaving as expected for a particular purpose. By embedding such evidence within each log entry, it is possible to guarantee that the cloud node was not attacked or modified at the moment of the creation of the log entry.

A modified version of the SK log entry creation scheme is provided in Fig. 6.5. A `TPM_Quote` has been added to the classical SK scheme to convey the state of the platform at the log entry creation time. Thanks to this quote, it is possible to decide if the platform that created the log entry was good. In order to bind each quote to its specific log entry and to ensure freshness, a digest (e.g. SHA1) of the information composing the log entry can be used as nonce of the `TPM_Quote`. As the state of the platform may change between two different log entries, a new quote must be computed each time an event is logged.

Note that this is a preliminary design where the TPM is highly involved. In order to make this proposition realistic, it is important to minimize its usage, since performing a Remote Attestation for each transmission of log entries in very large and distributed environments such as clouds may pose big performance issues. For this task, it is possible to employ one of the existing techniques which increase the performance of Remote Attestation (see, e.g., Stumpf *et al.* [SFKE08]).

6.5 Possible future research directions.

Beside the log entry creation, we can foresee TC applications also for the other logging phases. Here we only sketch these applications, which will be developed in future works.

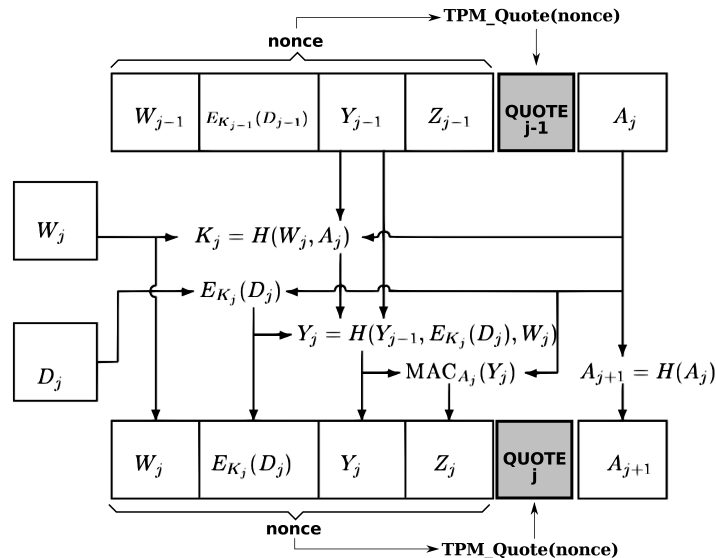

Figure 6.5: Schneier and Kelsey's log entry scheme extended with `TPM_Quote`.

Transport to the log service. While being transported from $\mathcal{C}$ to the remote $\mathcal{L}$, the log entry must be protected to ensure confidentiality and integrity. Moreover, both $\mathcal{C}$ and $\mathcal{L}$ must be authenticated. However, the authentication alone provides only little information about the trustworthiness of the remote parties. For instance, $\mathcal{C}$ can determine that it is talking with the authentic remote $\mathcal{L}$, but it cannot decide if $\mathcal{L}$ is trustworthy enough to send it the log entries created, and viceversa.

TC may be used to convey some evidence about the configuration of the peers, together with the authentication data. The coupling of a secure channel together with the attestation data is called Trusted Channel [AGS+08]. After the establishment of a Trusted Channel, each party can determine if it is safe to communicate with the other one. This can be achieved by comparing the integrity measurements received from the counterparts with a white (or black) list of accepted (or refused) possible configurations.

Storage of logs. Usually logging schemes such as SK require a trusted entity ($\mathcal{T}$) to keep logs authentication master keys. Since these keys, if compromised, can be used to forge log entries without being noticed, it is important to strongly protect them.

The TPM can be used to enhance the trustworthiness of the remote $\mathcal{L}$. In particular the *sealing* can be used to ensure that the log master integrity keys are strongly protected by hardware means and that their usage is possible only if $\mathcal{L}$ is in an acceptable state (for instance, only if no modifications to the wanted configuration were done).

Moreover, similarly to $\mathcal{C}$ at the log entry creation time, also $\mathcal{L}$ may embed a `TPM_Quote`. The former evidence guarantees that the log entry was created by a "good" entity, whereas this latter one ensures that the entity that stores the log entries was "good" when it stored the log entry, and hence that also the log entries aggregation operation was correct.

Review of logs. When $\mathcal{R}$ wants to verify the logs of a computing system, normally she should directly access such system. However, because a cloud is generally extremely distributed, directly accessing the resources may be difficult or even impossible. Moreover, Cloud Computing is based on virtualization, which implies that a Virtual Machine under investigation may have

migrated on many different physical nodes (e.g. for load balancing), making the physical access even more challenging.

For these reasons, it is essential to provide $\mathcal{R}$ with as much information as possible that may assess the trustworthiness of the logs. By inspecting the `TPM_Quote` embedded by $\mathcal{C}$, $\mathcal{R}$ may infer that the log entries were correctly created, while the `TPM_Quote` embedded by $\mathcal{L}$ ensure that $\mathcal{L}$ was "good" at the time when the log entries were aggregated to the log, and hence correctly stored.

However, $\mathcal{R}$ cannot say anything about the state of $\mathcal{L}$ at the time when the logs are reviewed. Since $\mathcal{L}$ is the trustworthy element that guarantees about the integrity of the logs, it is mandatory that it works properly, otherwise misbehavior cannot be detected. For instance in SK scheme, the log reviewer asks to the trusted entity if the logs she is reviewing are intact and authentic. The key point of this operation is the assurance that the trusted entity is actually trustworthy. Performing a Remote Attestation of $\mathcal{L}$ at the time of verification may provide the necessary warranties about its trustworthiness.

$\mathcal{R}$'s credential may give access to large portion of logs (and hence to many confidential data). Therefore, it is important to protect them. TC may help in this context as well. Since TPM is shipped with many commodity computers, $\mathcal{R}$ may employ its capacity of protecting data with hardware means in order to keep her keys safe (e.g. see [KS09]).

Chapter 7

Secure Cloud Maintenance - Protecting workloads against insider attacks

Chapter Authors:
Sören Bleikertz, Zoltán A. Nagy, Anil Kurmus, Matthias Schunter (IBM)

Malicious insiders are a substantial risk for today's Cloud Computing infrastructures. A single malicious cloud administrator can eavesdrop or damage business-critical or personally identifiable information and computations of thousands of cloud customers. To protect cloud users against such insiders, we propose a novel approach that enables a security team to protect privacy and integrity of cloud users against attacks by system administrators during operation and maintenance. We achieve this by minimizing the privileges of administrators during operation and maintenance while re-establishing the security of a compute node once administration is completed. By default, administrators' access to cloud servers is disabled since cloud operation is automated. For manual maintenance operations, we propose five fine-grained privilege levels that balance the security objectives of cloud users with the operational requirements of cloud administrators. We demonstrate how existing cloud architectures need to be extended to incorporate our approach. We prototyped our management approach using the OpenStack cloud platform. Policy enforcement has been prototyped by introducing SElinux policy enforcement into the KVM compute nodes, in order to demonstrate the feasibility of our approach in practice.

7.1 Introduction

In recent years, *Cloud Computing* has gained remarkable popularity due to the economic and technical benefits provided by this new way of delivering computing resources. Businesses can offload their IT infrastructure into the cloud and benefit from rapid provisioning, scalability, and cost advantages. Small companies including start-ups benefit from the low up-front investment as well as the ability to only pay for capacity actually used. Large customers including the US government benefit from increased agility and productivity while reducing costs [Kun10]. While Cloud Computing can be implemented on different abstraction levels, we focus on *Infrastructure Clouds* such as Amazon EC2 [EC2] that provide virtual machines, storage, and networks.

Although the benefits of Cloud Computing are evident and end-users demand cloud services, security is a major inhibitor [MG09a] and various security risks have been identified [Clo09a, ENI09]. A malicious insider, such as a cloud administrator, can easily inspect the virtual machines of cloud users and retrieve sensitive information [RC11]. Rocha *et al.* show that

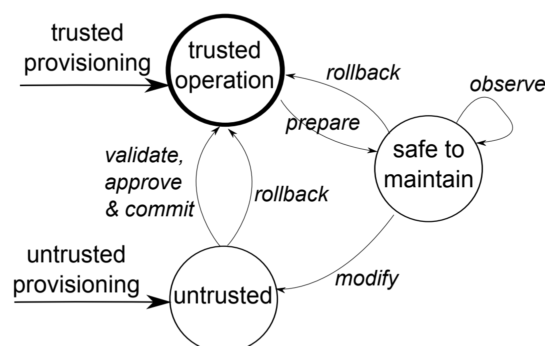

Figure 7.1: State Diagram for a Compute Node per Customer

root access to a Xen[1] virtualization technology, which is also used by Amazon EC2, or KVM cloud infrastructure can be used to retrieve passwords, keys and to migrate virtual machines while circumventing existing security controls. Insider attacks are constantly identified as a high-impact risk where a few malicious insiders can affect the security of many users. Furthermore, this risk of insider attacks is amplified by the fact that administration is often outsourced and thus trust in administrators is sometimes limited.

7.1.1 Protecting Cloud Users during Server Maintenance

Solutions to combat insider attacks cannot rely on cryptography alone [VDJ10]. While normal operations are automated and are based on trusted system management processes [SGR09], an open challenge is how to protect end-users during maintenance. In practice, maintenance requires substantial privileges and is usually performed manually. In particular resolving complex problems requires full access where administrators can see and modify all parts of a system. Today, no technology can protect against insider attacks during such maintenance activities.

Our focus is on today's dark data centers that are remotely managed and where only a few trusted administrators have physical access to the servers. We introduce three administrator roles where we require separation of duties. The first role is the hardware maintenance team, which is the only one allowed to access the datacenter and is responsible for adding and removing hardware. The second role is the security team that defines the security policy for the datacenter. This includes approval of the infrastructure cloud software executed on the systems. The third role are remote maintainers. Their responsibility is to maintain the individual compute nodes. A particular task is to perform problem determination and resolution by logging into individual infrastructure elements. While the first two groups are rather small, the latter maintenance team poses the biggest security risk since this task requires a large team that is often outsourced.

We ensure that such remote administrators cannot affect the integrity and confidentiality of customer workloads. Our approach is based on two key concepts. The first is a consistent enforcement of the principle of least privileges [SS75]: We define several levels of increasing privileges that administrators can select. This allows administrators to elevate their privileges as needed. Depending on the actual privileges chosen, we then protect privacy and integrity of the workload during the complete maintenance life-cycle. Furthermore, we reassess or restore the integrity of the platform before returning the compute node back into normal operation. Technically, this required us to resolve two key challenges.

[1] http://xen.org/

The first challenge is to ensure that for low privilege roles the platform remains trusted under maintenance. The second challenge is to ensure that once a platform can no longer be trusted due to high privileged access that does not guarantee any trusted computing base, integrity can be restored and potential modifications of the maintenance can either be approved as trusted by a specific customer or else be removed.

Figure 7.1 illustrates the resulting life-cycle of our systems. By default, a compute node is operating and no operator can perform maintenance. If maintenance is performed, the compute node is moved into maintenance mode and its payload is protected. If the node is only observed, then it can be rolled back into operation. If changes were made that may affect the integrity of the system, the node enters an untrusted state. This node can re-enter production only after an externally verifiable rollback or "approval and commit" of the performed modifications.

Note that unlike existing proposals, we are able to protect a core trusted computing base and thus do not require special hardware such as Trusted Platform Modules (TPMs).

7.1.2 Outline

In this chapter we make the following major contributions. Section 7.2 defines the security objectives, the threat model, and the maintainability requirements of cloud administrators.

Section 7.3 then proposes an extension for common infrastructure cloud architectures that embeds our core concept of privilege minimization. Section 7.4 introduces five distinct privilege levels that an administrator may have on a compute node, with no access being the default. We furthermore show how an administrator can elevate his privileges through an agent, how that agent can revoke the privileges, and how the impact of the maintenance task can be assessed before returning to normal operational mode. Section 7.5 then argues that our concept fulfills the security objectives stated in Section 7.2.

Finally, in Section 7.6 we describe and evaluate a prototype of our approach, we discuss related work in Section 7.7, and conclude our article in Section 7.8. Overall, we managed to build protection against insider attacks bottom-up: Once a cloud infrastructure can be protected against insider attacks, this lays the groundwork for securing higher-level services that provide actual applications to end-users.

7.2 Requirements and Threat Model

We now specify our functional and security requirements in detail. We do not cover the functional requirements of the cloud platform and focus on the maintenance requirements of the administrator and the security requirements of the cloud user.

7.2.1 Maintainability Requirements of Cloud Administrators

Protecting cloud users against malicious cloud administrators by entirely disallowing access to the Cloud Computing infrastructure is only feasible during normal operations when no problems occur. A new cloud architecture that protects users against insider attacks has to balance between the security objectives of the users, the functional requirements of the administrators, and operational usability.

We used the following maintenance tasks to motivate and evaluate our design: Reading log files, configuration files, and system parameters; Modifying configuration files; Patching and Installation of system binaries; Full administrative access for complicated trouble-shooting

(e.g., kernel related); Running existing or newly installed executables for testing purposes and root-cause analysis. While some of these maintenance tasks do not pose a threat to the end user, others such as full access have the potential to violate security and privacy of the cloud users.

7.2.2 Security Objectives of Cloud Users

Our main concern is the exposure of sensitive or personally identifiable information belonging to the cloud user to an administrator. The security objectives apply only to compute nodes, and we exclude the network from our protection approach.

Confidentiality requires that any remote administrator of the cloud provider must not be able to read information stored or processed by the cloud user. This includes information stored in memory and on the hard disk. A cloud user may grant an exception to an individual administrator if required due to maintenance reasons and prior informed consent is given by the user. An example motivating such consent is a problem that disappears once a VM is stopped and that cannot be reproduced with a test VM. **Integrity** requires that any remote administrator must not modify any data or executables belonging to cloud users. This includes the guarantee that the administrator cannot modify the run-time platform in a way that is not approved by the security team. **Availability:** Any remote administrator must not be able to degrade the availability of a compute node except during maintenance operations.

7.2.3 Threat & Trust Model

We consider curious, careless, and malicious inside attackers. We assume that an attacking administrator wants to read and/or modify data belonging to a cloud customer, which is stored or processed within the virtual machine and storage of that particular user. Due to their privileged role within the cloud provider, administrators have access to the servers hosting the virtual machines of the users. Typically this access is highly privileged (root access) and allows inspection and modifications of users' virtual machines. Dark datacenters with remote administration means that we assume that insiders cannot attack the physical hardware, e.g., through tampering.

Our aim is to protect against attacks by these remote administrators and we assume that the following entities behave correctly. *Infrastructure Management Software* is responsible for acting upon requests from the user and keeps state about the infrastructure. *Cloud Provider's Security Team* is responsible for approving server templates used for provisioning. In a high-security setting, this assumption may be reduced by requiring template approval from actual end-customers. *Server Provisioning* is based on templates approved by the security team. Other means of provisioning a server are disabled.

7.3 Extending an Infrastructure Cloud Architecture

In this section we briefly present current and commonly used architectures for infrastructures cloud using OpenStack as an example. We introduce our extensions to these architectures in order to fulfill the goal of minimizing administrator privileges.

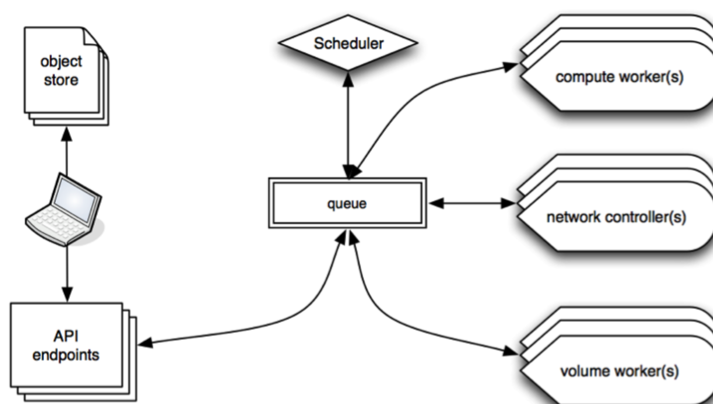

Figure 7.2: OpenStack Architecture[2]

7.3.1 Current Architectures of Infrastructure Clouds

We identified four components that are commonly used in infrastructure cloud architectures. We illustrate the overall architecture and explain the components in more detail for OpenStack. However, other infrastructure cloud architectures, such as OpenNebula or VMware's vSphere, are following the same principles. An architecture overview is depicted in Figure 7.2.

Management Interface: The management interface is responsible for serving requests from the infrastructure cloud users and forward these requests to the appropriate components in the infrastructure. Typically, the management interface is provided in form of a web service or a specific API server. In the case of OpenStack, the *API endpoints* fulfill the role of the management interface.

Management Communication and State: The management of infrastructure clouds is composed of multiple components interacting which each other. Furthermore, the state of the infrastructure clouds is required for management decisions, e.g., for virtual machine placement based on current load of compute nodes. In the case of OpenStack, the *queue* provides a distributed message queuing platform and the *Scheduler* maintains a state of virtual machine placement.

Compute Nodes: These nodes provide computational resources to the cloud users in form of virtual machines. In case a user requests a new machine, the management infrastructure will select a compute node, e.g., based on its current load, and provision a new virtual machine for the user there. On each compute node, an interface is provided in order to manage the computational resources on this particular node. The *compute workers* fulfill this role in OpenStack.

Network & Storage Nodes Besides computational resources, infrastructure clouds also provide network and storage resources to cloud users. Similar to compute nodes, there exist nodes providing network functionality and nodes providing storage volumes, which are also controlled via a management interface. In OpenStack, *network controllers* and *volume workers* fulfill these roles respectively.

7.3.2 Architecture Extensions for Minimizing Administrator Privileges

We propose to extend current architectures of infrastructure clouds with the following components, in order to achieve a minimization of administrator privileges. Figure 7.3 illustrates the overall architecture.

[2]Figure from `http://nova.openstack.org/service.architecture.html`

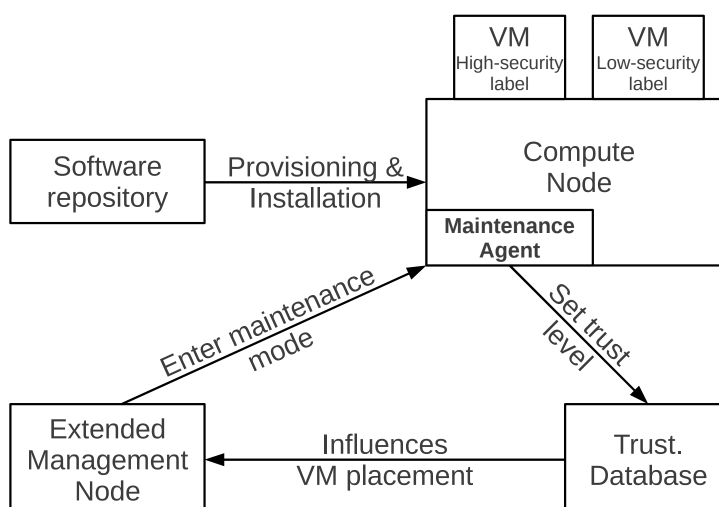

Figure 7.3: Extended Architecture Overview

Maintenance Agent: The maintenance agent (MA) is the major component in realizing an infrastructure cloud with minimized administrator privileges. It is running on each compute node and is responsible for managing administrator privileges on such a node. The maintenance agent extends the functionality of current compute nodes and their management, and it is essential for meeting the security objectives.

Maintenance Management: The management interface has to be extended for the administrators, in order to enable them to switch a compute node into maintenance mode. The extended management interface cooperates with the maintenance agent and the database of trustworthiness for switching a node into maintenance mode and record the trust level of that node.

Database of Trustworthiness of Compute Nodes: In the case of maintenance of a compute node, the trustworthiness of this node might be reduced depending on the cloud users' trust towards the provider and the criticality of their workloads. In addition, the system needs to track what nodes are in production and what nodes are in maintenance mode. Therefore, a database is required that records the state and trustworthiness of each compute node and is consulted during virtual machine placement, e.g., provisioning of a newly requested VM or due to migration.

Virtual Machine Security Labels: We introduce different security labels for virtual machines, in order to differentiate between the criticality of the workloads. A simple classification can be based on *Low*, *Medium*, and *High*. The criticality influences the placement of virtual machines when consulting the database of trustworthiness. Furthermore, it will impact the cost of switching from maintenance mode back to normal operational mode of a compute node. This requires an extension to the management interface and the management state, in order to enable the specification and recording of security labels.

Software & Template Repository: In order to start with an initial trusted state of the compute nodes, we assume a secure provisioning of the nodes with a server template approved by the security team of the cloud provider. In our architecture, we introduce a repository for software and server templates that is managed by the security team. All compute nodes are provisioned by these templates and software can only be installed from this repository. In most Linux distributions signed software packages are the norm and we can leverage this existing infrastructure.

7.4 Minimizing Privileges of Administrators during Maintenance

In Figure 7.1 we illustrate the overview of the maintenance life-cycle of a compute node. In this section we discuss the different states in detail and describe how the state transitions are performed. We propose a set of fine-grained privilege levels an administrator can request to enable maintenance tasks, and discuss how these levels can potentially impact the security objectives of cloud users. We explain how privileges are elevated using our maintenance agent and how eventually the impact of privileged maintenance tasks can be assessed before returning to normal operational state.

7.4.1 Initial Trusted State

The initial state of compute nodes within the maintenance life-cycle of a node is their normal operational state. In that state, the system is trusted by assumption, namely that the system was securely provisioned using server templates approved by the trusted security team of the cloud provider (cf. Section 7.2.3). Furthermore, administrators have no privileges on the compute node and the maintenance agent is running on that node.

7.4.2 Privilege Levels

Based on the analysis of the desired tasks and the resources and permissions of the cloud platform, we have identified five distinct privilege levels. The privilege levels monotonically increase in capabilities, as well as their potential impact, and are modeled in consideration of the maintainability requirements of administrators (cf. Section 7.2.1). An overview of the levels is given in Table 7.1.

Level	Capabilities	Restrictions	Security Impact
$P_{\emptyset}$	none	all	none
P_{read}	read-only access	No read to virtual machines related data	none
$P_{write_{\circ}}$	white-listed write access	P_{read}'s restrictions apply. Modifications limited to software installations, white-listed files, and certain system parameters	none
$P_{write_{\bullet}}$	black-listed write access	no modifications to bootloader, kernel, policy enforcement, maintenance agent, file system snapshots, package manager transaction logs, and certain system parameters	potential disclosure and modification of virtual machine data
P_{∞}	full access	none	any disclosure and modification of platform and data

Table 7.1: Overview of Privilege Levels

No access ($P_{\emptyset}$): This level provides no privileges at all on compute nodes and it is the default level for all administrators. For this level no impact on the cloud users' security exists.

Read-only access (P_{read}): The next privilege level provides read-only access on the compute node. The purpose is to gather system data, e.g., log files or system parameters, for initial investigations and trouble-shooting. Read access to data related to cloud users' virtual machines is prohibited, in order to maintain the security goal of confidentiality. Since this level only provides read-only access, the integrity of the users' data is not at risk.

White-listed write access ($P_{write_\circ}$): In this privilege level write access is allowed but limited with a white-list approach, i.e., we are limiting writing to a set of predefined files and resources. P_{read}'s restrictions also apply to this level. We are allowing software installation, update and removal, exclusively through a trusted repository. The software modifications are recorded in a package manager transaction log, that needs to be protected against modification. Furthermore, we allow the modification of a specified set of system parameters. This level allows an administrator to install new software or revert software to an older version, fine-tuning system parameters, and changing configuration files. Since only software from the trusted repository (which has been aproved by the security team) can be installed and modifications of files and resources are limited by a (conservative) white-list, the confidentiality and integrity of cloud users data is not at risk.

Black-listed write access ($P_{write_\bullet}$): In the case that the white-listed write access is still too restrictive for the trouble-shooting process, we can increase in privilege level to a write access with a black-list approach. Write access to any file or resource is allowed except for the following vital system and security components: bootloader, kernel, policy enforcement, maintenance agent, file system snapshots, package manager transaction logs, and certain dangerous system parameters. Since a black-list write access is much less restrictive compared to the previous white-list approach, we expect potential impact on the security objectives of cloud users. Therefore, the measures that will be explained in Section 7.4.4 have to be taken.

Full access (P_∞): At this level, the administrator has full access to the system and no restrictions are applied. Since an administrator can modify and read anything on the system (including the kernel) the security objectives can not be maintained at this level. Similar to $P_{write_\bullet}$, measures have to be taken for workload protection.

7.4.3 Policy Enforcement for Privilege Levels

The capabilities and restrictions of the privilege levels discussed in Section 7.4.2 are specified in a security policy that is enforced on each compute node. The privilege levels are modeled as roles and administrators are dynamically assigned to these roles. The policy enforcement system is based on *Mandatory Access Control* (MAC) and *Role-based Access Control* (RBAC). An example of such a policy enforcement system is *Security Enhanced Linux* (SELinux) [LS01], which we are using to illustrate our security policy.

SELinux associates a *security context* with each file, socket, device and process. A security context is essentially a *(user, role, type)* triple that is either directly specified, e.g., by *labeling* the files, or dynamically computed by SELinux. The dynamic computation is based on transition rules specified in a *security policy*. For example, the policy can specify that a process created by executing a file of type *httpd_exec_t* will transition into the type *httpd_t*. In order to cope with fine grained access control, the security policy also contains a white-list of operations, such as executing a file, binding a name to a socket, or sending a signal to a process. The white-list allows these operations to be performed by a given source type (such as a process) on a target type (such as a file, device, socket or process). Basically, SELinux policies uses these two essential mechanisms – type transition rules and allow rules – to implement the least privilege principle.

No access ($P_\emptyset$): The policy module implementing this privilege level has no allow rules at all, which means it will not allow any forms of login, e.g., spawn a system shell.

Read-only access (P_{read}): This is a restricted user shell based on SELinux's userdom_restricted_user_template macro, which creates a new role and type for restricted users. Furthermore, access to system and process information along with the read privilege on all file types except any type related to virtual machines (i.e., having the attribute *virt_domain* or *virt_image_type*) have been granted.

White-listed write access ($P_{write_\circ}$): White-listing files can simply be done by specifying allow rules for the corresponding types. We explain below how we allow trusted software updates and modifications to selected system parameters.

In most systems, package management is usually not done directly by calling the package manager (rpm/dpkg), but by using a frontend (yum/apt-get). For this domain type, the policy contains a rule allowing execution of this frontend and a type transition rule forcing this program to run under its own domain after being started. The available repositories only contain signed packages, and repository changes are not permitted. New repositories could be added by modifying the frontend's configuration, but the administrator has no write access to these files. Furthermore, unsigned binaries could be installed using the actual package manager (rpm or dpkg), but there exists no transition rules for these package managers – only for the frontend – and the installation would not succeed.

System parameters are usually changed using the `sysctl` tool, but neither SELinux nor this tool does permit granting access to selected parameters in a fine-grained way. Some parameters can be used maliciously by administrators to compromise the integrity of the platform. Therefore, we construct a privileged wrapper for sysctl, which allows access to selected system parameters, that can be invoked by the administrator instead of the original sysctl tool. This is realized by giving the wrapper access to the *sysctl_t* type and specifying a transition from the user's domain type to the wrapper's domain type.

Black-listed write access ($P_{write_\bullet}$): The permisssions of this role are based on the unconfined SELinux domain, whereby a process is permitted all operations without any restrictions. Since SELinux's policy description language does not provide deny rules, we essentially write very broad allow rules on all type except a few protected types corresponding to the kernel, boot process, maintenance agent and SELinux, for which we specify a *protected_type* attribute. Since the types we are associating with the protected attribute could have allow rules defined for other types, this user domain contains no transition rules. To illustrate this with an example, a yum process ran by an administrator won't transition into yum's privileged domain type (which would have access, for example, to the kernel files).

Full access (P_∞): The user will be assigned to SELinux's unconfined role. This allows unrestricted access to the administrator, while still keeping the running services confined.

7.4.4 Privilege Elevation

By default administrators have no privileges ($P_\emptyset$). Privileges can then be gradually elevated. The maintenance agent allows at most one administrator at the time to perform maintenance on a compute node, in order to prevent potential conflicts in the integrity recovery, as well as auditing concerns. Figure 7.4 illustrates the overall elevation process for the different privilege levels. In this section we will discuss the generic steps involved in the elevation of privileges. Furthermore, we will explain the special transition between the $P_{write_\circ}$ and $P_{write_\bullet}$ privilege levels, which has to cope with potential impact on the cloud users' security objectives.

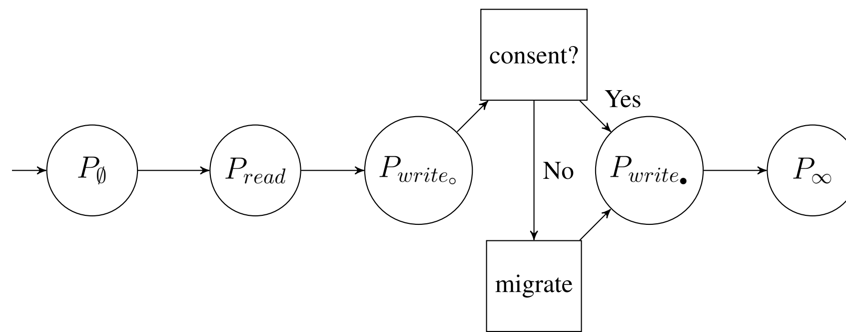

Figure 7.4: Overview of Privilege Elevation Process

Generic Elevation The *Maintenance Agent* provides an interface that allows an administrator to elevate his privileges from the current level to the next one. The ordering of privilege levels is shown in the chain-like illustration of Figure 7.4. Elevating privileges means that the administrative user is assigned to an SELinux user type, which is associated with the role type in the policy enforcement system that possesses the required privileges. This assignment to roles is generally sufficient for elevating to the next privilege level except for the transition from $P_{write_\circ}$ to $P_{write_\bullet}$.

User Consent and Virtual Machine Migration ($P_{write_\circ} \rightarrow P_{write_\bullet}$) In the case of the transition between $P_{write_\circ}$ and $P_{write_\bullet}$, we have to introduce additional mechanisms during the elevation, in order to cope with the potential impact on the cloud users' security objectives. We introduce the principle of a user consent that allows cloud users, who have virtual machines running on that particular compute node, to assess the impact of a privilege elevation and provide a consent that their virtual machines remain on the host. Imagine a scenario that a cloud user experiences problems with the cloud infrastructure and the problem only arises in combination with his workload. The user can consider to give their consent that an administrator gains $P_{write_\bullet}$ privileges, while the virtual machines remain on the host, for trouble-shooting purposes.

We envision that the user consent can be obtained by the maintenance agent in the form of sending an email to all the users served on the particular compute node. The email contains a link to a webservice call at the cloud provider, which requires user authentication, that would redirect the user consent to the maintenance agent. Incorporating a nonce, i.e., a random value, in the link would guarantee the freshness of the user consent. In order to have a more efficient consent process, the maintenance agent and the cloud user can consider the *Virtual Machine Security Label* (cf. Section 7.3.2): for *Low* implicit consent is given, for *Medium* either the security team or an automated decision at the cloud user side can be done, and for *High* a human administrator has to make the decision.

In the case that a user does not provide their consent, e.g., due to explicit deny or due to time-out, the corresponding virtual machines are securely migrated to a different compute node. The migration has to contact the *Database of Trustworthiness of Compute Nodes* (cf. Section 7.3.2), in order to decide on the selection of the target compute node. Furthermore, the maintenance agent has to record the untrustworthiness of the current compute node for the particular user in the database. Remaining traces of a virtual machine, i.e., swap storage and access to shared storage, have to be removed. For performance reasons, compute nodes have swapping disabled, therefore no traces are left in swap storage. Access to shared storage need to be revoked by the maintenance agent.

Limiting Privilege Exhaustion We are following the least privilege principle for the maintenance tasks, i.e., an administrator should only obtain the privileges required for a specific task. Therefore, administrators start in $P_\emptyset$ and have to gradually increase their privileges; instead of elevating privileges immediately from $P_\emptyset$ to P_∞. Furthermore, a barrier for the elevation needs to be introduced, in order to prevent gradual but immediate elevation to P_∞. This barrier can be implemented as a time delay, i.e., the administrator has to wait before the next elevation, obtaining approval from another administrator, or by audit logs (with each administrator logging a reason for the requested privilege).

7.4.5 Privilege Revocation and Recovery

Revocation of privileges is the counterpart to the elevation described previously. However, privilege revocation alone is not sufficient for guaranteed return to an initial trusted state once a maintenance task is completed. Our system has to assess the modifications performed during the maintenance task and act on these modifications based on the recovery strategy selected by the administrator. The administrator can decide between *rollback* and *commit*. Rollback dismisses all modifications to the system performed during the maintenance. These modifications can affect the state of the machine (processes, system parameters) and data at rest (files on disk). On a contrary, commit allows to keep the modifications, but they have to be approved by the cloud users and/or the security team using a consent process.

Figure 7.5 illustrates the overall revocation and recovery process when revoking privileges from $P = \{P_{read}, P_{write_\circ}, P_{write_\bullet}, P_\infty\}$ back to $P_\emptyset$. The *rollback* recovery eventually leads back to a trusted state. Recovery using *commit* however might lead to an untrusted state when the user consent is not obtained. For $P_{write_\bullet}$ and P_∞ we can only perform a partial commit recovery (illustrated by the dashed edges), because modifications of the system state cannot be detected under the influence of such a high privilege access. The revocation is performed analogously to *Generic Elevation* in Section 7.4.4, i.e., the administrative user is assigned to the role of privilege level $P_\emptyset$ using the maintenance agent. In the following we describe the recovery process.

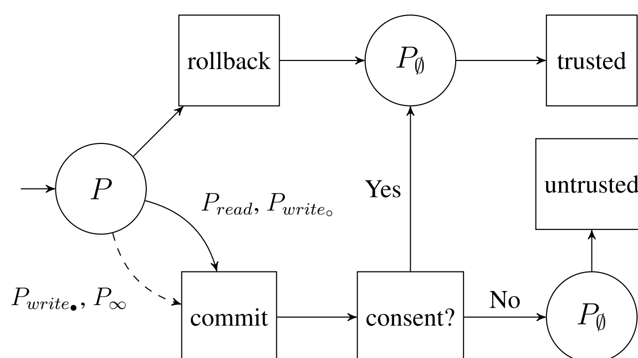

Figure 7.5: Overview of Revocation and Recovery Process

Preparing Recovery during Privilege Elevation: Before elevating to privilege levels that have write access, i.e., $P_{write_\circ}$, $P_{write_\bullet}$, and P_∞, the maintenance agent has to initiate the creation of a snapshot of the system. This snapshot covers data at rest (file system snapshot) and the system state (processes, system parameters). For $P_{write_\circ}$ and $P_{write_\bullet}$ one can create a system snapshot locally that will be protected by the policy enforcement. A snapshot of the file system can be created using the logical volume manager (LVM) or snapshot functionality in modern

filesystems, such as Btrfs[3]. We also have to take a snapshot of the run-time system parameters (e.g., using `sysctl`). In the case of P_∞, where the administrator has full access and the snapshots cannot be protected locally anymore, we either perform the snapshotting remotely or use the more expensive recovery strategy of re-provisioning. Remote snapshot requires that compute nodes are using an external storage provider.

Recovery from Read-only Access (P_{read}): Read-only access with program execution privilege only leads to modifications of the system state, i.e., running processes. This could impact the availability of the compute node, e.g., due to a started process with high resource requirements, therefore the recovery mechanism is needed. The maintenance agent identifies programs started by the administrator based on the process' security context, where the user equals the administrator. *Rollback:* Processes started by the administrator are killed. *Commit:* Processes started by the administrator need to be verified. The following information is sent to the verifiers: hash of the binary, binary for download, program arguments.

Recovery from White-listed Write Access ($P_{write_\circ}$): Using the snapshot of the file system, we generate a diff between the current file system and the snapshot. Furthermore, we generate a diff between the system parameters using the system snapshot. *Rollback:* Changed files are replaced by their copies from the snapshot. Using the transaction log of the package manager, we can rollback all modified packages. System parameters are restored with the values recorded in the snapshot. Recovery from started programs is also applied (cf. P_{read} recovery). If there exists changes in configuration files, we use the packages meta-data, i.e., which files belong to a package, to identify the corresponding service and restart it after reverting the configuration file. *Commit:* Updates to files and system parameters as well as the transaction log of the package manager are send to the verifier for review. The *commit* from P_{read} recovery also applies.

Recovery from Black-listed Write Access ($P_{write_\bullet}$): Recovering from $P_{write_\bullet}$ is similar to the $P_{write_\circ}$ recovery. However the system state cannot be assessed, because arbitrary changes to binaries are possible, e.g., modifying processes in memory. *Rollback:* Same file system rollback as shown for $P_{write_\circ}$. A reboot is required to rollback the changes in the state. *Commit:* Same file system commit as shown for $P_{write_\circ}$. A commit for the system state is very challenging and practically not feasible, because we would need to detect in-memory modifications of processes. Therefore a reboot is required.

Recovery from Full Access (P_∞): The recovery from P_∞ is the most expensive operation, because of the large possible impact of this high privileged access. Two possible strategies are available: recovery using remote file system snapshots or re-provisioning. *Rollback:* A reboot is required to rollback the system state. For file system rollback, we either revert from a remote snapshot or re-provision the node. *Commit:* In the case of remote snapshots, a file system diff can be computed on the storage node and send to the cloud users. Otherwise, *commit* is not available for the file system. It is also not available for the system state (cf. $P_{write_\bullet}$).

Returning to a Trusted State: The objective of the revocation and recovery process is to return from the maintenance mode back to an operational and trusted state. Figure 7.5 illustrates that the rollback recovery strategy leads to a trusted state after the privileges are revoked back to $P_\emptyset$. Alternatively, the commit recovery involves a user consent approach similar to the one introduced in the elevation (cf. Section 7.4.4). Each user that is or was hosted on that particular compute node has to approve the modifications performed during the maintenance task. Depending on the outcome of the consent process, each user decides if the system is in a trusted or untrusted state from their perspective. In the case that there still exist virtual machines, which belong to a user who did not give consent, these VMs have to be migrated away. Overall, this

[3] https://btrfs.wiki.kernel.org/index.php/Main_Page

might lead to a scenario where a compute node is both in a trusted and untrusted state for multiple users. This trust fragmentation of the system degrades its efficiency. Therefore the security team should employ a "garbage collection" process, where administrators are asked to either integrate their changes into an official server template or the affected nodes are re-provisioned.

In the case that a trusted state is reached, the trustworthiness of the compute node is updated for the users that provided consent in the trustworthiness database. When the system recovers from $P_{write_\bullet}$ or P_∞, the virtual machines that were migrated away during elevation are migrated back. In the case of an untrusted state, this compute node remains unusable for the affected users.

7.5 Security Discussion

We now discuss the actual security achieved by our system. We aim at satisfying the security objectives described in Section 7.2.2 based on the mechanisms described in Section 7.4. Our focus is to prevent mis-use of obtained privileges. We do not consider attacks through physical attacks, platform vulnerabilities, and covert channels since both are orthogonal to our approach.

We discuss how each privilege level may impact the security goals of integrity, availability and confidentiality, and how we ensure that the impact is mitigated by our system. $P_\emptyset$ is the trivial case with no privileges, therefore no impact on the security goals exists.

7.5.1 Integrity

The first objective is integrity. This includes integrity of all resources (memory, disk, networks) as well as integrity of the computing platform. However, in our approach we exclude integrity of network resources and the cloud user should employ other protection means, such as Virtual Private Networks.

Read-only access (P_{read}) prohibits write modifications and the integrity of the platform and resources is ensured. Furthermore, we prevent read access to users related data that might contain other forms of access credentials with write privileges. However, the administrator is allowed to execute programs that have to be terminated once the maintenance is completed.

White-listed write access ($P_{write_\circ}$) is tightly controlled and limited to modifications of white-listed files and system parameters, and software installations. The integrity can be restored in the rollback process by replacing modified files with copies from a snapshot, restore system parameters to previous values, and rollback all software installations using a package manager transaction log. The integrity of the snapshot and the transaction log is ensured by the policy enforcement system. Modifying system parameters could amplify the risk of software vulnerabilities due to the disablement of protection mechanisms. However, we are not concerned about software vulnerabilities in our approach.

Black-listed write access ($P_{write_\bullet}$) allows any modifications except of vital system and security components, namely, bootloader, kernel, policy enforcement, maintenance agent, snapshots, package transaction logs, and dangerous system parameters. The integrity of user related data cannot be ensured at this privilege level anymore, therefore the workload is either migrated away or the user accepts the risk by giving consent. Migration has ensured that all traces of the workloads are removed, i.e., swap files are disabled which could contain old memory pages and access to shared storage is centrally disabled. Overall it is crucial that the administrator cannot modify the policy enforcement, which we ensure by prohibited access to any SELinux related data. Integrity of the platform's file system can be restored by the rollback process (cf. $P_{write_\circ}$),

but it requires a correct and thereby protected maintenance agent as well as protected snapshots and transaction logs. The state of the system can be arbitrarily modified, e.g., by modifying programs directly in memory, therefore a reboot ensures a rollback of the system state. Since the bootloader and kernel are protected, a reboot will lead to a trusted state again.

Full access (P_∞) allows any modifications to the platform. User data was either migrated away or the user accepted the risk before entering $P_{write_\bullet}$. Platform integrity can only be restored by reboot and provisioning or reverting of remote snapshots. We are aware of potential attacks that could circumvent the reprovisioning process, i.e., stealth and persistent malware such as an attack demonstrated by Wojtczuk and Rutkowska [WR09]. However, these are out of scope for our approach and fall in the category of platform vulnerabilities.

7.5.2 Confidentiality

Confidentiality requires that an administrator cannot see payload data of the users of the virtual infrastructure. This includes memory, disk and snapshots of past disk contents, processor state, log-files, and network traffic. A pre-condition of confidentiality is the integrity of the platform; if an administrator can modify the platform, then it may remove policy enforcement mechanisms.

Read-only access (P_{read}) is restricted by prohibiting access to cloud users related data, which have their own specific label in SELinux. This includes the processor state and memory, as it is represented as a process with the SELinux label, as well as disks that are stored as labeled files. Similar to the integrity discussion of P_{read}, the risk of successful exploitation of software vulnerabilities could be increased by being able to read certain system parameters, e.g., reading memory maps of processes could defeat random address space layouts (ASLR[4]). In our approach, we do not consider the confidentiality of network traffic. Furthermore, log files do not contain sensitive information of the cloud users.

White-listed write access ($P_{write_\circ}$) has the same impact on confidentiality as P_{read}, which is covered by our approach.

Black-listed write access ($P_{write_\bullet}$) has potential impact on the confidentiality of cloud users data due to the possibility of platform changes. Therefore, the same measures apply as used for the integrity protection, namely, virtual machine migration or user consent.

Full access (P_∞) allows any disclosure of information, but user related data was either migrated away or consent was given.

7.5.3 Availability

The availability of a compute node can only be negatively influenced by an administrator when modifying the system state or the file system. For example, a process can be started that consumes all the resources on the node or modifications causes problems on the platform. Therefore, availability is tightly connected to platform integrity, and the integrity rollback will also ensure that availability returns to the prior level once maintenance is completed.

[4] http://pax.grsecurity.net/docs/aslr.txt

7.6 Implementation & Evaluation

Our implementation is based on the open-source cloud platform *OpenStack*[5]. In our test environment we have two compute nodes and one storage node; all running CentOS 6 as virtual machines on a VMware ESXi. The shared storage node served as an NFS server for instance storage on the compute nodes, which is a prerequisite for live migration support[6].

Implementation of the Maintenance Agent: The maintenance agent is written in 200 lines of Python and has a RESTful API for providing the maintenance functionality. For the interaction with SELinux, e.g., for role assignment of users, we are leveraging the SELinux's Python interface. We are highlighting the implementation details for performing a commit operation with user consent. In order to assess the modifications, the agent performs the following operations: compute a diff between the sysctl settings before and after the maintenance; generate a list of filesystem changes by running `rsync` against the snapshot: for binaries we provide a hash, for non-binaries a text diff; determine programs started by the administrator by iterating /proc and looking for pids with the inherited administrator's user label; generate a list of package changes using yum's internal transaction log. The assessed modifications are collected in an email that is sent to the compute node users with a link to the agent's web interface for approval or rejection.

Performance Evaluation: We are now evaluating the performance for executing the transition from the privilege level $P_{write_\circ}$ to $P_{write_\bullet}$. In particular, we are considering that upon elevation all virtual machines are migrated away and upon revocation all filesystem changes are discarded. We are measuring the time for performing the live migration, rollback of filesystem changes, and assigning an administrator to a new privilege level. *Live migration* of virtual machines with each 512 MB of RAM using a 1.2 GBit/s network took about: 6.7s (1 VM), 12.4s (2 VMs), 22.8s (4 VMs). We observe a linear relationship between number of VMs and migration time. *Snapshotting and rollback:* Creating a snapshot of a 14 GB volume using btrfs requires 0.3 seconds, and deletion 0.1 seconds. It takes 1.1 minutes to run rsync with checksumming enabled to compare the running system against the snapshot. Instead of using rsync, modern filesystems provide an efficient diff command. However in the case of Btrfs, the find-new command is incomplete and does not show deleted files. *Assignment to Privilege Levels:* We measured assigning a user to $P_\emptyset$ and on average it took 4.5 seconds.

7.7 Related Work

This report focuses on the security of infrastructure clouds. We build on related work from several areas: Virtual systems security aims at reducing the security risks introduced by virtual machines, network, and storage. Trusted computing enables stakeholders to verify the integrity of given IT systems. We focus on linux-based virtual machine monitors. As a consequence, a final area of related work are Linux security policies and their enforcement.

Infrastructure Cloud Security: The first area of related work is security of virtual machine monitors. Virtual systems introduce several new security challenges [GR05b]. This knowledge is needed to underpin the user's individual decisions whether to trust a given component. Analysis of well-known attacks such as jailbreaks [Woj08] allows one to detect vulnerable configurations. This includes information leakage vulnerabilities of today's infrastructure clouds

[5] http://www.openstack.org
[6] http://wiki.openstack.org/LiveMigrationUsage

that allow covert or overt communication between multiple tenants that should be isolated. Examples include co-hosting validation [RTSS09] and cache-based side channels [Aci07, Per05]. While these vulnerabilities are important in practice, they are orthogonal to our approach. Our goal is to prevent insiders from obtaining additional means of attack. We accept the fact that insiders can act as end-users and exploit potential vulnerabilities to attack a given system.

SELinux in Infrastructure Clouds: SELinux is used in many projects to provide fine grained access control: it is for example included by default in the widely used Red Hat Enterprise Linux distribution, mostly providing an additional sandboxing layer for various services that can be run on a Linux server. As such, it is not surprising that it SELinux and other MAC technologies are used in infrastructure clouds [BS10, KGP+11], mostly to enforce isolation of resources that are shared among cloud users (multi-tenancy). In contrast, we use SELinux for restricting administrative privileges.

Trusted Computing and Virtual Infrastructures: In our context, trusted computing shall be used to validate the virtualisation platform. This means that we do not need virtualised TPMs [BCG+06] but rather validation of the core virtualisation platform. For Linux-based virtualization platforms such as KVM and Xen, important pieces of related work are [MSMW03, MSWM03, MSW+04b] where a software architecture based on Linux is proposed that provides attestation of all executables and configuration files. Another approach to use trusted computing for verifying virtual infrastructures has been proposed in [GPC+03] where tamper-proof hardware is virtualised to allow for multiple concurrent yet isolated protected VMs. In [SGR09] the authors have sketched how to use trusted computing for validating a cloud infrastructure. While each of these approaches provides certain assurance to the end users, none provides a concept for securing the maintenance of such clouds. As a consequence, they are useful during normal operations but only manage to declare a system untrusted once it is maintained.

7.8 Conclusions and Future Work

In this chapter we have resolved the threat of insider attacks by remote administrators in dark datacenters. In practice, this is the most important type of insider attacks. Unlike other approaches, ours is applicable to commodity cloud infrastructures such as OpenStack or OpenNebula. In contrast to existing security concepts for clouds, our approach includes maintenance, does not require special hardware, and does not have significant impact on the efficiency of the infrastructure cloud.

While we addressed the challenge for compute nodes, some open questions remain. The first is to develop a similar concept for storage nodes and their administrators. Our concept uses storage nodes as a building block without elaborating how to securely maintain them. In particular, protecting also storage nodes against inside attackers would be desirable. A second area of further investigation is to further validate our concept. This includes evaluation in large-scale practical field studies as well as a formal evaluation of the SElinux policies similar to [JSZ03].

Chapter 8

Self Managed Services

Chapter Authors:
Imad Abbadi (OXFD)

8.1 Introduction

NIST define Cloud as *'a model for enabling 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'*[MG09b]. In this chapter we focus on one aspect of NIST definition which is about providing automated resource management (described by NIST as *minimal management effort or service provider interaction*). We call it self-managed services, which are software services providing Cloud computing environment with automated capabilities, e.g. availability, reliability, and resilience. Such automation of management are based on many static and dynamic factors including Cloud user properties and Cloud infrastructure properties, which we discuss in the chapter. The main objective of this chapter is to explore such automated services and their interdependency.

Moving current Cloud infrastructure to the potential trustworthy Internet scale Cloud critical infrastructure requires a set of trustworthy middleware. Middleware glues member resources in Cloud layers together by providing a set of automated self-managed services that consider users' security and privacy requirements by design. These services should be transparent to Cloud users and should require minimal human intervention. The implementation of self-managed services' functions in middleware would mainly depend on the middleware location within Cloud's layers.

For clarity we mainly focus in this chapter on automated self-managed services' functions and their interdependency at *Virtual Layer*. We have previously discussed self-managed services at application layer ([Abb]). Self-managed services are not about autonomic computing [IBM01]. Autonomic computing is concerned about providing self-management for *Physical Layer* (e.g. physical servers and storage) and it does not change dynamically based on changes in the requirements of end-users.

Some work on service automation in general has already been done, as in the case of automated resource management [Ora11b], real application server [Ora11a], clustering technology [Ora10], and virtual resources management [Mic10, VMw10]. However, such work is still preliminary and is not enough considering Cloud complexity and potential future. Current work on service automation solves simple problems, e.g. restarting a process on failure, relocating applications on node failure, and preliminary resource management on recovery. Such work

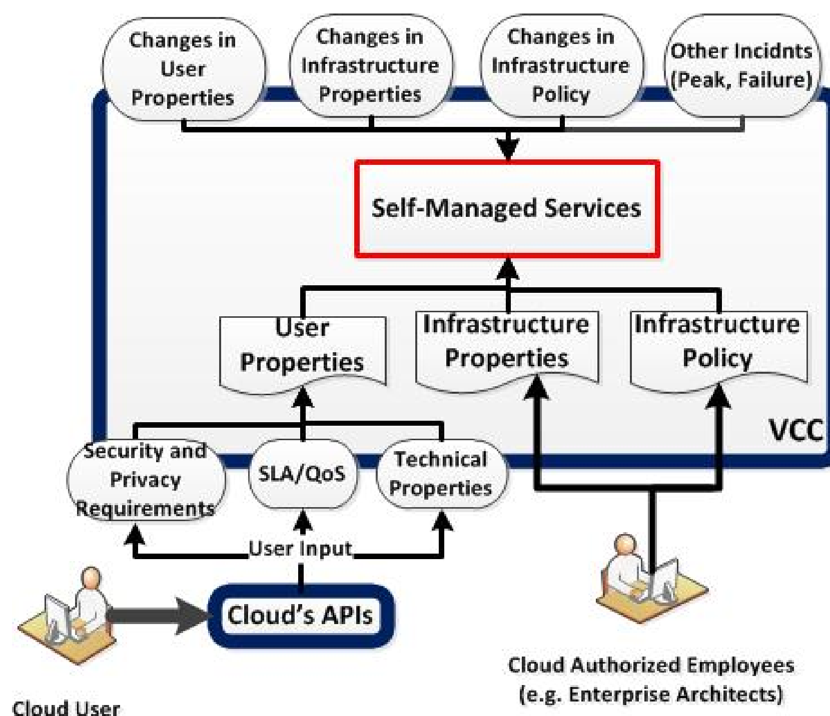

Figure 8.1: Factors Affecting Self-Managed Services Behavior

does not go far beyond that, as it focuses on specific area, does not look at the overall picture, and does not focus on finding the root cause of problems. The main reason behind not focusing on fully automated self-managed service is the complexity of technology, which is even much more complicated in Clouds' environment because of Clouds' heterogeneous nature.

In this chapter we extend our previous work ([Abb11a, Abb11b, Abb11c]). In ([Abb11a]) we propose a novel Cloud taxonomy focusing on Cloud infrastructure management that helps us to derive Cloud properties and identify the required self-managed services. In ([Abb11c]) we motivated the need for self-managed services, and for space limitations, we identify it as future research. This chapter builds on our extended abstract defining self-managed services ([Abb11b]). Specifically, we extend the definition of services' functions and their interdependencies in next section. We then discuss the challenges and requirements for managing self-managed services.

8.2 Self-Managed Services at Virtual Layer

In this section we outline the factors that affect managed services decisions, provide a conceptual model for functions required to support self-managed services, and discuss services' interdependency.

8.2.1 Factors Affecting Management Services

In this sub-section we briefly summarize part of our previous work on Virtual Control Center (VCC) [Abb11a, Abb11b]. VCC is a Cloud specific device that manages virtual resources and their interaction with physical resources using a set of software agents. Currently there are many tools for managing virtual resources, e.g. vCenter [VMw10] and OpenStack [Ope10b].

However, such tools have many security vulnerabilities ([Abb11c]) and only provide limited automated management services, which we discuss next in this section. We now summarize the factors, which would affect decisions made by self-managed services.

User Properties (Dynamic Properties) — A Cloud user interacts with the Cloud provider via Cloud webpage and supplied APIs. This enables users to define *user properties*. User properties for potential Cloud should cover technical properties (e.g. VMs, storage), QoS/SLA requirements (e.g. system availability, reliability measures, and lower/upper resource limits), and security and privacy requirements (e.g. location of data distribution and processing). These requirements might be more or less complicated based on the user type.

Infrastructure Properties (Static Properties) — Clouds' physical infrastructure is very well organized and managed by multiple parties, e.g. enterprise architects. Those people define the physical infrastructure properties, which should cover: components' reliability and connectivity, components distribution across Cloud infrastructure (e.g. how far components are from each other), redundancy types, servers clustering and grouping, network speed, etc.

Infrastructure Policy — Policies are defined by authorized employees to control the behaviours of self-managed services.

Changes and Incidents — These represent changes in: user properties, infrastructure properties, infrastructure policy, and other changes (e.g. increase/decrease system load, component failure, and network failure).

Management services should manage Cloud *Virtual Layer* by automatically finding the best match of user properties with infrastructure properties that considers infrastructure policy.

8.2.2 Functions of Self-Managed Services

In this sub-section we discuss the generic functions of self-managed services.

Adaptability — Figure 8.2 provides a conceptual model of *Adaptability* function. In the context of this chapter this function is concerned about adapting virtual resources which are hosted at the virtual layer to *Changes* and *Incidents*. The *Changes* include: changes in user properties, and changes in infrastructure properties (i.e. changes at physical layer's resources which host the user virtual resources). Example of *Incidents* include: physical or virtual resource failure, and increase in service demand. Such *Changes* and *Incidents* must *Maintain* the overall service security and privacy properties as agreed with customers. For example, i) adding/removing a VM to a virtual domain should not compromise the virtual domain security or integrity; and ii) removing a physical storage from physical domain should not reveal content confidentiality, e.g. storing an unprotected VM image at physical storage.

The adaptability function should automatically decide on an *Action* and *Cascaded Action* plans, which are either performed by the function itself or delegated to other processes. For example, when a group of VMs within a virtual domain needs more resources, the adaptability function first checks if the group is authorized to scale its resources up by adding another VM or increase resources allocated to an existing VM. If the group is authorized, the adaptability function identifies other resources that might be affected by such scaling (i.e. VMs member of the same domain, and virtual domains member of the same virtual collaborating domain). It then coordinates with the identified resources and decides if they should scale up and by how much they would need to scale up. Once everything is coordinated and planned, the adaptability function *Triggers* the scalability function to increase resources, as explained latter.

The adaptability function consults with the system architect function before taking an action. This is to ensure that the action plan does not have an impact on system properties.

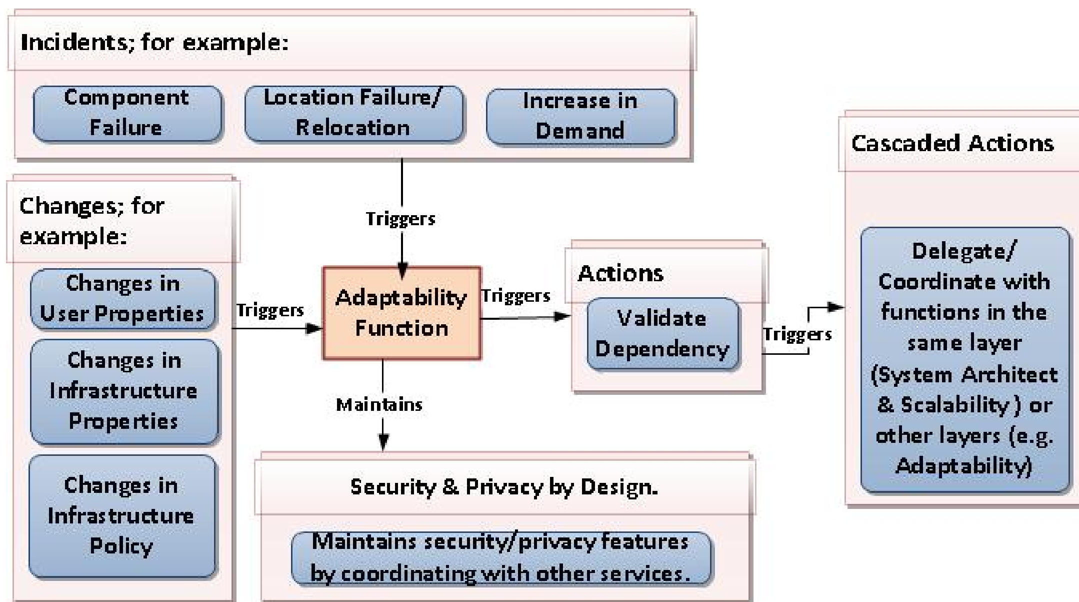

Figure 8.2: Adaptability Function

System Architect — Figure 8.3 provides a conceptual model of the *System Architect* function. This function resembles enterprise architect professionals. It is an automated function that receives *Triggers* from either the adaptability function (e.g. on incident) or from a Cloud user through a supplied APIs (e.g. a new hosting request). The System Architect in both cases should provide an architecture to the virtual layer. Such an architecture should be based on *Input* from both user properties and infrastructure properties. The system architect function provides a *Resilient Design* that is *Deployed By* the resilient function. The *Resilient Design* includes automatically selecting individual resources from the physical layer. Such selection should be based on the following: i) resources reliability, ii) type of redundancy/replication (e.g. RAID 1+0, RAID 5, dual channel), and iii) resources distribution, grouping and management across Cloud infrastructure. This process, i.e. the resilient system architect, generates well crafted process management scripts and documents.

The *System Architect* should consider user security and privacy requirements by design. For example, if the physical domain could not serve a virtual domain for any reason (e.g. network failure), in this case the *System Architect* function must ensure that the updated architecture does not compromise user properties. The *System Architect* should also lessen the effects of DoS attack by providing resilient and scalable design.

Resilience — Figure 8.3 provides a conceptual model of *Resilience* function. This function resembles system administrators, who *Deploys* the *Resilient Design* at the virtual layer. The *Resilient Design* is provided by the System Architect process in two cases: the first is when producing an architecture for a new service request, and the second when updating an already exiting architecture. The *Resilience* function communicates with other resources (e.g. physical servers' VMM) and/or other management tools (e.g. VCC) to *Deploy* the *Resilient Design*. The resilience function is also in charge of communicating failures of a resource to other services. This is performed by *Triggering Cascaded Actions*; e.g. on failure it might trigger the *Availability* function to divert traffic to other available routes. The *Resilience* function should also *Maintain* security and privacy by design, e.g. *Resilience* function should consider the hosting of virtual resources at physical domains which are not geographically located within boundaries restricted by the user properties.

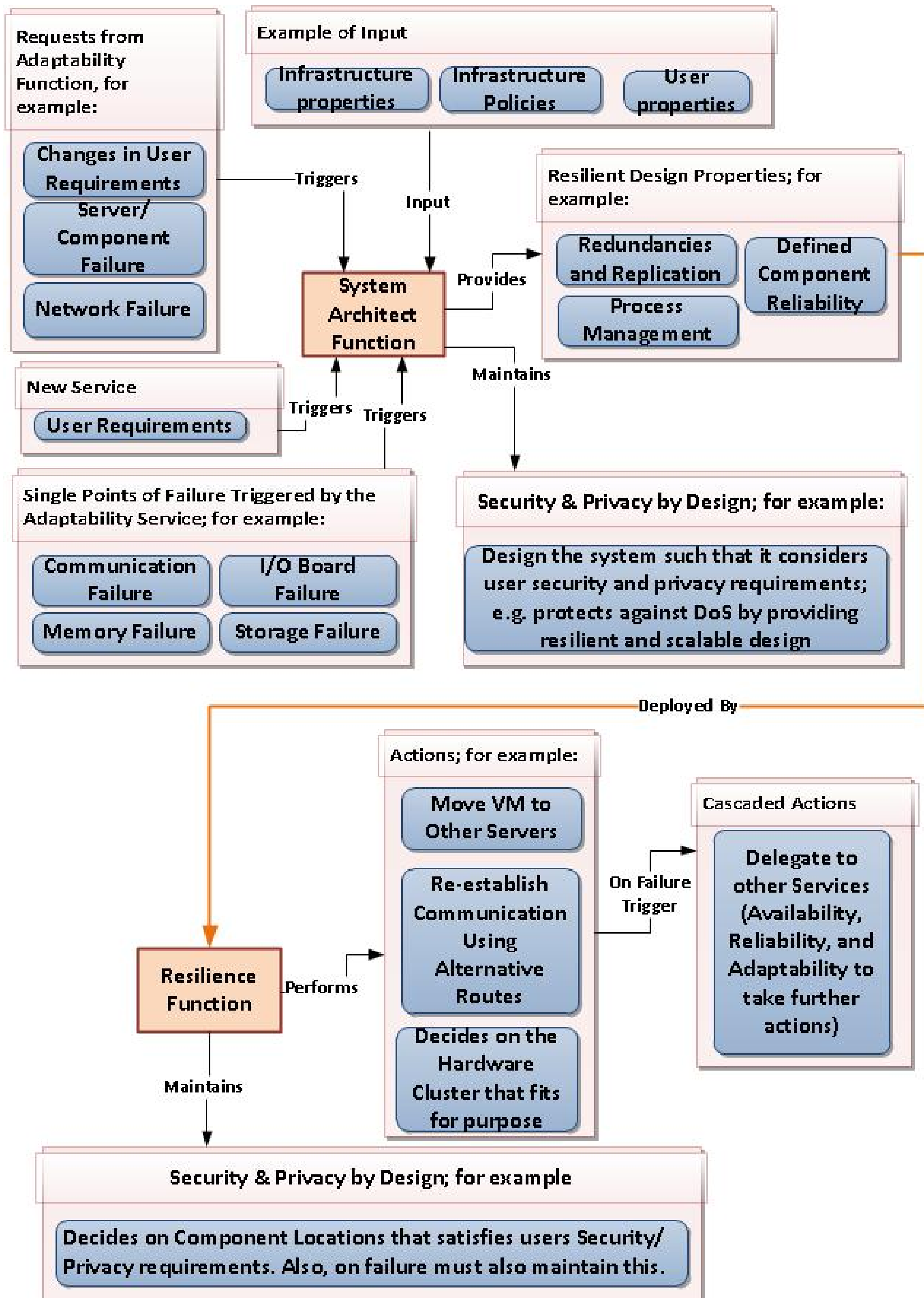

Figure 8.3: System Architect and Resilience Functions

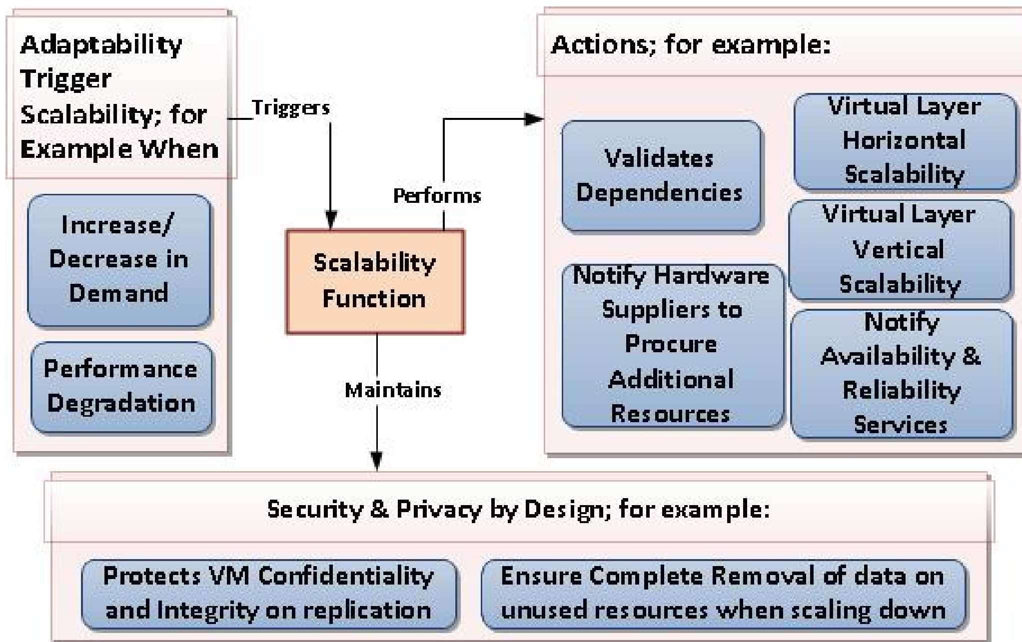

Figure 8.4: Scalability Function

Scalability — Figure 8.4 provides a conceptual model of the *Scalability* function. This function supports Cloud elasticity feature by scaling resources up and down when needed. The *Scalability* function is mainly *Triggered* by the Adaptability function when detecting a need for either adding resources or removing resources. *Scalability* function then *Performs Actions*. *Actions* include *Horizontal Scalability* by replicating VM resources and/or *Vertical Scalability* by increasing a VM resources. Before proceeding *Actions* should always validate user properties before scaling resources. The Scalability must *Notify* the Availability and Reliability functions when scaling up/down. Scalability should always *Maintain* security and privacy by design. For example, i) scalability must protects VM integrity and confidentiality when replicating a VM; ii) scalability should permanently remove data from released resources (e.g. remove data from storage) on downscaling; and iii) when performing horizontal scaling up the new virtual resources should be allocated based on user properties.

Availability — The *Availability* function is in charge of i) maintaining communication channels of available virtual services with resources at application layer and ii) distributing application layer requests evenly across available redundant virtual resources. Availability is supported by a correctly *Deployed Resilient Design*. The higher resilient a system the higher availability/reliability would be expected. Figure 8.5 provides a conceptual model for application *Availability* service. This Figure provides examples of *Incidents* from *Resilience* and *Changes* from *Scalability* service, which Triggers the *Availability* service. The *Availability* service in turn Performs *Actions* based on the *Incidents* and *Changes*. The *Actions* also Trigger *Cascaded Actions* to other services at both *Application Layer* and *Virtual Layer*. For example, if a channel is marked unusable by the *Resilience* function, the availability process immediately stops diverting traffic to that channel, and re-diverts the channel traffic to other active channels until the *Adaptability* function fixes the problem. Availability function should also consider security and privacy requirement by design. For example, it should maintain secure communication channels when distributing load, verifies the identity of communicating parties, and communicates securely with other services.

Reliability — This function is in charge of maintaining service reliability at virtual layer,

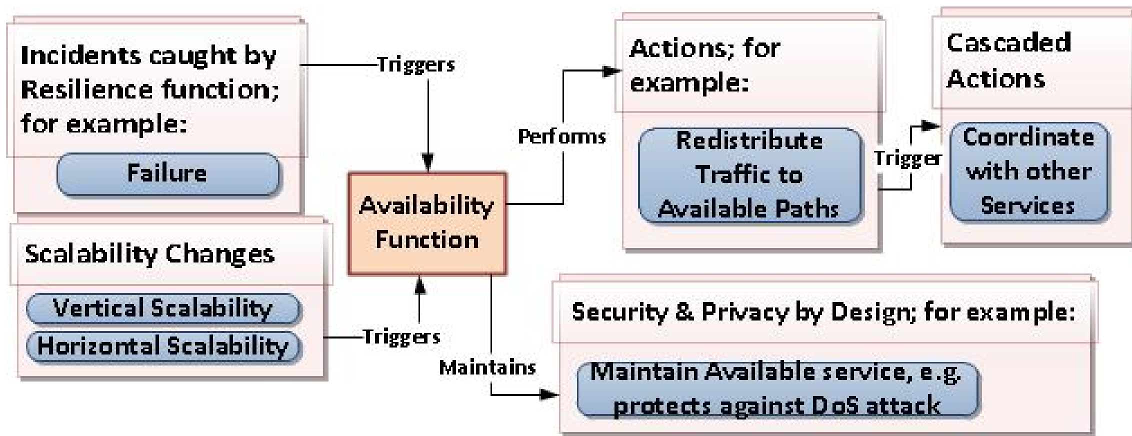

Figure 8.5: Availability Function

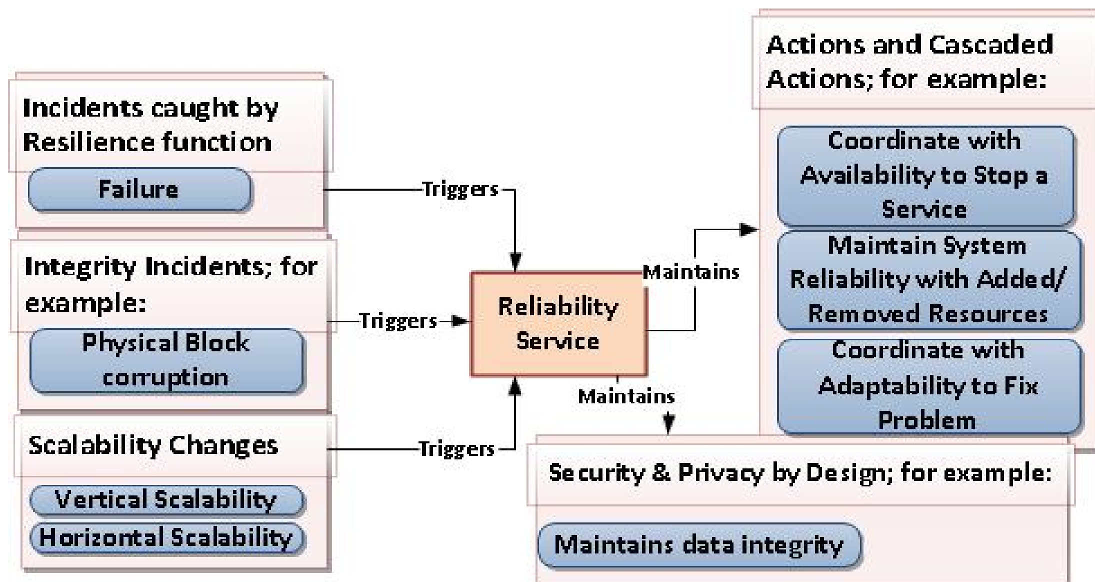

Figure 8.6: Service Reliability Function

which is of higher priority than service availability. Most importantly it ensures that the virtual service integrity is maintained (i.e. no data loss and correct service execution, as is conceptually illustrated in Figure 8.7). If service integrity is affected by anyway and cannot be immediately recovered, service reliability then notifies the availability service to immediately bring the service down. This is to ensure that data integrity is always protected. Simultaneously, *Adaptability* and *Resilience* functions should automatically attempt to recover the system, and notifies system administrators in case of a decision cannot be automatically made (e.g. data corruption that requires manual intervention by an expert administrator).

Figure 8.6 provides a conceptual model of *Reliability* service. This Figure provides examples of *Incidents* and *Changes*, which Triggers the *Reliability* service. The *Reliability* service in turn Performs *Actions* and *Cascaded Actions* based on the *Incidents* and *Changes*. It should also *Maintain* security and privacy by design.

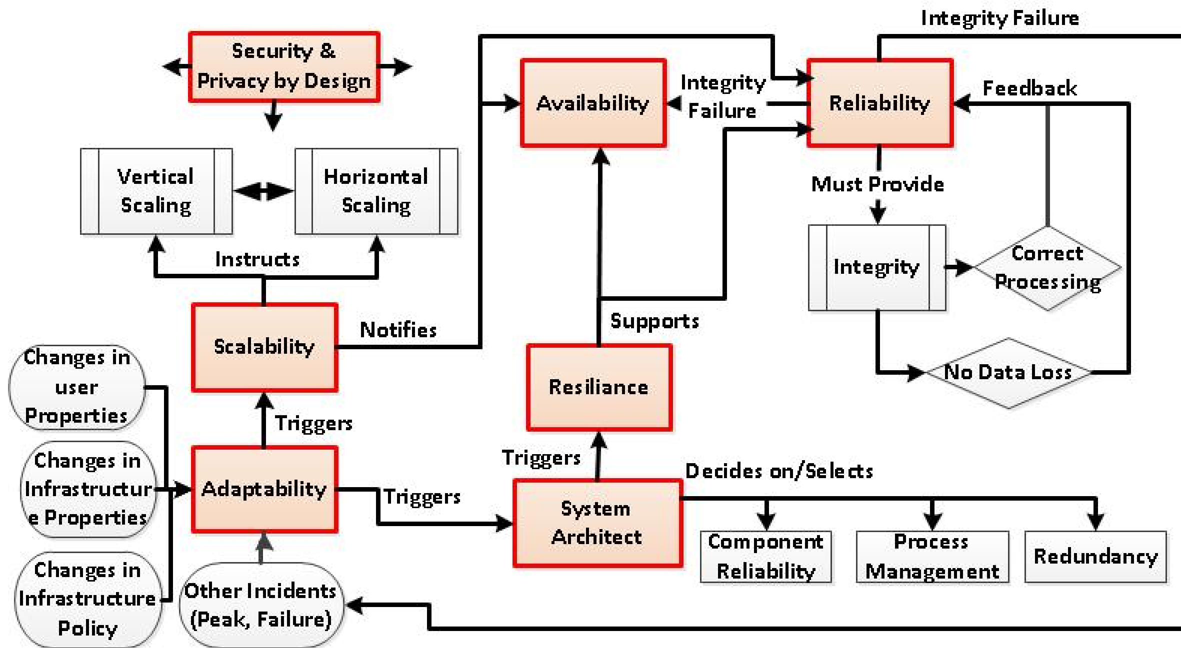

Figure 8.7: Self-Managed Services Interdependency

8.2.3 Services Interdependency

Figure 8.7 provides a summary for the interaction amongst Virtual Layer self-managed services, as we discuss throughout this section. This Figure provides a high level overview and it is meant not to cover deep details for clarity. In this Figure *Adaptability* function acts as the heart of self-managed services. For example, it intercepts *Incidents* and *Changes*, manages these by generating action plans. The *Adaptability* communicates with *System Architect*, *Scalability* and *Reliability* function to delegate part of the generated action plans.

The *System Architect* function provides resilient architecture by deciding on components reliability and redundancy. It then generates a *Resilient Design*. It *Triggers* the *Resilient* function to *Deploy* the *Resilient Design*. Excellent *Resilient Design* results in higher availability and reliability properties. This is indicated in Figure 8.7 using *Supports* relation between *Resilience* and *Availability*, and *Resilience* and *Reliability*. *Adaptability Triggers Scalability* to either do *Vertical Scaling* and/or *Horizontal Scaling*. Once scaling is done *Scalability Notifies Availability* and *Reliability* about the scaling. The *Reliability* function is linked with *Integrity* process using the "*Must Provide*" relation. The outcome of the *Integrity* process is fed to the *Reliability* function. If application integrity is affected the *Reliability* function sends an *Integrity Failure* to both the *Availability* and the *Adaptability* functions to take proper action.

8.3 Main Challenges and Requirements

In this section we discuss the challenges involved in providing trustworthy self-managed services by design. It is outside the scope of this chapter to discuss the challenges of implementing self-managed services (e.g. data replication). Specifically, we focus on discussing the challenges of providing security and privacy requirements for Cloud infrastructural services. We categorize such challenges into two parts: (a.) providing trustworthy management of services, and (b.) providing trustworthiness in the services. In our previous work ([Abb11c]) we have

identified part of the security challenges, but we did not discuss them in detail for space limitations and we left it as a planned future work.

For the first category (i.e. managing self-managed services), VCC, which is in charge of managing self-managed services, should be trusted to enforce infrastructure policy, and host user virtual resources at physical resources by considering both user properties and infrastructure properties. Chapter 5 discusses the challenges of managing self-managed services using VCC.

The second challenge which is about providing trustworthiness in self-managed services requires the following.

1. As we explained earlier Cloud's infrastructure is conceptually composed of several intersecting layers. Self-managed services should take into consideration the heterogeneous and complex layering, and the horizontal and vertical communication channels amongst these layers. Specifically, providing automated self-managed services for a resource requires: (a.) understanding the relative position of the resource, i.e. identifying the resource's horizontal layer, vertical layer, domain, and collaborating domain; (b.) what are the infrastructure properties associated with the physcial environment hosting a virtual resource (i.e. the properties associated with physical domain and collaborating domain which host virtual domain), (c.) what are the user properties associated with resource virtual domain and collaborating domain, (d.) how the management of the resource would affect other resources of the same domain, and (e.) how the management of the resource would affect other resources of the same collaborating domain.

2. Policy distribution, coordination, and management across Cloud entities is a big challenge considering Clouds' complex infrastructure.

3. Cloud infrastructure is not hosted at a single data centre that is located at a specific location; it is rather the opposite, as most likely it is distributed across distant data centres. This factor has a major impact on decisions being made by self-managed services for several reasons; for example, i) the distance and the communication medium between distant data centres will have an impact on data transfer speed, and ii) Cloud users might have security and privacy concerns on location of their data. Automated services must consider this important factor and other related factors (e.g. data volume, data access mode, etc) when providing a service.

4. Cloud-of-Cloud is a term that is used to refer to the collaboration of multiple Cloud providers to support dependable Cloud infrastructures; i.e. Cloud providers collaborate to help each other in enhancing self-managed services as in the case of higher resilience, reliability, scalability, and dependability. For example, if a Cloud provider has an urgency other Cloud providers can temporarily provide their unoccupied resources to support customers eliminating service failures. Self-managed services must consider the existence of Cloud-of-Clouds, and it must also be designed to enforce Cloud provider related policies when considering a decision to use other Cloud resources, as this would have a major impact on security, practicality and legislation related issues. Specifically, hosting user resources at another Cloud provider should be done only after ensuring user defined properties are enforced.

5. Key management is a fundamental requirement when discussing Cloud's security. This is especially the case of Cloud internal employees as they are considered insiders for Cloud users. Thus, content protection keys should not be accessible to Cloud employees.

We now summarize a set of high level requirements to mitigate the challenges identified above.

Trustworthy management of self-managed services requires the following: (1.) VCC and VMM should attest to each other execution environment, so that communicating entities can get the assurance of the security and reliability of VMMs and VCC; (2.) VCC and VMM need to have management agents that are trusted to behave as expected. These agents are in charge of implementing the self-managed services' functions and enforcing them at each component. Agents' trustworthiness must be assured to communicating parties; (3.) VCC and VMM should provide protected storage functions; (4.) VCC and VMM should be able to exchange each other identification certificates in a secure and authentic way; (5.) VCC needs to be resilient and scalable to provide distributed and coordinated services; and (6.) VCC should provide hardware trustworthiness mechanisms to prevent infrastructure single point of failure.

Providing Self-Managed Services require the following: (1.) A mechanism to attest to VMM trustworthiness to ensure that it would enforce user properties; (2.) A mechanism to communicate user properties across Cloud related components, and ensuring the properties are not tampered with whilst being transferred/executed/stored; (3.) Providing secure information sharing across Cloud components in the same layer and across multiple layers; (4.) Mitigating insider threats as discussed next; (5.) Standardization — Most technologies, which are used in Cloud, are not new; however, the Cloud heterogeneous nature requires reconsidering many issues, as in the case of standardization. For example, different software and hardware providers need to provide standard interfaces enabling cross communication between Cloud components; and (6.) Interoperability — This requirement is not only to avoid vendor lock-in, but also to enable collaborative efforts. For example, hypervisor and VMM interoperability enables VMs from different suppliers to work on hypervisors from different manufacturers. This in turn helps in supporting self-managed services.

The Cloud infrastructure must be capable of protecting the integrity, confidentiality and availability of Cloud critical data from Cloud insiders. This covers all types of data and communication messages whether directly related to Cloud users or used to manage internal resources, e.g. data stored inside VCC, VMM, and exchanged across Cloud entities. In our previous work ([Abb11c]) we identified the set of requirements for mitigating insider threats.

8.4 Conclusion

The complexity of Cloud infrastructure means a large number of subsystems have to work perfectly together to keep the operation running. In addition multiple and different groups need to cooperate, exchange critical messages and coordinate amongst themselves when taking a self-managed decision. Current Cloud computing does not provide the full potential of automated self-managed services, and relies on Cloud's employees to support the infrastructure. In this chapter we present a conceptual model of self-managed services and identify the factors, which affect services' decisions. This model helps in understanding the required functions and their interdependency when providing self-managed services in Cloud computing. Also, it helps in realizing the challenges involved in providing automated management functions. Finally, we discuss the challenges and requirements for managing and providing automated services security and privacy by design.

Chapter 9

Resource Efficient BFT

Chapter Authors:
Johannes Behl, Klaus Stengel (FAU)

9.1 Introduction

There is an ongoing process for replacing conventional infrastructure with computer-provided services accessible over the Internet. While this is convenient for users as these services are typically available around the clock, and also for companies as provision costs are reduced, our society increasingly depends on their well-functioning. This becomes clear once services fail or, sometimes even more alarming, provide faulty results to users.

Today, counter measures to faults applied in practice are almost purely dedicated to handle crash-stop failure, for example, by employing replication. Furthermore, specific techniques are used to selectively address non-crash faults (e. g., the detection of a bit-flip using a checksum). While the first handles typical hardware-induced problems (e. g., disk failures), the second is often dedicated to address specific problems that are considered as likely or had a severe negative impact in the past. What remains unconsidered is an arsenal of threats ranging from software bugs, intrusions, viruses to spurious hardware errors. To handle these arbitrary faults in a generic fashion, Byzantine fault tolerance (BFT) is required.

In the past, BFT has been considered to be of mainly theoretical interest but over the last few years this has changed due to numerous research efforts ranging from making Byzantine fault tolerance practical [CL99], over improving its performance [KD04, KAD+07], to efforts decreasing the number of demanded replicas to tolerate a certain number of errors [YMV+03, CNV04, WSV+11]. Lately, a debate has been started why industry, despite all this progress, is reluctant to actually utilize the provided research results [CMW+08, KR09]. From our point of view, economical reasons mainly dominated by high resource demand are still the key inhibitor for wide spread use of Byzantine fault tolerance.

In the pure sense, BFT requires at least $3f + 1$ replicas to tolerate f faults [CL99]. At the execution stage, the number of replicas can be reduced to $2f + 1$ [YMV+03] or even $f + 1$ [WSV+11] when the execution of requests is separated from the agreement on the execution order. However, in both cases, agreement still has to be performed by $3f + 1$ nodes, and in the second case, substantial parts of the system have to be trusted, lowering the assumption of BFT to some parts of the system. This hybrid fault model, in which some parts can fail arbitrarily whereas others can only fail by crashing, has also been explored to reduce the number of ordering as well as the number of execution replicas to $2f + 1$ [CNV04]. While some systems assume the trustworthiness of, for example, a hypervisor and parts of the communication

infrastructure, it has been shown that in principle this trusted part can be reduced to a secure log [CMSK07] or a special form of trusted counters [CLVV09] that prevent *equivocation*, that is the ability to make conflicting statements to different participants in a distributed protocol. Technically, these systems have come near the resource footprint of crash-stop driven solutions. However, the associated overhead is still too high to get widely adopted in industry.

As a consequence, this chapter presents CheapBFT, a protocol suite consisting of three sub protocols to handle Byzantine failures at a minimal possible resource footprint. By using a small trusted subsystem preventing equivocation, *PrimaryBackup* is the first protocol that requires only $f + 1$ active replicas at both the agreement stage and the execution stage during normal operation. In case of suspected or detected faults, we switch to *PlainBFT* by rapidly activating up to f additional nodes at both stages, which enable us to fall back to the minimal configuration of $2f + 1$ under attack. Furthermore, we present *BackupRotation*, an extension of PrimaryBackup that supports dynamic rotation at both stages, distributing the load for agreement and execution of the $f + 1$ active replicas over all $2f + 1$ nodes.

The remainder of this chapter is structured as follows: Section 9.3 explains the background of CheapBFT. Next, we outline our system model and provide a sketch of our architecture. Section 9.4 gives an overview of the basic setting of CheapBFT. Then, we detail PrimaryBackup (Section 9.5) that reduces agreement and execution to $f + 1$ active replicas and f additional replicas that witness progress. As an extension to PrimaryBackup, we introduce rotation in the context of BackupRotation in Section 9.6. Section 9.7 outlines PlainBFT to handle the case where $2f + 1$ are demanded. Finally, Section 9.8 details related approaches and Section 9.9 concludes.

9.2 Background

Our proposal of a resource-efficient Byzantine fault-tolerant system is based on a trusted subsystem that prevents equivocation, meaning the ability to make conflicting statements to different participants in a distributed protocol. We explain this by: Firstly, detailing the concept of a trusted subsystem; secondly, illustrating equivocation and how it can be inhibited and thirdly, by outlining a specific form of a trusted subsystem that we utilize in the context of CheapBFT for preventing equivocation.

9.2.1 Trusted Subsystems

The notion of a trusted subsystem assumes that hardware and software of a system can be subdivided in a trusted part and a non-trusted part. We refer to the trusted part as trusted subsystem since it is typically less complex and provides only a limited set of functionality compared to the remaining main system. The core idea of a trusted subsystem is that it is tamper proof, thus it cannot be manipulated even in the case of a malicious intruder gains root access to the non-trusted part. Depending on the security demands, various ways have been proposed to enforce the isolation and protection of a trusted subsystem from the main system, ranging from software-centric solutions (e. g., provided by a hypervisor) to hardware-driven variants relying for example on a trusted platform module (TPM) of a processor, a cryptographic coprocessors [ADDS91, SW99] or a smartcard. Hardware-based isolation is typically considered as more secure and can even be resistant to physical attacks. Initially used to secure financial transactions by supporting secure authentication and encryption [ADDS91], availability and use

of trusted subsystems is of more general nature nowadays (e. g., in form of TPM chips provided in current notebooks) [Pea02].

9.2.2 Preventing Equivocation

Recently, trusted subsystems gained rising interest in the domain of distributed systems for preventing equivocation. As stated before, equivocation characterizes the ability of a malicious party to make conflicting statements for a single action in a protocol with multiple participants. For example, in a Byzantine fault-tolerant agreement protocol like PBFT [CL99] there is a dedicated process, the leader, that proposes the order of messages. In case of a malicious leader, it can forward conflicting proposals to the other replicas ultimately leading to inconsistencies. While this is handled by the protocol, it requires one communication round in which every replica broadcasts the received proposal to ensure all correct nodes received the same proposal from the leader.

Chun et al. [CMSK07] proposed the notion of an attested append-only memory that basically introduces a trusted log that records messages transmitted in a protocol and can be remotely validated. Thus, in the aforementioned case of a leader sending conflicting proposals the participating replicas can detect this by consulting the log.[1] By focusing on inhibiting equivocation, it was possible to decrease the number of required replicas from $3f + 1$ as needed for BFT without a trusted subsystem to $2f + 1$ and avoid the aforementioned communication round with a much smaller and less complex trusted subsystem than previously proposed subsystem-based approaches [CNV04, RK07]. The idea of preventing equivocation using a trusted subsystem was later on generalized by Levin et al. [LDLM09] to the context of distributed protocols and conceptually simplified from a secure log to monotonically non-decreasing counters that are signed by the subsystem in conjunction with arbitrary data. Building the basis for preventing equivocation in CheapBFT, this is further detailed in the remainder of the section.

9.2.3 Trusted Signed Counter

To implement a trusted signed counter for preventing equivocation in CheapBFT, we require two operations supported by the subsystem: One that unforgeable binds data to a specific counter value and another one that securely validates that data is associated with a certain counter value. Therefore, it has to be assured that a counter value can only be bound once to data and that the counter is monotonically increasing. In sum, if all messages in a protocol are registered with a trusted counter it enables us to validate that a message in a protocol has only been sent once, the order messages have been sent by a node and it enables us to detect if messages got lost. This builds the foundation for prevent equivocation. For simplicity reasons, we outline a minimal trusted subsystem supporting one trusted signed counter similar to the implementation detailed in [VCBL09a]. However, it can be easily replaced by an extended version supporting multiple trusted counters like proposed by Levin et al. [LDLM09].

As prerequisites, we assume that every replica involved in CheapBFT has a trusted subsystem at its disposal (see Figure 9.2). Such a subsystem has been initialized with a secret key, it can be uniquely identified and possesses a local counter. None of these variables can be accessed from the outside but only changed by the trusted subsystem, and via its interface. While the counter value and the identity of a subsystem is publicly know, the key is shared amongst all

[1] In fact a leader proactively sends information for validating its correct behavior as a part of the proposal message.

trusted subsystems involved in the protocol but secret otherwise. The first of the two operations is provided by $createMC(m)$ taking the message m to be associated with a new counter value. Internally, the current value of the counter c is incremented and then, in combination with the message and the identity of the subsystem s, a message authentication code (MAC) a is generated using the secret key k known to the subsystems (see Figure 9.1, L. 2-3). As a result, a message certificate structure mc is returned that is composed of (s, c, a). The second operation, $checkMC(mc, m)$, simply takes the message certificate and the associated message as parameters. It checks if the MAC is correct and then validates that c corresponds to the next expected counter value from the sending subsystem s (L. 9, $isNext(s, c)$). While $createMC(m)$ must be entirely implemented by the trusted subsystem, this only partly applies for $checkMC(mc, m)$. In the latter case, the MAC must be verified by the trusted system whereas the order of messages can be checked by the untrusted main system.

```
 1 upon call createMC(m) do
 2     c := c + 1;
 3     a :=MAC(k, s||c||m));
 4     mc := (s, c, a)
 5     return mc;

 7 upon call checkMC(mc, m) do
 8     (s, c, a) := mc;
 9     if MAC(k, s||c||m) = a ∧ isNext(s, c) do
10         return TRUE;
11     else
12         return FALSE;
```

Figure 9.1: Implementation of a trusted signed counter

9.3 System Model and Architecture

We assume the standard system model used for BFT state-machine replication [CL99, KAD+07, KD04, YMV+03] that comprises the possibility of replicas and clients behaving arbitrarily with the additional restriction that each replica is equipped with a trusted subsystem that can only crash fail. The involved machines may operate at different speeds. They are linked by a network that may fail to deliver messages, corrupt and/or delay them, or deliver them out of order. The system guarantees safety under this asynchronous model. Liveness is ensured when the *bounded fair links* [YMV+03] assumption holds. Replicas implement a deterministic state machine [Sch90] to ensure consistency. Meaning for the same sequence of inputs every non-faulty replica must produce the same sequence of outputs. Moreover, all state machines start with the same initial state and have to be in an identical state between processing the same two requests.

Architecture-wise, we employ a classical separation of agreement and execution stage [YMV+03] (see Figure 9.2). The trusted subsystem enables us to prevent equivocation and therefore replication can be restricted to $2f + 1$ nodes at all times [CMSK07, VCBL09a, LDLM09]. In addition, we introduce the notion of *active* and *passive* replicas to BFT. While active replicas take part in agreement and execution, passive replicas primarily receive state updates during normal operation. In case of suspected or detected failures, even within the bounds of f, the system is not prepared to immediately overrule them but has to activate passive replicas to join the agreement as well as the execution stage. To ease this process, active replicas have

to support a function for providing state updates on a request basis during normal operation and supply these to passive replicas.

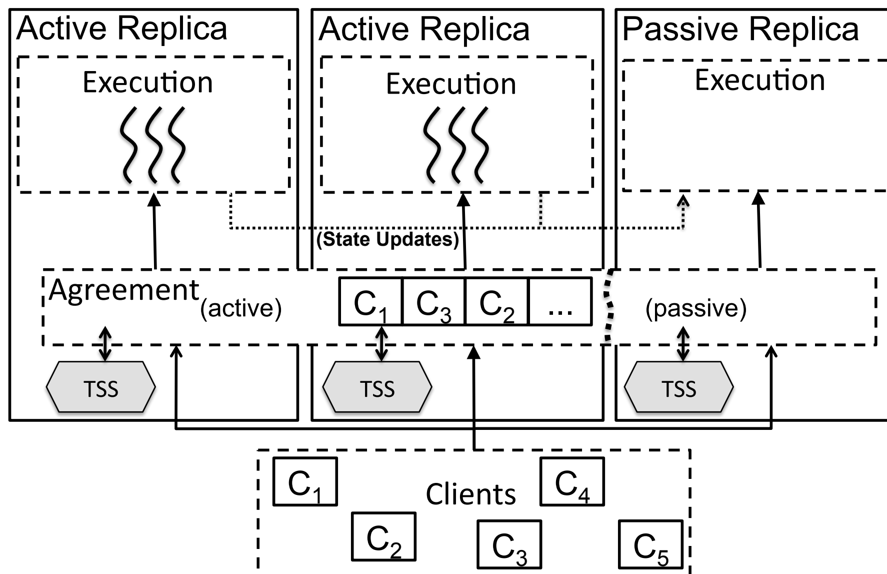

Figure 9.2: A minimal configuration of CheapBFT consisting of two active and one passive replica each equipped with a trusted subsystem (TSS).

9.4 CheapBFT Protocol Suite

We detail our protocol suite using Abstract [GKQV10], a framework that allows the implementation of BFT protocols on basis of less complex sub protocols. Furthermore, we profit from the key insight of Abstract to model reconfiguration due to errors (i. e., a faulty leader), that is, a protocol can be aborted in favor of another, often more resilient one. The CheapBFT protocol suite is composed of three sub protocols:

- **PrimaryBackup** utilizes only $f + 1$ active replicas running the main load in terms of agreement as well as execution, and f further passive replicas that basically record state changes.
- **BackupRotation** extends PrimaryBackup by rotating the role of active replicas with the aim to evenly spread the load of the agreement and execution stage amongst all replicas.
- **PlainBFT** takes over in case of a suspected or detected failure. It partly resembles MinBFT [CLVV09] and implements a variant of PBFT [CL99] that requires only $2f + 1$ due to the availability of a trusted subsystem. However, at this stage, we are not bound to use a specific protocol. The only requirement is that it requires only $2f + 1$ replicas.

9.4.1 Clients

From a client perspective PrimaryBackup is similar to standard BFT state machine replication protocols. Clients send requests to all active replicas in form of signed requests that include the command to be executed, the client identity and a sequence number. The signature enables

replicas to verify the client identity, the sequence number is utilized to ensure exactly-once semantics. The client waits for $f + 1$ matching replies.

9.4.2 Active and Passive Replicas

As earlier introduced, we distinguish two roles a replica can take: active and passive. Active replicas exchange messages with the aim to reach an agreement on the order of requests and perform their execution once this is achieved. To enable clients to identify faulty replies, it is essential that they receive $f + 1$ matching replies, ensuring that at least one replica provided a correct reply. To minimize communication as well as execution overhead, we constrain the role of active replicas to $f + 1$ nodes under fault free conditions. In case of any kind of malfunctioning, this is clearly not enough to make progress and we need up to $2f + 1$ active replicas in total. The latter being the minimal number to tolerate f Byzantine faults under the assumption of a trusted subsystem inhibiting equivocation.

To provide these additional up to f nodes at the agreement as well as the execution level, we further require passive replicas. These f replicas can be considered as a special form of observer as they receive *state updates* caused by the execution of a requests similar to clients receiving replies. A passive replica accepts an update only if the update is provided by $f + 1$ active replicas and if these updates are identical and therefore fault free. This provides them with an up-to-date service state and enables them to step in once they are needed[2].

9.4.3 When to Abort?

Accounting for the observation that $f + 1$ replicas as supported by PrimaryBackup and BackupRotation are only enough to detect or suspect errors and the requirement to activate up to f further nodes, we identified the following two conditions in which they have to be aborted in favor of a protocol that utilizes $2f + 1$ at the agreement as well as at the execution stage:

- A client does not receive matching replies for a request from all $f + 1$ replicas within a certain amount of time[3].
- A passive replica does not receive matching state updates for a request from all $f + 1$ replicas within a certain amount of time.

In either case, the failure-detecting/suspecting entity sends a PANIC message to all replicas and aborts the protocol. Note, an abort can only be triggered by authenticated clients and replicas but no other external party.

9.5 PrimaryBackup

PrimaryBackup is composed of five different messages, exchanged in four phases (see Figure 9.3). It basically resembles the phases of PBFT but omits the pre-prepare step as equivocation is not possible due to the use of a trusted subsystem (see Section 9.2 for details). Furthermore, it manages passive replicas.

[2] In fact, this is primarily needed to step in to provide additional results at the level of the execution stage.

[3] Active replicas could also detect some forms of malfunctioning at an earlier state, however, for simplicity reasons we restrict this to clients.

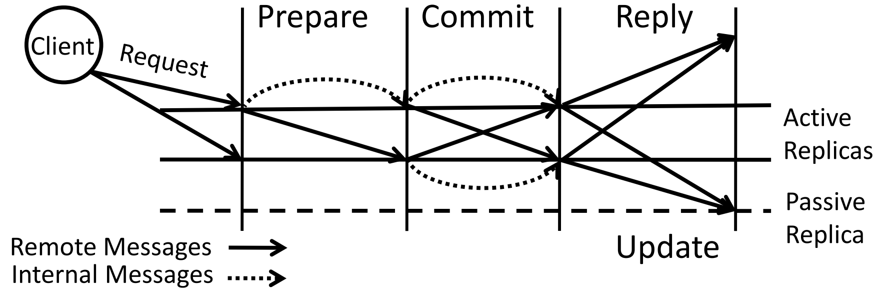

Figure 9.3: PrimaryBackup (for $f = 1$) message exchange between two active and one passive replica

1 **upon** initialization **do**
2 $active := \{p_0, p_1, \dots p_f\}$;
3 $leader := select_leader(active)$;
4 $passive := \{p_{f+1}, p_{f+2}, \dots, p_{2f}\}$;

6 **upon receiving a message** $\langle \text{REQUEST}, m\rangle$ **such that** $leader$ **do**
7 $mc_l := createMC_{ag}(m)$;
8 **send message** $\langle \text{PREPARE}, p, m, mc_l\rangle$ **to all in** $active$;

10 **upon receiving a message** $\langle \text{PREPARE}, p, m, mc_l\rangle$ **from** $leader$ **do**
11 **if** $checkMC_{ag}(mc_l, m)$ **then**
12 $mc_j := createMC_{ag}(m\|mc_l)$;
13 **send message** $\langle \text{COMMIT}, p, m, mc_l, mc_j\rangle$ **to all in** $active$;

15 **upon receiving** $\mathcal{C} := \{\ f+1$ **messages** $\langle \text{COMMIT}, p, m, mc_j, mc_l\rangle$ **such that** $checkMC_{ag}(mc_l, m) \wedge checkMC_{ag}(mc_j, m\|mc_l)$ **and all** m are equal $\}$ **do**
16 $(r, u) := execute(m)$;
17 $mc_u := createMC_{up}(r\|u\|\mathcal{C})$;
18 **send message** $\langle \text{UPDATE}, p, r, u, \mathcal{C}, mc_u\rangle$ **to all in** $passive$;
19 **send message** $\langle \text{REPLY}, r\rangle$ **to** $client$;

Figure 9.4: Protocol PrimaryBackup for active nodes

9.5.1 Agreement Stage

After initialization, each replica knows about active and passive nodes as well as a dedicated node, the leader, that is also member of the active nodes (L. 1-4, see Figure 9.4).

Similar to other PBFT-inspired agreement protocols the leader proposes the order of client provided commands. The protocol starts once this node receives a request which a client broadcasts to all active replicas in form of a $\langle \text{PREPARE}, p, m, mc_l\rangle$ message (L. 6-8). Here, p denotes the target of the message, m the command received from a client, and finally mc_l representing the message certificate that has been created using the trusted subsystem (L. 7). We use a dedicated counter ag per node for all messages related to the agreement stage.

Next, the message is eventually received by all fault free active replicas. Before processing the message, its certificate is checked (L. 11). Correct messages and the leader certificate are jointly signed by the local node and forwarded to all nodes as a $\langle \text{COMMIT}, p, m, mc_l, mc_j\rangle$ message (L 10-13). Here, we include the received message certificate by the leader and the message certificate by the processing replica. Signing the message together with the leader provided certificate helps to determine the status of pending requests in case of a protocol abort.

This is further detailed at the end of the section.

Once a COMMIT message is received by an active replica, the message certificate by the sending node as well as the leader is verified. In case a set $\mathcal{C}$ with $f+1$ correctly certified COMMIT messages with matching command are received by an active replica, the execution of the associated request is performed (L. 16). At this point, we have reached an agreement on the ordering of a request. The remaining protocol deals with the handling of the execution stage and preparations for an abort of the protocol in case of some form of suspected or detected malfunctioning.

9.5.2 Execution Stage

The execution of a command returns two results, the reply r and the service state update u caused by the execution of m. The set of commit messages are signed together with the reply and the state update. Here, we use another trusted counter up. The reason to split the signing of messages on two counters is based on the different purposes of signed messages. While in the first part of the protocol messages are ordered, in the second part message exchange is dedicated to keep passive replicas up to date.

For this reason, as a next step, a message is send to all passive replicas in the form of $\langle \text{UPDATE}, p, r, u, \mathcal{C}, mc_u \rangle$ and a reply is returned to the client (L. 18-19).

1 **upon receiving** $f+1$ $\langle \text{UPDATE}, p, r, u, \mathcal{C}, mc_u \rangle$ **such that**
2 $checkMC_{up}(r \| u \| \mathcal{C})$ **and all** r are equal **and all** u are equal **do**
3 $log(r, u)$;

Figure 9.5: Protocol PrimaryBackup passive nodes

During normal operation, for every committed and executed request, passive replicas receive the resulting service state changes and reply message as UPDATE messages (see Figure 9.5, L. 1). Once a passive replica received $f+1$ matching update messages, the update is logged to stable storage (L. 2).

As detailed earlier, a passive replica aborts PrimaryBackup by sending a PANIC message to all $2f+1$ replicas if it does not receive $f+1$ matching UPDATE messages for a certain request within a certain time span or if replies as well as updates inside these messages do not match. It learns about a request by the first state UPDATE message, starts a timer that, once it expires, triggers the abort. As every update is signed by its sender, it can be ensured that updates are applied in the order as requests are processed. To avoid out of order execution as well as out of order logging of updates, for each received message, it is verified if a previous message was received whose counter is the direct predecessor (see Section 9.2.3). This is also the reason to employ two trusted counters since the order of commands is only relevant for active replicas and passive replicas do not receive the associated messages. The same applies for updates. Their order is only important to passive replicas and associated update messages are only received by them. Accordingly, two counters are needed as otherwise all messages have to be send to all replicas.

9.5.3 Snapshots and Garbage Collection

Active replicas need to log messages in case they have to be resend (i. e., protocol abort). Passive replicas need to log received UPDATE messages to stay up-to-date to be ready in case of a protocol abort. To let logs not grow at an infinite size, snapshots of the service state are taken each n number of requests by active replicas and a $\langle \text{CHECKPOINT}, p, h, mc_{lmsg}, mc_{lup}, mc_{msg}, mc_{up} \rangle$

is send to all replicas. It includes the hash of the snapshot h, the identifier of the last included message mc_{lmsg} and update mc_{lup}. These values are concatenated and signed by both counters. Thus, the idea is to decouple snapshot creation (mc_{lmsg}, mc_{lup}) from the actual point when the snapshot message is transmitted. Once a replica received $f+1$ matching checkpoint messages, it is safe to truncate the log. For active replicas this means they keep the snapshot and erase the log. For passive replicas this includes to apply the updates using the previous snapshot to generate the new snapshot.

During this protocol we rely on $f+1$ correct active replicas. We could extend the snapshot protocol to passive replicas, however, in any case of malfunctioning, it is safe to abort the protocol in favor of a more resilient implementation.

9.5.4 Switching Protocols

In case a client or a passive replica triggers a protocol abort by sending a PANIC message, at least each correct active replica further on provides an abort history in response to client requests until the protocol has been replaced in favor of a more resilient one (see Figure 9.7, L. 1-4). Next, a client retries the (re-)execution of its aborted request by sending $\langle \text{REQUEST} | m, histories, p \rangle$. Here, m is the original command issued by the client, $histories$ the received abort histories and p the next protocol instance to use. Due to the protocol abort, the client no longer constraints the sending of requests to active but sends its requests to all replicas. Once a replica receives such an extended request, it first validates the message and than checks the abort histories. If a history is correct, the pending messages inside of it are jointly ordered and executed together with the new client request as the first messages in the new protocol. This way, the order of the abort history is maintained.

While this details the principle workflow, the following section will provide details how to determine a correct abort history and the pending requests.

9.5.5 Validating an Abort History

As every active replica provides an abort history, this guaranties that a least one correct history is available. Such a history includes all messages sent and received that were either explicitly or implicitly signed by the trusted subsystem of the history providing node. With implicitly, we mean that a message was sent and signed by the trusted subsystem including information that links a received message to the send operation (e. g., a message certificate of a PREPARE message is included in a sent COMMIT and COMMIT messages are included in UPDATE messages, see Figure 9.6 for examples).

As there might be only one correct message history available, we need to determine a correct history just by the provided data from a single node. This is possible for every replica due to the trusted subsystem as we can verify if a history is correct by checking if all messages since the last snapshot were included by checking the message certificates of all provided messages and verifying that there is no gap between the counter value included in the latest CHECKPOINT message (mc_{lmsg}, mc_{lup}) and the abort history message that is also signed by both counters (L. 1-3). In fact, such a correct history can even be provided by a faulty replica. If multiple correct abort histories are provided by different nodes, we can freely choose one of them. Typically, the shortest one might be a good pick as a faulty node can provide a correct history that has been slightly extended by sending messages after receiving a PANIC message but before sending the ABORTHISTORY message.

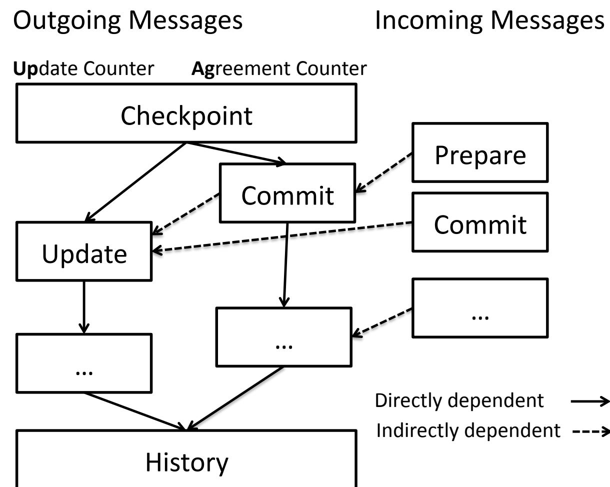

Figure 9.6: Abort history log

1 **upon receiving a message** $\langle$REQUEST, $m\rangle$ **such that** received PANIC **do**
2 $mc_{agh} := createMC_{ag}(history)$;
3 $mc_{uph} := createMC_{up}(history)$;
4 **send message** $\langle$ABORTHISTORY, $p, mc_{agh}, mc_{uph}, history\rangle$ **to** client

Figure 9.7: Respond to a PANIC message

1 **upon receiving a message** $\langle$ABORTHISTORY, $p, mc_{agh}, mc_{uph}, history\rangle$ **such that** leader **do**
2 // check history....

Figure 9.8: Checking an abort history

9.5.6 Processing an Abort History

Assuming at least one correct history message was received after a PANIC, we have the basis for the next protocol stage. Client commands reflected in this abort history can be categorized as follows:

- **Undecided:** A request was received by a subset of nodes. The leader might have send out PREPARE messages that might have even been received by a subset of the active nodes. However, the message history only provides evidence that a PREPARE message was sent out in case the history is provided by the previous leader.

- **Potentially decided:** Some nodes received a PREPARE message for a request and sent out COMMIT messages up to the point where some of the active replicas received $f + 1$ COMMIT messages and started executing the associated request. In this case, the message history provides evidence that a PREPARE message was received and a COMMIT message was sent out. However, there is no prove that it actually got decided.

- **Decided:** For a request $f+1$ matching COMMIT messages have been received, execution was performed and one or more UPDATE messages were sent out. The latter can be verified by analyzing the abort history as it includes the UPDATE message.

Undecided requests have to be reconsidered for ordering as input for the next protocol stage, but besides this, no special treatment is required. In fact, the client will eventually learn about the protocol abort and resend the pending request.

More critical is the group of decided and potentially decided messages as their processing is required to reach a consistent state across all correct replicas at the very beginning of using a new protocol. For example, passive replicas need to learn about commands that were decided but where they received no or not enough matching UPDATE messages. If a message history includes an UPDATE message that refers to a set of $f + 1$ correct COMMIT messages with a matching command this message is decided. Furthermore, there might be messages included in the abort history that are potentially decided. If an abort history provides no information about $f + 1$ COMMIT messages in form of an UPDATE message but a COMMIT message was sent out by the history providing node then some of the remaining $f + 1$ active replicas might have accepted the request. However, we do not know for sure.

Accordingly, we collect decided and potentially decided messages in the order they were proposed by the former leader and schedule them as the first messages to be ordered by the new protocol instance. This way, we ensure the order by which they were originally proposed. The abort history includes the necessary information to prevent nodes that already decided on the message and executed some of them from performing their re-execution.

Assuming that a faulty replica provides an abort history, its impact is rather limited as due to the message certificates a replica cannot undo previous actions but only influence the requests that are currently in flux. For example, a malicious replica, once it receives a PANIC message, it can send out additional messages before providing the ABORTHISTORY message. In this direction, it is possible to raise the status of a message from not known to the system to undecided and from undecided to potential decided and even from potentially decided to decided. However, this is only possible for the next logical step in the protocol and actions are preformed according to the protocol.

9.6 BackupRotation

In PrimaryBackup active and passive replicas experience a very asymmetric load. While active nodes are heavily involved in request processing at the agreement as well as the execution stage, passive nodes are not at all engaged in the first stage and only need to apply state updates in the context of the second one. Taking this into account, the target of BackupRotation is to evenly spread load over all nodes, thereby improving resource utilization and scalability. We achieve this by instantiating multiple configurations of PrimaryBackup and executing them in parallel. However, this requires coordination amongst the individual instances at the agreement and the execution stage.

For a cluster of $2f + 1$ nodes, we introduce $2f + 1$ instances of PrimaryBackup. Instances overlap in a way that no two of them collocate the leader role to the same replica. Hence, the different PrimaryBackup configurations can be identified by their leaders. To support the creation of instances, a unique identifier between $[0 - 2f]$ is assigned to each node and all nodes are virtually arranged in a ring. For each single configuration of PrimaryBackup, one node is selected as the leader. Then, the $f + 1$ nodes following the leader in the virtual ring are chosen as active nodes within the configuration and the subsequent f nodes take the role of the passive nodes (L. 1–4, see Figure 9.9). Furthermore, a global round variable is introduced which is shared amongst all configurations on a node (L. 5).

Clients submit requests to specific instances on basis of a hash $\bmod 2f + 1$ over the request that uniquely identifies PrimaryBackup configurations. Leaders accept requests for ordering based on the same partitioning scheme (L. 7). The ordering within the corresponding configuration (L. 8–14) is performed as previously detailed in Section 9.5.1. However, this results only

```
1  upon initialization do
2      l := select_leader(p_0, p_1, ..., p_2f);
3      active := {p_l, p_(l+1) mod 2f+1, ..., p_(l+f) mod 2f+1};
4      passive := {p_(l+f+1) mod 2f+1, p_(l+f+2) mod 2f+1, ..., p_(l+2f) mod 2f+1};
5      global round := 0;

7  upon receiving a message ⟨REQUEST, m⟩ such that responsible l do
8      mc_l := createMC_ag(m);
9      send message ⟨PREPARE, p, m, mc_l⟩ to all in active;

11 upon receiving a message ⟨PREPARE, p, m, mc_l⟩ from responsible l do
12     if checkMC_ag(mc_l, m) then
13         mc_j := createMC_ag(m||mc_l);
14         send message ⟨COMMIT, p, m, mc_l, mc_j⟩ to all in active;

16 upon receiving C := { f + 1 messages ⟨COMMIT, p, m, mc_j, mc_l⟩ such that checkMC_ag(mc_l, m) ∧
       checkMC_ag(mc_j, m||mc_l) and all m are equal } do
17     wait until r = l then
18         (r, u) := execute(m);
19         round := (round + 1) mod 2f + 1;
20         mc_u := createMC_up(r||u||C);
21         send message ⟨UPDATE, p, r, u, C, mc_u⟩ to all in passive;
22         send message ⟨REPLY, r⟩ to client;

24 upon receiving f + 1 ⟨UPDATE, p, r, u, C, mc_u⟩ such that
25 checkMC_up(r||u||C) and all r are equal and all u are equal do
26     log(r, u);
27     round := (round + 1) mod 2f + 1
```

Figure 9.9: Protocol BackupRotation balances load over all nodes by means of multiple configurations of PrimaryBackup

in a partial order as other configurations order request in parallel. To establish a global order across all instances, a round robin scheme is introduced in which a partially ordered request is only executed if the preceding configuration has successfully finished its request processing, If the node is an active replica within the preceding configuration, requests assigned to it have to be executed directly (L. 16–22), if the node is a passive replica, state updates have to be performed (L. 24–27). A configuration determines if it is allowed to execute a request using the round variable which is increased by any configuration after each execution (L. 19) and state update operation (L. 27). Once the round variable matches the identifier of a particular configuration (L. 17), requests belonging to this configuration can be executed.

Using this basic scheme, low or imbalanced request load would slow down the overall performance as configurations are dependent on each other. To avoid this, configurations can skip a round by proposing a *noop*, so that an other configuration can immediately take over execution of requests.

9.7 PlainBFT

In case PrimaryBackup or BackupRotation get aborted, more replicas at the agreement as well as at the execution stage are required to ensure progress despite up to f faults. Here, we propose PlainBFT that resembles MinBFT in the context of Abstract. More concrete, in PlainBFT all $2f + 1$ replicas take part in the protocol and PrimaryBackup is changed in the following way:

The set of passive replicas is set to zero and all replicas are marked as active. There is no need to signal a $Panic$ message in case a replica does not behave correct, despite the case the leader does not respond or provides incorrect messages. In the latter case the suspecting replicas send out a PANIC message. If this message is supported by at least f further replicas, a new leader has to be determined. In case of Abstract, we model this view change as a protocol abort and than switch again to PlainBFT using a higher abstract number.

9.8 Related Work

Besides performance, resource demand reduction can be considered as one main direction of optimization in the context of BFT. It started with the separation of agreement and execution enabling to limit the execution of services to $2f + 1$ nodes [YMV+03]. The same has been achieved for the agreement stage by the integration of a trusted subsystem [VCBL09a, CMSK07, RK07, CNV04].

Wood et al. [WSV+11] presented ZZ, the first practical system that constrained execution to $f + 1$ nodes and started new replicas on demand. Compared to this work it relies on a trusted hypervisor and machine management system. Furthermore, it requires $3f + 1$ nodes for agreement. SPARE [DKP+11] reduced the overhead compared to ZZ by integrating the ordering stage into the trusted subsystem and efficiently enabling proactive recovery using virtualization. However, it relies on an even larger trusted computing base and is dedicated to cluster systems. In contrast, CheapBFT is the first system that constraints agreement and execution to $f + 1$ nodes under fault free conditions and relies on a minimal trusted subsystem.

Under the crash-stop assumption, cheap Paxos can be considered as conceptually related [LM04]. Here, agreement is performed by a set of $f + 1$ main processors and a set of f auxiliary processors. In case of a failure, auxiliary processors take part in the agreement protocol and support the reconfiguration of the main processor set. CheapBFT considers arbitrary faults and balances the load using rotation at the agreement as well as the execution stage.

Up to this point, several protocols (Mencius [MJM08], Spin [VCBL09b] and Aardvark [CWA+09]) assign the role of a primary dynamically for better distribution of extra load at the agreement stage which is incurred by the leader role of Paxos and PBFT inspired protocols. Recently, executing multiple interconnected agreement protocol instances has been proposed with the aim to further disperse the load of the leader role and improve scalability [KJ10]. We extended this idea to PrimaryBackup, being even more promising in this context due to the asymmetric roles of passive and active replicas. Moreover, we employ it to distribute not only load of the agreement but also at the execution stage.

9.9 Conclusion

We presented CheapBFT, an approach for handling Byzantine failures that is tuned to a minimal resource footprint, making Byzantine fault tolerance more practical from an economic point of view. By utilizing a small trusted subsystem, CheapBFT is the first system that requires only $f + 1$ replicas at the agreement stage as well as the execution stage during normal operation. In case of suspected or detected errors, we rapidly activate up to f additional nodes at both stages and fall back to the minimal configuration of $2f + 1$ replicas.

Chapter 10

Tailored Cloud Services

Chapter Authors:
Johannes Behl, Klaus Stengel (FAU)

10.1 Introduction

Todays Cloud Computing infrastructures are more than just providers for virtual machines and network connections. Apart from the very essential virtual counterparts of common computing infrastructure, many small additional services are commonly offered by cloud providers, which are supposed to aid development and provision of scalable applications. Typical examples for such services range from all types of simple storage facilities to coordination services and load balancing. Those services are usually accessible using HTTP-based protocols and are entirely managed by the cloud provider. From a user's perspective, their main advantage is that they are easy to incorporate into cloud-based applications and don't cause any administrative overhead. However, there are also numerous downsides to those services, not only from a user's perspective.

A major problem with those services is that it's very hard to judge the security and reliability implications of using such a service. Information about the hosting environment and the service implementation are usually entirely opaque. Additionally, the programming interfaces for those services often aren't standardized across different cloud providers. This is especially a problem when applications have to be moved to another vendor because of reliability issues or excessive cost increase. The offerer of such services must maintain additional infrastructure besides the IaaS platform for administration and billing purposes.

10.2 General concepts

The basic idea to remedy the above problems is to implement those services as normal virtual machines hosted on top of the existing IaaS infrastructure. In order to do this, a service implementation has to be packaged together with the required runtime environment and a suitable operating system. Many existing service implementations are based on the Java platform, which, compared to the requirements of the service itself, requires an excessive amout of resources to run. While hard disk storage is quite cheap and not so much of a problem, even a minimal system easily consumes 200 MB of RAM which is no longer usable by the actual service. This corresponds to one third of the memory offered by a ”Micro Instance” at Amazon EC2 [LLC11]. To make matters worse, it is often desirable to start multiple instances of such

services at different sites to ensure availability in case of single failures, multiplying the amount of wasted resources.

The obvious solution is now to identify the components really required for each service and load only those. In order to make creation of the virtual machine images easier and add more flexibility, this tailoring process is delayed until the service is finally instantiated in the cloud environment. That way we can simply distribute the same image to several providers at once and also have a chance to adjust the service for the specific needs of different applications.

In order to demonstrate the advantages of our approch, we planned to develop a caching service on top of our platform. It provides a key/value store similar to the popular *memcached* product [web], that stores all data in RAM and is commonly used to accelerate the delivery of web-pages on the Internet.

10.3 Operating System design

Another major issue with packaging a simple service to run on an IaaS cloud is the choice of an operating system. The very popular GNU/Linux distributions are primarily intended to run on real hardware instead of virtualized environments and serve as a general purpose operating system. This leaves much room for improvement towards a system that is specifically designed to run inside a virtual machine, especially regarding the required abstraction layers and programming interfaces offered to the application programmer: While the virtual machine is expected to already handle most of the low-level hardware details, the kernel can provide interfaces that are more geared towards distributed programming using web protocols than simple TCP sockets.

10.3.1 Virtual hardware support

Fortunately, there is only a very limited number of virtualization environments available for the Intel x86 architecture, that is also suitable to be used in an IaaS cloud. Thus it is quite easy to attain support for all the virtualized hardware found at cloud hosters. The following list of virtualization products ist often encountered on cloud platforms:

- Xen (OpenStack, Amazon EC2)
- VMware ESX (VMware vSphere, corporate private/hybrid clouds)
- Linux KVM (OpenStack, OpenNebula, Eucalyptus, ...)
- Microsoft Hyper-V (Microsoft Azure cloud)

Our service platform is designed to offer non-interactive services which are only reachable over the Internet, so there is no need for supporting complex graphics adapters, nor human-interface devices like e.g. keyboards and computer mice. Thus we can concentrate on support of network and (optionally) any persistent storage hardware emulated by those virtualization products. As it turns out, we can re-use the Xen interface for both the Xen Hypervisor as well as the devices from Hyper-V, as Microsoft claims to be compatible:

- XenBus: Xen, Microsoft Hyper-V
- VirtIO: Linux KVM

- PVSCSI/VMXNET: VMware ESX

With this rather small set of different interfaces it is possible to support all of the virtualization environments usually encountered in today's IaaS clouds. Note that these are already specialized interfaces that use ring buffers in shared memory segments for the data transfer instead of slow emulations of real hardware.

10.3.2 Tailoring process

As mentioned in 10.2, it can have several advantages to delay the tailoring process to the latest possible moment, which is just before the service is actually needed. This also means that the tailoring will happen in the same virtual machine that will host the final service. As it is very difficult to host a fully featured compiler to build a tailored operating system and service, we first boot a commodity Linux operating system just for the tailoring process. Thus the general procedure to instantiate a service involves the following steps:

1. Start the virtual machine image
2. Inspect the VM environment (which Hypervisor, available resources)
3. Query the application for any specific requirements for this instance
4. Select required components to run service and assemble
5. Replace running operating system with the actual service instance

10.4 Choice of Programming Languages

Another major goal of our platform for tailored software components is the improvement of security and performance. As outlined in 10.2, we can make many assumptions about the execution environment and thus reduce the general complexity of the system. This should already by itself result in less possible faults in the software and faster execution speed. Additionally, the choice of programming language also has severe impact on those goals, as shown by several studies, e.g. by comparing Ada and C [Zei95]. A big factor seems to be how much of the intentions can be expressed in the type system of a language and what properties it can expose to static checking by the compiler.

10.4.1 Formal verification

According to the Curry-Howard isomorphism [SU98], the types associated with objects in any programming language can actually be interpreted as propositions for a proof and the program logic is a formal proof for those propositions and vice versa. This correspondence can be exploited to establish proofs of certain properties of a program. The desired behaviour of a program has to be expressed as a set of propositions using predicate logic. Using constructive type theory, one has to show that the propositions always hold.

As this can be a rather time-consuming and error-prone process for a human, such a task is commonly supported by special proof assistent software. Equipped with a large rule set and an interactive user-interface, these programs can significantly reduce the time required to construct a proof and also verify that the proofs do not contain any mistakes.

While the *seL4* project has shown that the proof assistants are powerful enough to perform a complete verification of a microkernel operating system, doing so is quite time-consuming. Moreover, they still need to make many assumptions regarding complex pieces of hard- and software, like CPU caches and the compiler [KEH+09]. Thus a more practical approach could consist of taking a programming language with a safe type system for implementing simpler parts of the platform, and just resort to the strong safety guarantees of formal verification in parts of the system that are especially security critical or that are hard to get right. Therefore we consider it worthwhile to look at several currently available system programming languages and their interaction with formal verification methods.

10.4.2 Language candidates

The following subsections contain a short description of languages that are currently evaluated for the final implementation of our system. Besides a general overview, we focus on how much runtime support is required for each language and how suitable they are for verfication and correctness proofs.

C/C++

C and its successor *C++* are probably the most popular languages used for system and application development and are closely tied to the invention of the *UNIX* operating system in the early 1970s. The C language supports an imperative programming model that is relatively close to the bare hardware, but features a uniform, portable syntax (compared to writing assembly code) and allows specification of custom data types as well as structuring code using expressions, code blocks and functions. C++ extends the C language mostly with additional support for object-oriented programming techniques and a powerful template metaprogramming facility. Although many valid C programs are also valid C++ programs, both languages are currently developed independently from each other. Thus there are differences regarding the type system and newer language features, which means that C++ generally isn't a superset of C.

Unfortunately, static verification of programs from the family of C languages is very hard. The main reason for this is that the behaviour at runtime of certain expressions is intentionally left implementation-specific, or sometimes even entirely undefined. Although there are many tools available that specifically try to detect such constructs, it is impossible to do so reliably: Very often the validity of a expression depends on the value of some variable to be within a certain range, which often can't be deduced until the program is actually executed. Another issue is that the language relies on calculations with memory addresses to implement references to variables. Those properties make it easier to write a compiler and allow certain optimization tricks to improve runtime performance, but signficantly hinder static analysis.

There exist multiple approaches to make the language family more accessible to formal verification methods, either by modifications to the language itself, or by providing external frameworks. A good example for the former approach is the *Cyclone* dialect [JMG+02], which disallows most error-prone features of C, replacing them with alternatives that either allow verification at compile-time or at least mandate insertion of corresponding runtime-checks. Those changes are mostly aimed at memory integrity, i.e. they only ensure that dereferencing a pointer always leads to a valid object. There is no support to enforce any constraints regarding the behaviour on the application level. The other possible approach is taken by the *Frama-C* project, which keeps the semantics of the C language intact, but specifies an additional language called *ACSL* [BCF+10] to allow specification of certain properties the program should exhibit.

There are plenty compilers available to choose from. Most support a special "freestanding" mode, in which the generated machine code won't rely on existance of any kind of execution environment. Thus the language is very well suited to build operating systems.

10.4.3 Java

The Java language is also very popular, especially for server application development. It was created in the early 1990s and it's syntax is very similar to C++, but lacks some advanced features like multiple inheritance and operator overloading. Raw memory pointers were also removed from the language and replaced with a reference mechanism which no longer allows any pointer arithmetic. The outcome of each valid expression is defined in the specification; There are no undefined or implementation-specific results possible, which greatly improves safety and portabillity. The language ships with a huge standard library, containing all sorts of data structures, predefined classes to handle many common data formats and two different toolkits for building graphical user interfaces.

The language was designed to be translated into a byte-code representation, which can be executed on the Java Virtual Machine (JVM). Thereby, programs written in Java can be executed on any machine for which an implementation of the JVM exists. The JVM is responsible for interpreting or translating the byte-code into native machine code at runtime. It also has to manage the lifetime of the objects in RAM, as there is no mechanism in the Java language to explicitly delete an object. Instead the JVM has to scan all objects periodically and determine whether they still might be accessed to reclaim memory (Garbage Collection). Unfortunately, this also means that a substantial amount support code is necessary to execute Java programs, which can't be expressed in the safe Java language itself. Programming mistakes in those layers and the nature of the Just-in-Time compilation environment can provide additional attack vectors, as shown by very recent research [RI11].

Regarding verification of Java programs, there seems to be only the *ESC/Java2* [Kin] project, which allows the programmer to add special comments to the Java code that are interpreted by a separate checker tool. It enables static verification of pre- and postconditions of methods and seems to integrate with more powerful proof checking systems like *Isabelle*. Unfortunately, development seems to have stopped a few years ago and the latest release only supports outdated versions of the JVM, severely limiting its usefulness.

Due to its widespread use, there are many Java development kits available from different vendors, mostly for free and often with an open-source license.

ATS

ATS [Xi04] is a research language that tries to support several programming styles (e.g. functional and imperative) at once with an extended type system. Using a variant of so called *dependent types*, the language can express many constraints and rules regarding the use of functions just as part of the types. One feature that makes the type system special is that function definitions include rules for modifying the types of the parameters passed to that function. As an example, this can ensure that a file handle is never closed twice by changing the type to an incompatible one at each invocation of the close function. It is an error detected at compile-time if there is a codepath that might result in a type mismatch.

The ATS compiler doesn't generate machine code, but translates ATS programs into portable C code and leaves the translation into machine code to a C compiler for the target platform. As the language doesn't require any additional runtime management, it is possible to construct

types that have a one-to-one correspondence to types in the C language, but with the extra checking the ATS type system allows. This makes is possible to interact seamlessly with low-level C and assembly routines while still retaining much of the safety ATS provides.

As ATS is still a research prototype, it lacks easily understood documentation for more advanced language features and the syntax is subject to change. Additionally, the linear constraint checker currently uses machine word arithmetic, which can overflow and make type verification unreliable. Although the language can be used without any garbage collector, there are no clear rules which language constructs even cause memory allocations to be performed. This is important if we want to directly write programs on top of a virtualization layer, where dynamic memory managment isn't generally available.

There exists only the reference implementation of ATS, written largely in ATS itself. It is available under the *GPL* open-source license.

Haskell

We picked Haskell for this comparison as a representative for a more general set of functional languages like *OCaml*, *Clean* or *LISP*. It has the advantage that it encapsulates all computations with side effects into so called *Monads*, making all functions pure. This property makes reasoning about code much easier. Haskell is also a quite popular language with multiple open-source compilers available from different groups.

Powerful theorem provers like *Isabelle* [UT] or *Coq* [INR] can extract Haskell code directly from proof terms. This makes Haskell an excellent choice for writing verified programs. A disadvantage is that the language is still quite abstract and requires more effort than other languages to be translated into efficient machine code. There is no explicit memory managment possible, so the language generally requires a garbage collection framework like Java does. However, there are limited approaches that can statically deduce the memory requirements by region inference [TBEH04]. This allows bootstrapping the language in environments where no garbage collector is available.

Ada

Ada is almost as old as the C programming language, with the first official standard appearing in 1983 and development still in progress. Newer versions support object oriented programming in a similar style to Java. Memory managment must be done manually in most implementations because Ada doesn't guarantee the availability of a garbage collector. Despite having many features, the language stays easily readable and comes with extensions for supporting tasks, interoperability with other languages and distributed execution of program modules. There is also a standardized subset of Ada95 for embedded and especially safety-critcal systems called *SPARK* [CA08], which was designed for easy automated verification using special annotations in the comments of the program text. Unfortunately, this subset is severly limited and excludes variable references and explicit dynamic memory allocations, making it unsuitable for certain algorithms and systems processing datasets of varying sizes.

The Ada programming language is mostly found in the military and aviation sector, which also means there is a lot of industry support available. Apart from many commercial vendors, the popular *GNU Compiler Collection (GCC)* also contains an up-to-date implementation called *GNAT* and is available as free software.

10.5 Summary

As shown in the previous sections, there are many opportunities to improve security and the ressource footprint of simple cloud services. While supporting a wide range of cloud platforms is quite easy due to the limited number of virtualization layers currently in common use, ensuring security properties of such an platform is very hard. We looked at a set of commonly available programming languages to find out how well they are suited for formal verification and get a general feeling for issues that may arise when used to interface with a bare virtualized environment instead of being hosted on an operating system.

Chapter 11

Trusted Platform Agent

Chapter Authors:
Emanuele Cesena, Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia, Davide Vernizzi (POL)

In this chapter we present the architecture of the Trusted Platform Agent (TPA). Further details, including examples of use of the TPA demonstrating its features can be found in [CCS+11].

The TPA is designed to minimize the effort of writing applications that use TC technology and employ the TPM. Writing applications for TC requires a TCG Software Stack (TSS), but there are still a number of tedious and repetitive operations to be carried out. One way to reduce these is by linking an application with the TPA library which can avoid the complexity of the TSS interface and simplifies specific tasks where TC primitives need to be integrated with other commonly needed functions (e.g. cryptographic or network-related functions).

11.1 Introduction

The Trusted Computing technology has succeeded in defining industry specifications, where the most visible result is a cost-effective and tamper-resistant chip called Trusted Platform Module [Tru07b].

Despite the fact that the TPM has already been deployed on over 200 million computers, the complexity of building a complete trusted platform is a major obstacle in the diffusion of applications that use TC technology to enhance software protection. Furthermore, even though the TCG specifies a complete interface for accessing the TPM from an application layer through a TCG Software Stack [Tru07a], application developers still have to accomplish a number of tedious and repetitive tasks, mainly due to the high complexity of the TSS interface.

The TPA overcomes this issue by providing a simple and complete solution for writing trusted applications. The most evident outcome is the *aggregation* of TSS functions into a higher level API, which hides many technical details imposed by the TSS from the programmer. Nevertheless, the peculiar characteristic of the TPA is to provide secure *integration* of TC functionality with other commonly needed functions, for instance, cryptographic or network-related, thus creating a bridge between research – where TC is developed – and real-world applications, that gain state-of-the-art security features. Finally, the TPA provides *consolidation* and reference implementation of procedures needed to fill the gap between TSS elementary tasks and well-established application activities.

11.2 Software Development with TC

In this section we outline the basics of TC which are the most relevant when developing software. For a broader discussion of this technology we refer to [Mit08], while readers who are more interested in the development may find [CYC+08a] of interest.

As previously mentioned, the core of TC technology is the TPM, a cost-effective chip bound to a platform whose functionality includes: random number generation; RSA key pairs generation and their usage for encryption or digital signatures; protected storage by binding (encrypting) data to the platform. An important feature of the TPM is to guarantee that the private part of RSA keys stored in the chip are never exposed.

Applications can access the TPM through the TSS, whose architecture consists of three layers: the TCG Service Provider (TSP) which provides the `Tspi` programming interface to applications, the TCG Core Services (TCS) that runs as a system service and multiplexes requests made by the TSP and finally the TCG Device Driver Library (TDDL) that directly accesses the TPM.

As a first idea, TC does not try to build a computing platform that never fails, but focuses on the apparently simpler question of how to identify application binaries, together with libraries, relevant files and required system components. Identification is a crucial task in providing an adequate software protection and deciding whether to trust a platform or an application. Refer to [PMP10] for an extensive discussion on the relationship between identification and trust, which also highlights research challenges.

In TC, code identification is achieved by *measuring* (computing a cryptographic hash) each software binary together with its input, libraries and configuration files. The best time to measure a software is before its execution begins, therefore a component C_n must be measured by the component C_{n-1} which is in charge of executing it. This logical progression continues recursively, leading to a Chain of Trust, where each component *measures-and-executes* its successor. The starting point of this chain is a hardware component called Root of Trust for Measurements (RTM). In commodity computers, the first portion of the BIOS, called Core Root of Trust for Measurements (CRTM), initiates the Chain of Trust by measuring the remainder of the BIOS code. The BIOS then measures and executes the boot loader which in turn will do the same with the OS kernel, and so forth.

TPM "accumulates" measurements into shielded memory locations, called PCR and guarantees that the value V of a PCR can only be updated through the *PCR extension* operation, which avoids V from reverting to a previous value (technically, $V =$ SHA-1$(V||M)$ where M is the measurement to extend with).

Measurements are used locally for sealing and can be reliably reported to a remote party through the Remote Attestation. *Sealing* enriches TPM binding by linking data to a set of PCR values so that sealed data can only be unsealed (decrypted) if the current values of the PCRs match the ones specified at the time of sealing. Remote Attestation is the process of reporting measurements (signed by the TPM) to a remote verifier, to convince her that the platform is actually running the measured code.

11.3 Trusted Platform Agent

The TPA is a C library designed to minimize the effort required to write applications that use TC technology.

Features	TPA	TrouSerS	jTSS	μTSS	JSR 312
Language	C	C	Java	C++	Java
Learning curve	Low	High	Medium	Low	Low
Abstraction of TC features	High	Low (TSS)	Low (TSS)	Medium	Medium
Description	Open source library designed to minimize the effort of writing applications that use TC technology and employ the TPM. Also provides integration with commonly needed functions.	Open source TSS implementation. It is considered the TSS reference implementation.	Platform-independent TSS wholly written in Java. It provides an API compliant with the TSS, and can also be used as a wrapper on TrouSerS.	Simplified TSS with a lightweight API. It allows the use of only a subset of the functionality as it is required, e.g., for embedded systems or in the context of security kernels.	Java API for TC, alternative to the TSS. It reduces the complexity of TC-aware applications and makes TC accessible to a large group of developers.
Web page	security.polito.it/tc/tpa	trousers.sourceforge.net	trustedjava.sourceforge.net	N/A	jsr321.dev.java.net

Table 11.1: Libraries related to TC. The table does not list TPM/J and TPM4JAVA since they are no longer developed.

In Table 11.1 we present a comparison between TPA and other projects related to TC. TrouSerS and jTSS are implementations of the TCG specification, while μTSS [SZ10] provides a lightweight TSS implementation specifically designed for embedded systems. The most similar library to the TPA is JSR 312 [TWNH09], that provides an abstraction useful for writing applications (and furthermore is undergoing standardization). However, as opposed to JSR 312, the TPA tries to provide a complete solutions integrating TC functionality with commonly needed functions, specifically cryptographic and network-related.

The TPA was developed following principles of simplicity, modularity and security, and features the following architecture:

- The lower layer wraps all the libraries needed by the TPA (e.g. TSS, cryptographic and network) to provide a unified interface to the upper layers.
- The core is organized in several modules, one for each functionality exposed by the TPA. Each module defines abstract objects that map onto TSS-specific objects and composes lower layer functions into aggregated operations.
- The high level API allows developers to access all of the TPA features. While the TPA simplifies the TSS interface, it still permits the same level of flexibility.
- The μAPI is a light interface built on top of the high level API, which implements the most common patterns used by applications, at the expense of some limitations (e.g. it is single-threaded). In order to make the most of both APIs, the TPA allows to switch between the two at any time.

Note that the TPA functions have been defined by factorizing and generalizing the TC-related code fragments most frequently found in Trusted Applications. As such, TPA supports code reuse by avoiding tedious and error-prone repetitions of the same code fragments.

The TPA is available for Linux and Windows and has been tested with TrouSerS (see also [tro]) as TSS, with Infineon TPM and TPM emulator [SS08]. The TPA also requires OpenSSL for cryptographic operations, SQLite as database library to implement its local storage and cURL for networking interaction.

11.4 Features of the TPA

This section describes the TPA's features, grouped by logical functionality. In [CCS+11] we present two examples of use of the TPA, via the μAPI, showing advanced features of the TPA and demonstrating its novelty in providing an application-oriented API and integrating TC functions with other libraries.

11.4.1 Foundation of Protection

TPM basic commands. The TPM is managed by an entity called *owner*, defined through the *take ownership* operation, who controls the secret required to perform critical tasks.

Applications only require taking ownership or verifying if ownership has already been taken, usually during the installation or setup phase. The TPA assists applications with these two functions.

PCR, SML and system state. The TPM stores integrity measurements into PCRs. To help with the interpretation of PCR values, additional logs called *Stored Measurement Logs (SML)* may be kept. Together with the PCR values, they represent the *system state*.

The TPA inherits the concept of *PCR set* from TSS and implements an application-specific SML, needed for sealing and Remote Attestation.

TC is based on the concept that a component must be measured before being used. It is crucial that measure and use are executed as an atomic operation, to prevent an attacker from substituting the component between the two. The TPA implements this concept on common functions (e.g. file open, library load, SSL connect/accept) providing wrappers where relevant data for the function is measured before usage.

In future work we plan to employ database protection mechanisms to protect privacy-sensitive measurements.

11.4.2 Local Protection

Key management and SMK. TPM keys are *wrapped* (encrypted) by a parent key, which leads to a key hierarchy with a root called Storage Root Key (SRK). Each key can be protected with a secret, which must be provided to authorize operations. While normally TPM keys can only be used by the TPM that generated them, sometimes they can be defined as *migratable* to be used with other TPMs or software.

The TPA exposes a set of functions for managing the life cycle of TPM keys. Applications can use the TPA to create, load and delete keys. Similarly to TSS, keys are hierarchically organized and identified by labels. Moreover, the TPA allows the limitation of key visibility to application keys only, introducing an application-specific database, while TSS only supports system/user-level storage. The TPA also guarantees isolation from other applications by transparently wrapping any stored object with a unique TPM key called Storage Master Key (SMK), which is protected by a secret only known to the application.

Binding and sealing. As binding and sealing are performed with a 2048-bit RSA key, the TSS imposes constraints on the size of the data to be protected. The TPA abstracts binding and sealing of data by removing this limitation. Therefore, the TPA (1) generates a symmetric key,

(2) encrypts the data with this key, (3) binds (or seals) the symmetric key, and (4) eventually encodes the resulting blob.

Backup and restore. Availability of bound data, as well as of any wrapped key, is a serious concern in case of TPM failure, but this issue is not addressed by the TSS. Currently, the TPA provides a simple way to backup and restore bound (or sealed) data by returning the symmetric key used to actually encrypt the data together with the encrypted blob itself, to the application. By default, the TPA protects the symmetric key with a passphrase. However the application may recover the symmetric key in clear and protect it with any alternative method.

In future work, we plan to extend the backup functionality to TPM keys, exploiting the migration feature provided by the TPM.

11.4.3 Vouching for Protection

EK and AIK. Each TPM is provided with a RSA key pair, the Endorsement Key (EK), which is generated and certified at manufacturing time. As the EK uniquely identifies a TPM, and thus the platform, its use (or even disclosure of the public part) poses serious privacy issues. An Attestation Identity Key (AIK) is an RSA key pair created for use in Remote Attestation. It is an alias of the EK, generated by the TPM and tied to its identity. An AIK can be proven to be created by a genuine TPM, without exposing any part of the EK. The TPA manages the complete lifecycle of an AIK before it can be used for Remote Attestation, i.e. generation and certification according to TCG specifications. AIK certification is a complex procedure where a TPM proves that it is genuine by showing its EK certificate to an external authority (as external authority we choose `www.privacyca.com`). After successful certification, the AIK can be used for Remote Attestation because its certificate proves that the TPM is genuine, without exposing any part of the EK.

Remote attestation. To perform a Remote Attestation, the TPM signs a set of PCR values using an AIK and the resulting signature is called *quote*. The TSS only offers the support for computing a quote.

The TPA offers a complete support for Remote Attestation by providing functions for: (1) computing a quote, i.e. a signature over a set of PCR values with an AIK; (2) serializing/deserializing the quote, together with the corresponding measurement logs and AIK certificate chain; (3) verifying the remote attestation data.

Verification can be cryptographic or semantic. The former is needed to guarantee that measurements are genuine (e.g. check the correctness of the quote), while the latter aims to decide whether measurements are acceptable or not against reference values.

The TPA implements cryptographic verification and returns a list of validated measurements to the application which is in charge of the semantic verification. In some specific contexts, the TPA also provides functions useful for semantic verification (see [CCS+11] for an example related to SSL channels).

11.5 Concluding Remarks

Although TC is a promising technology, it has several limitations which prevent immediate usage in many real world applications. The most evident is the slowness of the TPM (e.g., it takes up to 2 seconds for a single unsealing or quote) and, therefore, its direct use is not suitable

when a high workload is expected. Additionally, both sealing and Remote Attestation rely on the concept of system state. In order to capture this state, it is essential at boot time to create a complete Chain of Trust from the RTM up to the application. Unfortunately, coping with all aspects of chains of trust is still a research problem with many facets [PMP10]. However, it is currently possible to create a basic, yet complete, Chain of Trust under the condition of using (1) a trusted boot loader [Kau07] and (2) a TC-enabled OS kernel, for instance the Linux Kernel with the Integrity Measurement Architecture (IMA) [SZJvD04], capable of measuring applications binary, libraries and all files accessed by any process.

Currently, the lack of a widely available mechanism for a complete Chain of Trust is the major security constraint of the TPA (but also of all TC-related libraries). The TPA is still under development and in future work, in addition to cover the TSS functions not yet supported (e.g. migration), we plan to enhance the local storage to support privacy-sensitive measurement logs, extend the backup-restore functionality to keys, as well as increase the number of *measure-and-use* features to other common functions (e.g., load configuration files or cryptographic material).

Chapter 12

Trusted Infrastructures

Chapter Authors:
Michael Gröne, Norbert Schirmer (SRX)

In this chapter we present the trust model, functionality and architecture of Trusted Infrastructures (also referred to as TrustedInfrastructure (Sirrix)) and its cloud specific Trusted Virtual Domain (TVD) and information flow control mechanisms. Today's information systems still lack efficient protection against both outsider and insider threats. Targeted malware attacks and data leakages are the most visible examples of these increasing threats. Today, IT infrastructures are shared, distributed, and heterogeneous. They extend into Cloud Computing. Thus, a comprehensive approach to endpoint and information flow security by using TC technology is needed.

12.1 Introduction

Organizations often struggle with the challenge that employees have to use IT systems for different tasks with different security requirements. They have to deal with confidential data while they are also working on data and documents that are supposed to be shared with others. Employees perform different tasks with different entitlements (e.g. roles), for example using Intranet services, editing unclassified documents, as well as editing classified documents, such as patents or accessing the Internet. Each of these kinds of tasks has different security requirements. In security-critical environments such as military and government, classified documents are isolated by using physically separated computing environments. However, in typical enterprise environments users perform these tasks using one computing environment providing a questionable isolation between them. Instead we can observe the opposite trend, i.e., more and more infrastructures are shared for several tasks. Sometimes infrastructures are even shared for multiple organizational units or even complete enterprises. For example, Cloud Computing providers offer Infrastructure as a Service (IaaS) for different customers on the same hardware platforms. While this provides more flexibility for the provider and is cost-efficient for the user of a Cloud Computing based service, it increases the security problems organizations have to deal with in order to isolate data of different workflows and to fulfil confidentiality demands if sharing data is required. In addition, employees may use mobile computing platforms which are not always under control of the organization's domain. Sending documents to private computers in order to work from home, and later bringing them back into the organizational domain is also not unusual. If private computers are not protected sufficiently, confidential data may leak or malware may enter the organization due to such a data transfer. [CLM+10] In this context, security concerns become even more urgent when mobile storage devices, such as portable hard drives and USB memory sticks, and cloud storage, like Amazon S3, are used, which offer

additional flexibility for the accessibility of data across multiple working locations and devices [CLM+09]. This needs to take into account diverse security risks with regard to the data stored on the storage devices. For example, mobile storage devices can easily be lost or stolen, cloud administrators may gain access to data, and consequently the confidentiality of data becomes an issue. Once left unattended by the user, mobile devices can be manipulated with the goal to break the integrity of the data or to disseminate corrupted data or malicious code once the device is reconnected to the enterprise platform. In case of cloud storage the situation is even worse since the user is not able to watch over his data. Many security solutions for mobile storage devices adopted in practice rely on a mixture of different techniques. In fact, the choice of appropriate mechanisms is guided by trade-off between their costs and offered benefits [BCG+08], [PKM08]. Recent surveys show that existing security policies vary across organizations from none to very restrictive ones disallowing those devices completely [Fab07], [NE08].

It is possible to deal with all these security concerns in a still manageable way by using Virtual Computing Environments which implement a trust anchor in hardware, such as TVD.

12.2 Infrastructure Overview

In a virtualized environment, different applications and services together with their underlying operating systems are executed by different VM instances that share the same physical infrastructure, hardware platform, and network. Each VM instance runs in a logically isolated execution environment (which we call compartment), controlled by the underlying Virtual Machine Monitor (VMM). In such an environment, the User's work space is executed in a VM instance. In the Trusted Infrastructure a central management component, called TrustedObjects Manager (TOM) (cf. D.2.4.1, Chapter 9), manages a set of appliances, e.g. TrustedServers (cf. D.2.4.1, Chapter 7) (cf. Figure 12.1).

All components, appliances as well as the TOM, are equipped with a TPM. When a component is started the TPM is employed for secure booting to ensure the integrity of the hardware and software (in particular of the security kernel) of the component. Moreover, the local hard drives of a component are encrypted by a key that is stored within the TPM. Via sealing, the component can only decrypt the local hard drives if the TPM has cross-checked the integrity of the component. Hence only an integer security kernel can access the decrypted data. All administrative tasks are controlled by the TOM, so on an appliance there is no need for manual administration. Therefore no almighty administrator/root account is needed. This mitigates the risks of malicious insiders such as administrators within the premise of the organization. The TOM is in charge to deploy configuration data to the appliances. Via the Trusted Management Channel (TMC) (cf. D.2.4.1, Chapter 9) the TOM ensures the integrity of an appliance using remote attestation before transmitting any data. The communication between TOM and the appliances is secured by a trusted channel (also referred to as TrustedChannel (Sirrix)). Security services within the TURAYA ™SecurityKernel then handle the configuration and ensure that security policies are properly enforced by the component.

12.3 Trust Model

In this section we give a brief overview of the trust model of the Trusted Infrastructure. The trust model is based on the idea of TC: implement the trust anchor in hardware. This Trusted Platform Module is able to create secrets and to store and use them. Moreover, the TPM can be

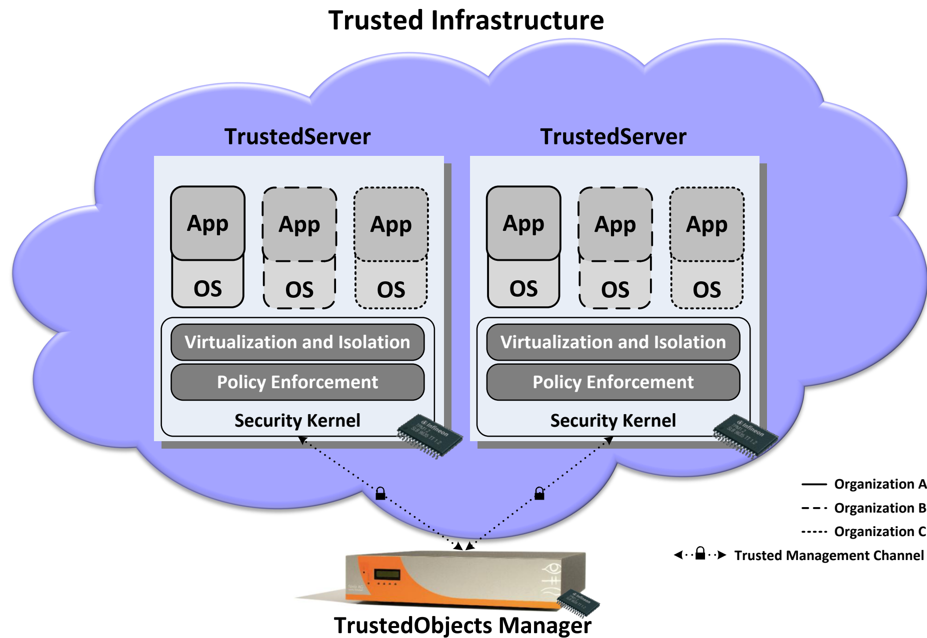

Figure 12.1: Schematic Trusted Infrastructure - TrustedServers managed by the TrustedObjects Manager.

used to report the system status, for example to verify if certain safety protocols are in place or if the compartment is in a predefined state.

Assumptions This description of all assumptions characterizes the security aspects of the environment in which the components of the Trusted Infrastructure will be used or are intended to be used.

- **Trusted Organization Security Admin:** The organization's security administrator (in TClouds terminology this is a User) of the management component is non-malicious.
- **Untrusted Cloud Admin:** The Cloud Admin of the cloud provider may be malicious and has only remote access.
- **Physical Access Control (Cloud Admin):** The Cloud Admin of the cloud provider does not have physical access to infrastructure components, e.g. to the Random-access memory (RAM) of the trusted servers. She has remote access only.
- **Correct Hardware:** The underlying hardware (e.g., Central Processing Unit (CPU), devices, TPM, etc.) does not contain backdoors, is non-malicious, and behaves as specified. See also 'Physical Access Control' assumption.
- **Correct Software:** The Trusted Computing Base (TCB) (Hardware, TPM, TURAYA ™SecurityKernel and TOM) does not contain backdoors, is non-malicious, and behaves as specified.

12.4 Central Management

To get along with the trust model, a technical design has to be done that ensures data integrity, confidentiality and availability. In this section we give a brief overview of the central management concept and its features. We describe the TOM, which is also the *TVD Management Component* (cf. section 12.5). More details on the TOM component are described in Deliverable D2.4.1 (cf. D2.4.1, Chapter 9).

The TOM is the heart component of a *Trusted Infrastructure* in such way that it controls the configuration of all appliances used in those infrastructures. Appliances typically are a piece of hardware and software, conforming to a specification depending on the security solution: e.g. for the *TrustedServer* component the appliances are the individual servers on premise or cloud servers.

The TOM provides the graphical user interface (GUI) to define TVDs and corresponding intra-TVD and inter-TVD information flow policies. Moreover, the TOM manages the physical infrastructure including networks, services and appliances (physical platforms). Since appliances remotely enforce a subset of the overall security policy, a permanent trusted channel [AGrS+08], [GPS06] between the TVD Management Component and its appliances is used for client authentication, to check their software configuration using attestation, and to upload policy changes and software updates. Finally, the TOM creates an independent TVD-specific Root Certificate Authority (CA) for each defined TVD.

The *TVD Manager* security service receives the TVD-based security policies on its appliance from the TOM through the *TrustedChannel Daemon* and also security policy updates if deployed by the TOM. The component contains a TVD policy engine which is able to enforce the TVD-based security policies to each compartment that has the correspondent registered and certified TVD membership.

12.4.1 Functionality

The main functionalities of the TOM are:

- **Central Infrastructure Management:** The TOM allows organizations to centrally manage their IT infrastructure.
- **Central Security Management:** The TOM allows organizations to centrally manage the security policy enforced by the IT infrastructure.
- **Public Key Infrastructure (PKI):** The TOM provides a PKI for management and provisioning of keys.
- **Security Module:** The TOM uses TPM (or Hardware Security Module (HSM)), e.g. to check integrity.
- **Crash Recovery:** If a primary TOM fails, the user is always able to import backup data into a backup TOM that was not known when the backup was created.
- **Cold Standby:** The TOM supports Cold Standby. When the primary TOM fails, a backup TOM take over the role of the primary TOM by importing a backup of the primary TOM.
- **Warm Standby:** The TOM supports Warm Standby. When the primary TOM fails, a backup TOM takes over the role of the primary TOM within a few minutes and with only a minimal loss of configuration information.

12.4.2 Management

The management of a *Trusted Infrastructure* is done through a Graphical User Interface (GUI) that allows administrators to manage logical objects and the *Trusted Channel* that connects and configures the appliances.

GUI-Based Management of Logical Objects

- User Management (authentication, authorization, add and delete users)
- Organization Management
- Site Management
- Appliance Management
- Network Management
- Server Management
- Cluster Management
- Cloud Management (single cloud)
- TVD Management

Trusted Channel / Trusted Management Channel

- Accept Connection
- Configure Appliances
- Update Appliances

More details on the Trusted Management Channel (TMC) component are described in Deliverable D2.4.1 (cf. D2.4.1, Chapter 9).

Key Management

Trusted Appliances (e.g. TrustedServer) all have a TPM, their keys are only stored in a space which is encrypted and sealed by a TPM (or equivalent hardware anchor, such as a HSM). The TOM has a Root-CA or may use existent Root-CA of an PKI an organization has. TOM generates two (root) keys per TVD: a signature key and a encryption key (asymmetric), which is certified by the signature key. All appliances get the public key of the encryption key through a trusted channel.

A possible service of the TOM could be the management of S3 credentials, which could be saved into the sealed part of TOM and accessed only by Cloud Admins. This may be part of the TOM part of the 'Confidentiality proxy for S3' component described in Deliverable D2.4.1 (cf. D2.4.1, Chapter 8). More details on Key Management in the TOM component are described in the Deliverables D2.3.1 and D2.4.1 (cf. D2.3.1, Chapter 6 and D2.4.1, Chapter 9).

User Management

The administrators of a TOM need to administer users in groups, roles etc. This is needed for users in several TOM modules and for the administration users themselves. Different user classes need to be managed: TOM-Administrators and organization-specific users. The synchronization should be one-way, i.e., the TOM imports user information from ActiveDirectory (AD) or LDAP and updates its local data base. The TOM may add attributes locally. Especially the organization of users (groups, roles) should be synchronized with AD/LDAP. That is, an AD-/LDAP-group should be mapped to a TOM user-group.

Cluster Management

TOM will provide features to manage a cluster of TrustedServers and allocated TVDs. Building an automated, dynamic cluster infrastructure requires the ability to communicate, to collaborate, to integrate network and application network infrastructure with the TOM. This shall include the following functionalities:

- Migrate VM instances
- Load Balancing
- High Availability (HA)

Affected components are:

- TOM
- TrustedServer

Migrate VM Instances The Cloud Admin shall be able to migrate a VM instance from a TrustedServer$_A$ to a TrustedServer$_B$ without interrupting service. This shall be realized trough a manual drag and drop functionality in the TOM GUI and/or automated by a scheduler, e.g. for scheduled backup and/or maintenance of a TrustedServer. The TrustedServer component, together with the storage and network components used in the TrustedInfrastructure shall support and technically enable this. VM image and snapshots shall be saved in storage such as SAN, only information in RAM has to be sent from TrustedServer$_A$ to TrustedServer$_B$.

Load Balancing A load balancing functionality allows Cloud Admins to configure what load should be reached on each TrustedServer min/mid/max and if needed, evaluate and configure where to migrate VM instances. Therefore the TOM shall be able to monitor capacity for TrustedServer$_A$. The TOM then triggers an event that indicates a new instance is required to maintain availability. How the TOM determines capacity limitations is variable and might be based solely on VM status, data received from a load balancer, or a combination thereof. Afterwards a new instance is launched via the TOM and the TMC. The TOM grabs the IP address of the newly launched instance and instructs the load balancer to add it to the pool for resources. The load balancer adds the new VM instance the appropriate pool and as soon as it has confirmation that the instance is available and responding to requests, begins to direct traffic to the instance. This process is easily reversed upon termination of an instance. Other infrastructure components that are involved in this process must also be notified on launch and decommission.

High Availability (HA) High Availability (HA) is needed for the central management component TOM, for TrustedServer appliances and for each of the VM instances. This shall be realized through instant migration of VM instances and may be in hot standby mode.

Cloud Management

Based on Cluster Management the TOM will provide additional features to manage a TrustedInfrastructure-based cloud. This means an virtual infrastructure management capabilities with additional features:

- User-Interface
- User-Model
- Identity and Access Management (IAM) (maybe externalized)
- VM image management
- IP address management
- Key management
- Cloud Load Balancing

12.5 Trusted Virtual Domains

In this section we give a brief overview of the TVD concept and its features.

TVDs provide a key-concept of a secure IT infrastructure through offering a homogeneous and transparent data protection and enforcement of access control policies on data and network resources. TVDs are a suitable framework for the implementation of secure multi-domain / single-infrastructure computer networks such as centralized data centers, where computational resources from different owners share the same physical infrastructure, or a single organizational Local Area Network (LAN) that spans over different offices, branches or functional areas [CLM+10].

Amongst the strengths of TVDs is the *transparent data protection* and *enforcement of access control policies* — platforms and users logically assigned to the same TVD can access distributed data storage, network services, and remote servers without executing any additional security protocols, while the resources belonging to different TVDs are strictly separated. Those resources remain inaccessible for unauthorized participants. Moreover, data that is stored on mobile storage devices is automatically protected by encryption and can only be decrypted within the same TVD the device has been assigned to. Hence, users cannot forget to employ encryption, and data on mobile storage devices such as memory sticks cannot be used outside the TVD.

12.5.1 TVD Features Overview

A TVD is a set of compartments that trust each other, share a common security policy and enforce it independently of the particular physical hardware platform they are running on. Moreover, the TVD infrastructure contains the hypervisor, also called VMM, and the physical components, such as CPU, memory, and hardware security modules, on which the compartments

rely to enforce the policy. In particular, the main features of TVDs and the TVD infrastructure are [CLM+10]:

1. **Isolation of execution environments:** containment boundaries to compartments from different TVDs are provided by the underlying *secure hypervisor (Security Kernel and VMM)*, allowing an isolated execution of several different TVDs on the same physical hardware platform.

2. **Trust relationships:** a TVD policy defines which platforms (including *secure hypervisor*) and which compartments (on top of these platforms) are allowed to join the TVD. For example, platforms and their virtualization layers as well as individual compartments can be identified via integrity measurements taken during their start-up.

3. **Transparent policy enforcement:** the secure hypervisor enforces the TVD-based security policy transparently from the user or any applications running within compartments.

4. **Secure communication channels:** Compartments belonging to the same TVD are connected through a virtual network that can span over different platforms and that is strictly isolated from the virtual networks of other TVDs. Depending on the application scenario, different mechanisms, such as Virtual Private Network (VPN) providing encryption, can be used to secure the communication.

5. **Information flow control:** information can only flow between compartments belonging to the same TVD. This is defined by the information flow policies. There can be only one information flow (directly) between two end points.

6. **Central management:** main feature of the Trusted Infrastructure: different TVDs, their infrastructure, trust relationships, and security policies are centrally manageable. Centralized infrastructure management allows registration and authentication of all hardware platforms, security services and VMs. System-wide Security Policy Management defines allowed information flows between TVDs, Network Access Control (NAC) within TVDs, and user & role based policies.

12.5.2 TVD Architecture Overview

In this section the architecture overview, as well as its main components, are described. Figure 12.2 shows an example of the TVD architecture with two organizations, Organization$_A$ and Organization$_B$, and three TVDs, TVD$_1$ to TVD$_3$. Compartments belonging to different TVDs are strictly isolated, as well as organizations are.

The main components of a TVD-based architecture are:

- **Compartment:** a *compartment* is an active entity such as a VM or an application running in a VM. Attributes of a compartment are the name, the assigned TVDs and version.

- **Organization:** an *organization* defines a security policy based on a set of TVDs and information flows between them. Organizations are strictly isolated. Attributes of an organization are name, TVDs, information flows and users and roles.

- **Trusted Channel:** a standard approach for establishing secure channels over the physical network, e.g. the Internet, is to use security protocols such as Transport Layer Security

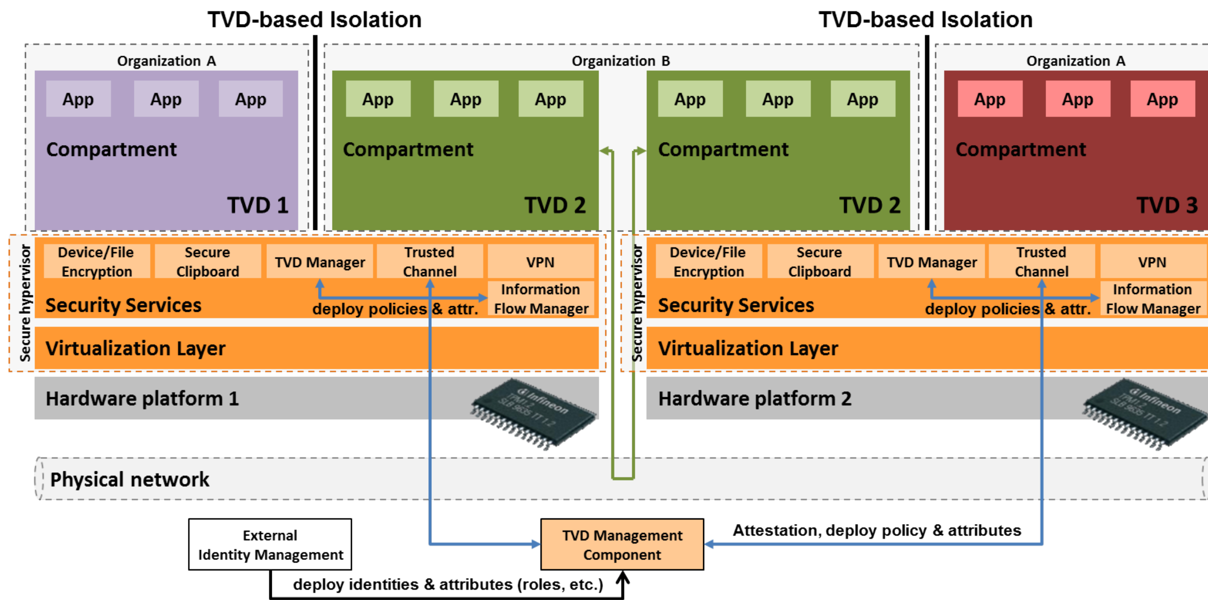

Figure 12.2: Overview of TVD architecture with logical view of three TVDs distributed over two physical platforms with four compartments and the physical deployment of the TVD components.

(TLS) [DR06] or Internet Protocol Security (IPsec) [KS05], which aims at assuring confidentiality, integrity, and freshness of the transmitted data as well as authenticity of the endpoints involved. However, secure channels do not provide any guarantees about the integrity of the communication endpoints, which can be compromised by malware. Based on security architectures that deploy *Trusted Computing* functionality, one can extend these protocols with integrity reporting mechanisms. Such extensions can be based on *binary attestation* or on *property-based attestation*. [CDE+10]

In the context of the TVD architecture the *Trusted Channel Daemon* establishes a secure management channel with configuration authentication between the *TVD Management Component* and the *Security Kernels* on the platform tht hosts the VMs.

Trusted channels are mainly used for mutual authentication between the *Security Kernel* and a remote management entity, the *TVD Management Component*. A trusted channel transfers the public key certificate of the *TVD Management Component* to the *Security Kernel*, and authenticates the *Security Kernel* to the *TVD Management Component* using a unique identity derived from the *TPM*, a part of the hardware platform, and an integrity proof using remote attestation.

Security Requirements:

1. Confidential, integer, and authentic channel to the management component
2. Remote attestation of the appliance's configuration

Functionality:

1. Registration and certification of appliances for TVD membership

2. Compartment download and update
3. Security policy updates
4. Security kernel updates
5. Backup and key storage

- **Information Flow Manager:** the secure hypervisor / SecurityKernel provides secure communication for intra-TVD information flows between local compartments and external entities by enforcing the TVD-based security policy. The *Information Flow Manager* component enables inter-TVD information flow by ensuring the enforcement of information flow policies and obligations if an information flow is permitted. An information flow policy includes information flow rules which are evaluated through a decision component in the *Information Flow Manager*. An inter-TVD information flow is only permitted if the evaluation leads to a positive result.

- **VPN:** the VPN security service provides encryption to secure the communication of the compartments by building up secure communication channels. Compartments on different platforms belonging to the same TVD are connected through this virtual network that is strictly isolated from the virtual networks of other TVDs.

12.5.3 Functionality and Management

In this section the functionality and management of TVD-based Trusted Infrastructures are described.

TVD-based Policies

TVD-based security policies include TVD attributes and information flows between TVDs. Those TVD-based policies are managed through the graphical user-interface (GUI) of TOM, which provide a tree of TVDs an Informational flow which could be described, configured and allocated per drag and drop mechanisms.

TVD policy model: since a TVD is a labeled security domain a TVD Policy is the machine-readable security policy of such a security domain that specifies all components that can be admitted to a TVD. It contains a basic form of access control rules and specifies the configuration of the TVD resources, such as virtual network. [CDE+09]

A TVD has the following (universal) attributes (non-exhaustive enumeration):

1. **ID** (internal and never shown to the user)

2. **name**

3. **color** (RGB 3*8bit for visualization of the TVD; it is expected that the color will be also used to illustrate information flows inside the management component of the TVDs)

4. **validity period** (i.e. how long it can be trusted that the cached information is correct when the appliance is not connected to a TVD Management Component)

5. **CA**(one for all or one for online and one for offline communication)

    (a) **certificate**

(b) **private key**

6. **encryption settings** for online and offline communication:

   (a) **IPsec** settings

      i. Mode
      ii. key lifetime
      iii. cipher
      iv. hash
      v. group and Perfect Forward Secrecy (PFS)

   (b) **file encryption** settings

      i. method
      ii. cipher / algorithm
      iii. modes of Operation
      iv. file type

   (c) **platform properties** (identities of the platforms that can host compartments of this TVD)

      i. platform identity
      ii. has TPM
      iii. accessible by identity / user / group / role *

TVDs have back pointers to the organization containing them and include basic access control rules which are based on users and roles which need a *User Management* and *Role Management* component, e.g., an external Identity Management (IdM) System which is queried by the *TVD Management Component*. The platform properties attribute never leaves the TVD Manager and specifies the identity of the platform that can host TVD compartments. The TCB of such a platform is remotely attested by the TVD Management Component: if it matches one of the platforms included in the platform properties, then the trusted channel is established.

The platform identities are specified as collections (platforms) of references to reports, generated by TPM. In this context, reports are, e.g., binary measurements. The reports therefore represent building blocks for a white list of allowed platforms. The identity clause for each TPM defines how these reports shall be authenticated by the remote attester. For example, it could be the fingerprint of the public part of the AIK, (or its X.509 certificate) used for the attestation. More details on Key Management in TVD-based infrastructures are described in Deliverable D2.3.1 (cf. D2.3.1, Chapter 6).

The TVD Policy is sent to computing platforms via trusted channel and is handed out to the TVD Manager that will spawn and configure the Security Services with the policy settings. The latter are then used to set up the TVD networks and to check whether a VM can be admitted to the TVD or not. Allowed TVD compartments are expressed as collections (platforms) of references to reports (as for nodes) to be matched and to resources to be attached to the VM, like logical networks and storage volumes. The configuration of each resource is also specified: usual Internet Protocol (IP) parameters and the encapsulation type (Virtual Local Area Network (VLAN) tagging, IPsec, etc.) for networks, the volume parameters and security features to apply for storage, like encryption. Other virtual resources with their configuration can be specified: they can also be indirectly attached to VMs, like virtual switches or VPN setups. [CDE+09]

The TVD Policy format allows the complete definition of TVD components and resources and can be further extended, e.g., to specify new types of resources. The structure of collections of reports simplifies the description of the allowed systems and makes it more compact. However, if the list of systems is large, evaluating a system against the policy during a remote attestation may require a considerable amount of time; this aspect requires further tests and analysis. XML-based policy languages may be unsuitable for implementing minimized components, e.g., the trusted channel, because they require the usage of large libraries. A simple to parse text format is required to express TVD-based security policies. [CDE+09]

12.6 Inter-TVD Information Flow Control

TVDs provide a secure infrastructure that enforces transparent information flow control on data and network resources through isolation mechanisms in a manageable fashion. The challenge is to expand the TVD technology with inter-TVD information flows based on policies which explicitly allow sharing of information between strictly isolated security domains. Everything exchanged between two TVDs, such as clipboard information or files, is called an information flow. Such an information flow has to be explicitly allowed by a decision component and to be enforced by an enforcement component. Both are part of the *Information Flow Manager*. The decision is done by evaluating an information flow policy.

12.6.1 State of the Art

A standard way to protect information flowing between different security domains is to provide access control mechanisms on every system information objects are located. This protects the information from unauthorized access. The information is only accessible if the identity, e.g. a human user or an IT system, is authenticated to the system and authorized to access the information, e.g. by a role and included privileges to access a selected file or service. Various software-based Identity and Access Management solutions exist on the market. They normally support Role-Based Access Control (RBAC) with privilege management functionalities.

The main problem of those access control systems and the implemented techniques is: once the authorization is done, the access is permitted and an authorized user can take full control of the information and, e.g. copy it on a mobile device so the protection is gone if this device is plugged into a system of another security domain with a different security level.

In addition to IAM solutions, which provide user management, privilege management and access control there are several Information Rights Management (IRM) products on the market today. Such products use data flow control mechanisms to handle the shortcomings of previously mentioned systems in case of outsourcing and information sharing between security domains. They promise to ensure automated protection of information, so information that needs to be shared can be protected automatically and without human intervention. One problem of IRM products is that the needed rights enforcement agent is running on untrusted hardware and there is no isolation between other applications running on the same operating system.

TVDs cover the shortcomings of IRM products with a trusted infrastructure, isolation mechanisms and information flow control between security domains and are well established today. However, at best of our knowledge, inter-TVD information flows are not possible since TVDs are strictly isolated through a secure hypervisor. The challenge is to improve TVD-based technology such as Trusted Infrastructure with a information flow control mechanism for inter-TVD information flow, e.g. to enable data exchange in collaboration scenarios such as typical out-

sourcing or Cloud Computing. In a TVD-based infrastructure data is transparently encrypted and bound to the TVD.

For example storing data on an USB device will result in an encrypted file on the stick which is automatically decrypted when the stick is used from the same or another compartment in the same TVD. If it is used in another TVD only the encrypted data is visible. So there can be data flow between different TVDs in the sense of encrypted data being visible in another TVD, but no information flow since the data can only be encrypted in the same TVD. The goal is to weaken this strict isolation and allow controlled information flows between different TVDs, without losing confidentiality of information.

12.6.2 Requirement Analysis

To reach the goal of a secure and manageable inter-TVD information flow this should be enabled and governed by a new component, the Information Flow Manager, which is not available in actual TVD-supporting architectures. This component complements the existing infrastructure and realizes a policy engine with a policy decision point and a policy enforcement point. Also there is the need of a suitable information flow policy framework which allows defining rules which use attributes such as system or user properties (e.g. available encryption and roles) to define when an information flow must be permitted.

Figure 12.3 shows an example of an information flow between two TVDs and the supporting architecture. Note here that evaluating an information flow policy, if applicable, always leads to a positive result. It allows an information flow and does not prohibit one. All information flows that are not covered by an information flow policy are implicitly forbidden.

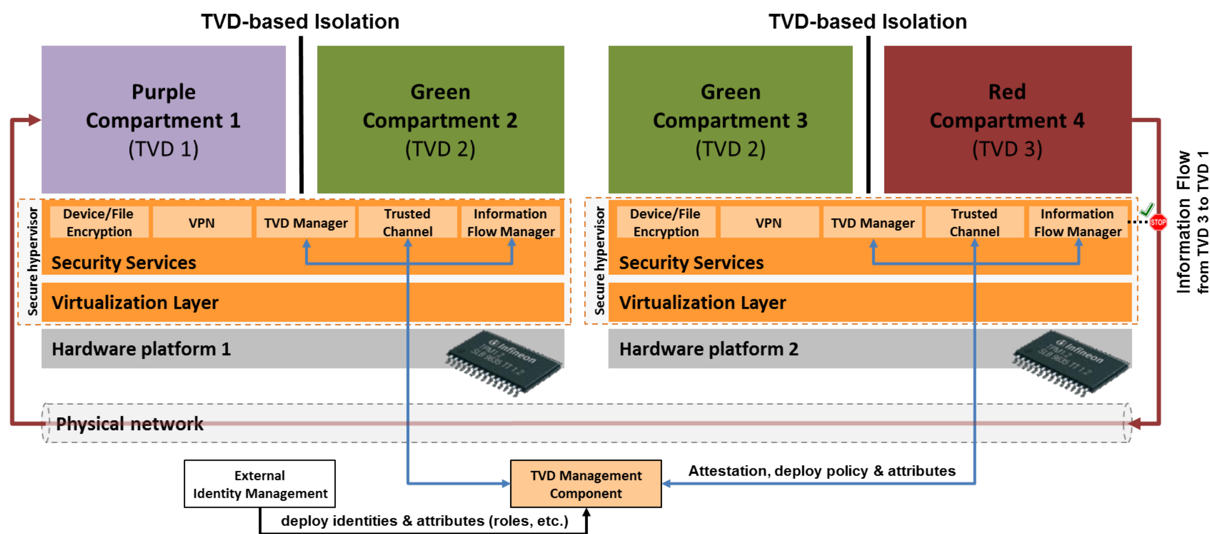

Figure 12.3: Example of an inter-TVD information flow between two TVDs on different platforms and supporting security services such as TVD Manager and Information Flow Manager.

TVD Manager The TVD Manager component receives the TVD-based security policies on its appliance from the central TVD Management Component through the Trusted Channel Daemon. The component contains a TVD policy engine which is able to enforce the TVD-based

security policies to each compartment that has the correspondent registered and certified TVD membership.

Information flow An information flow describes inter-TVD data exchange and has the following attributes:

1. ID
2. source (a TVD: foreign key/reference)
3. target (another TVD: foreign key/reference)
4. information flow policy

Note here that an information flow policy is always positive. It allows an inter-TVD information flow and does not prohibit one. All inter-TVD flows that are not covered by an information flow policy are implicitly forbidden, TVDs are strictly isolated.

Information Flow Manager The Information Flow Manager component enables inter-TVD information flow by ensuring the enforcement of information flow policies and obligations if an information flow is permitted. This component determines and applies (decision and enforcement) the current information flow policy depending on the current TVD ID of the compartment as stated by the TVD Manager. The Information Flow Manager is located in the secure hypervisor layer and is running as a security service.

Information flow policy framework and language An information flow policy includes information flow rules which are evaluated through the decision point part of the Information Flow Manager security service. An inter-TVD information flow is only permitted by the decision point if the evaluation leads to a positive result. Since information flow rules are always positive an information flow needs at least one applicable rule to be permitted. It has to be surveyed, which kind of language is needed to define and manage information flow policies.

Obligations If an inter-TVD information flow is permitted, then an obligation, set in an information flow rule, may be enforced before or after the information flows. An obligation is an operation which is set in an information flow rule and will be performed by the policy enforcement in conjunction with the enforcement of a decision about an information flow. Examples of obligations that might be expressed in an information flow rule are:

1. **Ask** - The user initiating the information flow has to confirm the information flow.
2. **VirusCheck** - Check the information for viruses and allow the information flow only if no viruses are found.
3. **Log** - The information flow will be logged. This may be configured further as to what attributes are logged.
4. **CheckUser** - Only specific users are allowed to initiate the information flow.

The list of obligations is not meant to be exclusive, but rather gathers a basic set of obligations which are enforced before or after an information flow is permitted. The information flow policy and the respective components should be designed in an extensible way to allow the integration of more obligations that may be required in the course of the project. Examples of such extensions are anonymization of data or obligations on the file type, e.g. information flow is permitted to a user but only if the file type is "pdf".

Enforcement of information flow policies An information flow is allowed by the decision point, which is part of the Information Flow Manager, if it at least conforms to one valid information flow policy. This is the case if at least one information flow rule results with true. Otherwise the information flow is declined. There is an invariant for every information flow:

- source.organization == target.organization

The offline validity, the time that the computing platform may be offline from the TVD Management Component and still trust the configuration, is the minimal offline validity of source and target, because an illegal information flow may break confidentiality with the source and integrity with the target, so both may be affected by an illegal information flow. The Information Flow Manager enforces every information flow policy that was deployed by the external TVD Management Component. Information flow objects may have redundant back pointers to the organization containing them.

Enforcement of obligations If an information flow is allowed by the Information Flow Manager one or more applicable rule(s) could define an obligation and when it has to be enforced. If an obligation, e.g. a *VirusCheck*, has to be enforced on the source TVD before data is send to the target TVD, then the enforcement point has to invoke the Anti-Virus-Software in a special compartment, wait for the result and if the file is not positive for malware, then enforce the information flow. If the file is positive for malware, then no information flow should allowed and the user compartment should be inform the user of what happened. This is another possible obligation which has to be enforced only if the *VirusCheck* obligation had a positive result. A *Log* obligation could have to be enforced before or after an information flow was allowed. For example an information flow rule could state, that all information flows, a special role (or group of users) with a special target initiate, have to be logged before the information flows between the source and target. Further, such a rule may state, that the information flow is commited only if the logging-service return to the enforcement point that the logging has been completed (another dependent obligation).

Policy management tools On the one hand an information flow policy framework as described above needs a feature-rich *policy management* with tools to define those policies and rules, on the other hand there should also be visualization tools and auditor tools. In scenarios with a high number of TVDs and information flows between them there is a need to check the policies and rules. An auditor tool as part of the policy management simplifies the analysis and validation process of information flow policies.

Users of the policy management tool can analyse information flow policies and their effects by writing and submitting a wide range of queries to the auditor component. The types of queries may be:

1. Who (user/role) can initiate a specific information flow (source/target)?

2. What obligations must be fulfilled if a user with role X is permitted to send files with file type Y?
3. What obligations has a specific information flow (from TVD(ID) to TVD(ID))?
4. Under which obligation can the information flow be allowed?

12.7 Conclusion

We conclude that the features of *Trusted Infrastructures* as described here enable outsourcing of infrastructures into an IaaS cloud without losing control of the components and trust into such a infrastructure. Extensions to existing Trusted Infrastructures components which will be developed in TClouds project will provide Cluster Management and Cloud Management. Key Management, as fundamental part of Management of Trusted Infrastructures, could be expanded in order to use Cloud Computing with multiple organizations. This is described in Deliverable D2.3.1 (cf. D2.3.1, Chapter 6). TVD-based Information Flow Control techniques and trustworthy components supporting these could solve problems some IRM products have today and allow secure and privacy conform sharing of information, e.g. in storage clouds.

Figure 12.4 shows an example of an holistic TVD-based trusted infrastructure use-case with information flows between different TVDs in different locations and on different platforms, such as server, desktops and mobiles (see Figure 12.4).

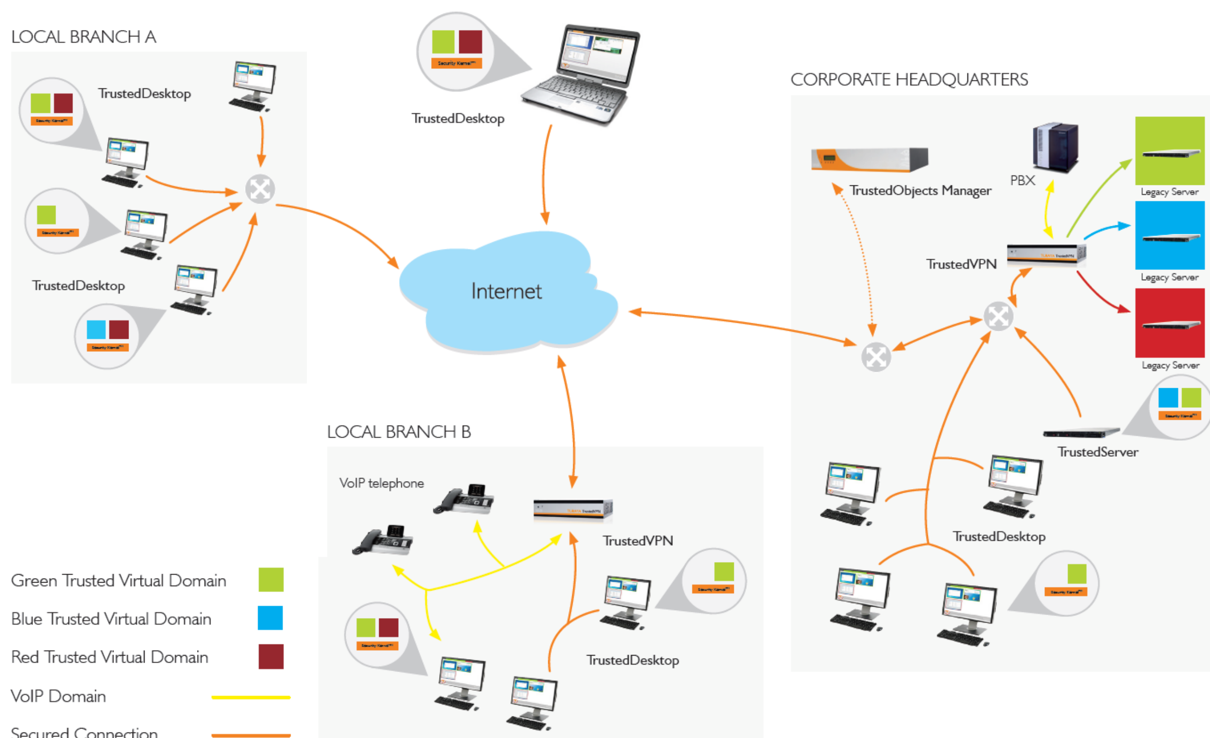

Figure 12.4: Overview of a TVD-based trusted infrastructure use-case.

Chapter 13

Recursive Virtual Machines

Chapter Authors:
Allysson Bessani, Bernd Kauer, Paulo Verissimo (FFCUL)

13.1 Motivation

Virtualization has contributed to a better resource utilization of a machine. Its generalized use is bound to be amplified by the emerging Cloud Computing paradigm. In fact, Cloud Computing is essentially a business model, made possible by some technological advances that enable modularization of resources. Virtualization is probably the most important of these technological capabilities and as such has been deserving considerable attention in cloud-computing related research, since it is not exempt from drawbacks. One issue we address in this report is the conflict between performance and security.

Virtualization was used to improve the security of a machine's OS or applications: hypervisors spawn virtual machines to protect the kernel code [SLQP07] or application data [CGL+08]; they provide a trusted execution environment [MLQ+10b] and implement fault tolerance [BS95]; they detect intrusions [GR03] and help in analyzing them [DKC+02]. There are different reasons why virtual machines are an appealing environment to improve an operating system. First, they allow to execute the target OSes unmodified. This minimizes the development effort and allows to reuse closed source software. Secondly, a virtual machine provides an already established interface that is independent of OS APIs. Porting a solution to another OS is therefore relatively easy. Finally, security-critical code can be protected against attacks by relying on the virtualization layer for additional security defense lines.

A drawback of VMs is the performance overhead of a fully virtualized hardware architecture. Fortunately, the introduction of hardware support for CPU virtualization [UNR+05] and nested paging [BSSM08] have significantly reduced these overheads [SK10]. Furthermore, they have simplified hypervisor implementations, because they made older techniques such as binary translation [AGSS10] obsolete. However, the success of hardware support has led to another problem: operating systems themselves like to run hardware-accelerated virtual machines. Windows 7, for example, relies on VMs to start legacy XP applications. The security layers now compete with the OS to be the hypervisor, because the current hardware only supports one of them in the system.

Such a restriction can be avoided by nesting two hypervisors [GR09, YDD+10]. One becomes the outer hypervisor (HV) and runs directly on the hardware, whereas the other one will run inside the virtual machine the first HV provides. However, nested virtualization is not sufficient, if more than two layers need to be available. Generalizing nested virtualization

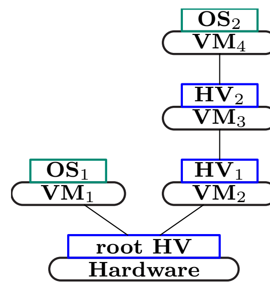

Figure 13.1: Recursive virtualization with three hypervisors. The root HV multiplexes the hardware, whereas the other hypervisors wrap the OS to implement security functions.

leads to *recursive virtual machines*, where the nesting depth is only limited by the platform resources [PG74]. Figure 13.1 shows an example where two OSes are running inside recursive virtual machines. The first OS runs directly on the root hypervisor. The second OS is wrapped by three hypervisors. The intermediate hypervisors (HV_1, HV_2) can be used to provide additional security features to the system.

Nevertheless, there is a major problem to be solved before the benefits of recursive virtualization can be fully enjoyed. Previous work has argued that hypervisors based on hardware architectures can not be efficiently nested, as the performance *worsens exponentially with* the *stacking depth* [FHL+96]. One challenge of this report is to show that virtual machines can indeed be stacked with a performance overhead that is (largely) independent of the nesting level. Once this is solved, the doors are open for the design of advanced security mechanisms, and this avenue will be further explored in the report.

Structure

The rest of this report is organized as follows: We first review related work (Section 13.2). Then we analyze the overheads of existing nested VM approaches (Section 13.3). We introduce a new design for recursive virtualization (Section 13.4) that avoids these performance penalties. Furthermore, we suggest several ways in which to take advantage of high-performance recursive virtualization to improve OS and application security, in Section 13.5. Finally, we draw conclusions (Section 13.6).

13.2 Related Work

The theoretical requirements of recursive virtualization were first described in the 70s by Popek and Goldberg [PG74]. Multiple hardware architectures were proposed to build recursive virtual machines for example by Belpaire and Hsu [BH75].

More recent work has focused on nesting two VMs on top of hardware support for virtualization. Graf and Roedel [GR09] show for the first time that nested VMs can be efficiently implemented with AMD processors. Later, Yehuda at al. [YDD+10] extended nested virtualization to Intel's virtualization extension (VMX). They also described the folding of two layers of nested page-tables and the use of direct device assignment to optimize the I/O path. For more than two layers of virtual machines they assumed that the *emulation of VMX can work recursively*. However, a recursive emulation leads to a cascade of traps, which is known for

overhead that scales exponentially with the nesting level [FHL+96]. We show in this report how this overhead can be avoided.

Poon and Moon [PM10] have proposed a hardware extension that reduces the overhead of event delivery in recursive virtual machines from exponential to linear runtime. Our solution does not rely on any hardware extension for synchronous events but still requires only linear or even constant overhead. Furthermore, we will explain why their extension is insufficient for asynchronous events such as interrupts.

In the meantime nested virtualization has become an established feature in x86 virtualization environments such as KVM [YDD+10] or VMWare ESX [AGSS10].

From an operating system point of view, Fluke by Ford et al. [FHL+96], is most similar to our work. Fluke is a software-based virtualizable architecture that provides *recursive* OS containers or *virtual machines*. They aimed not at security but modularity and extensibility of the operating system. Within the VMs, Fluke provides a virtualized system-call interface instead of an existing hardware architecture. Interestingly, they excluded VMs and hypervisors based on hardware interfaces for two reasons. First, they cause exponential overhead. Second, they can not provide shortcuts in the chain of virtual machines. We will show in this report how these two issues can be solved.

13.3 Nested Virtualization

Nested virtualization is a feature of a hypervisor that allows its VMs to act as hypervisors and start virtual machines themselves. Even though a hypervisor might support only a single layer of nesting, recursive virtualization can be implemented with this technique, if all intermediate hypervisors support nested virtualization.

In previous sections we have claimed that nested virtualization should not be used as the building block for recursive virtualization due to its low performance in deeply nested scenarios. This mainly stems from the use of trap-and-emulate style of virtualization: The parent hypervisor multiplexes the CPU between its child VM and the nested grandchild VM, by trapping the virtualization instructions executed by the child and emulating their behavior.

In the following we will analyze the overheads of nested virtualization. This should give us a better understanding of how the performance scales with the nesting depth and what problems our design should avoid.

Classifying the Overhead

We can distinguish three classes of overhead when using nested virtualization multiple times: constant, linear, and exponential.

Constant $O(1)$ overhead occurs when an event can be completely handled within a single layer (hardware or hypervisor) without causing further events. One example are untrapped instructions, if we do not consider cache and TLB effects. Another example are trapped instructions such as `cpuid` that can be completely handled by the parent hypervisor.

The overhead scales *linearly* with the nesting depth n, if a single event is propagated through the hierarchy: $O(n)$. Such an event can flow down the hierarchy. One example is the delivery of interrupts, where the parent hypervisors will propagate the IRQs to one of their children. The event can also flow up the hierarchy. If a child programs one of its virtual devices, for instance a timer, and the parent needs to program its timer as well, to emulate the side effects of the original operation.

AMD SVM	Intel VT
`clgi`	`vmread(exit-reason)`
`vmload(child-state)`	`vmread(exit-qualification)`
`vmrun(child)`	`vmread(instruction-pointer)`
`vmsave(child-state)`	`vmread(instruction-len)`
`vmload(parent-state)`	`vmwrite(instruction-pointer)`
`stgi`	`vmresume(child)`

Figure 13.2: Virtualization instructions usually executed by a hypervisor to handle a single trap on AMD and Intel CPUs. A nested hypervisor will need six virtualization instructions to emulate one virtualization instruction from its child.

The worst case are cascading effects which will lead to *exponential* overhead: $O(b^n)$ with a branching factor $b >= 2$. This can happen if a single event of a child causes the parent to generate multiple events of the very same type. One example are the emulation of virtualization instructions. Figure 13.2 shows that emulating a single virtualization instruction requires the parent to execute six virtualization instructions itself for the AMD and Intel versions of x86 CPU-virtualization extension. A grandparent that emulates these six instructions would already need 36 of them. Another example is the page-table lookup in multiple layers: a single memory operation can result in five memory accesses[1] with a four-level page-table and in 25 when two layers of page-tables are involved [BSSM08].

Nesting Limit of VMs

Based on these observations we can now estimate the maximum number of virtual machines that can be nested until the system will only handle interrupts and will not make other progress anymore. Since the exponential part has the most influence, we will ignore the constant and linear overhead in this estimation.

We assume a simplified scenario of nested virtual machines: At the beginning the deepest VM runs. When an external interrupt, such as a scheduling timer tick for the deepest VM, arrives at the root hypervisor all nested VMs are stopped. The nested hypervisors then recursively forward the timer interrupt and resume its virtual machines. This generates an exponential number of events depending on the number of virtualization instructions used in a single layer to resume a child VM. This number is equivalent to the branching factor of the event tree.

We now calculate for this scenario the maximum number of virtual machines that can be nested and still sustain a given interrupt rate. We vary the IRQ frequency between 1 Hz and 1000 Hz and consider a branching factor between 2 and 10. We assume a 3 GHz machine that can handle a single trap of a virtualization instruction within 1000 cycles. This is slightly faster than what was previously reported [SK10]. Furthermore, we do not take into account that handling certain traps will be more costly than others.

Figure 13.3 shows the results of our estimation. If we are interrupting the virtual machine once every second and assume a branching factor of six, to emulate the six instructions from Figure 13.2 to resume a virtual machine, we are restricted to nine nested VMs. If we assume a more realistic frequency of 1000 Hz for a scheduling timer, we cannot nest more than five VMs!

[1] Excluding access- and dirty-bit updates.

Branching Factor	Interrupts per Second 1	10	100	1000
2	22	19	15	12
4	11	10	8	6
6	9	8	6	5
8	8	7	5	4
10	7	6	5	4

Figure 13.3: Maximum number of nested VMs for different branching factors that can sustain a given interrupt frequency. Calculated for a 3 GHz machine where a single trap can be handled within 1000 cycles.

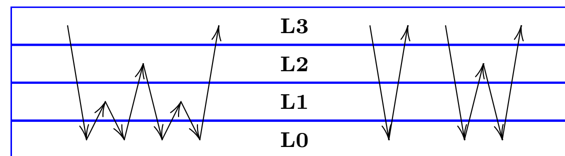

Figure 13.4: Event-flow within three levels of VMs. With nested virtualization (left) an event causes an exponential number of traps to be forwarded through the hierarchy. In our design either the root hypervisor can handle the event itself (middle) or it forwards it directly to the upper layer (right).

13.4 A Novel Design

The previous section has shown that nested virtualization cannot be used as building block for recursive virtualization, because exponential overheads limit it to a handful of nesting levels. Any design for recursive virtualization should therefore avoid any exponential overheads. In this section we present a novel design that achieves this.

The core idea of our design is the following: instead of repeating the support for nested virtual machines in every layer, we just implement recursive VMs in the root hypervisor. This approach has several advantages. Foremost, all virtualization instructions can be already emulated at the root and need not to be propagated trough the hierarchy. Other virtualization events can be directly forwarded to the right upper layer, since the root hypervisor knows all VMs in the system (See Figure 13.4). Furthermore, the root hypervisor can aggressively cache intermediate values to reduce the overhead. Finally, upper layers can be simpler as they do not need to be involved or in most cases even aware of recursive VMs. In the following we will describe how we implement this idea.

13.4.1 CPU Virtualization

If an upper layer parent wants to start a child VM, it uses an architecture specific instruction, such as `vmrun`. This instruction traps to the root hypervisor which can not directly execute it on behalf of the parent, since the child VM state is specified relative to the parent VM. For example, the memory for the new VM is a subset of the parent address space, but to execute a VM on a physical CPU, its memory has to be taken from the host memory.

Thus, the root hypervisor synthesizes for the child VM a corresponding *shadow VM* that can be directly executed on the hardware. This approach is similar to a shadow page-table algorithm,

where the hardware is using shadow page-tables synthesized by the hypervisor, instead of guest specified page-tables.

Creating Shadow VMs

Shadow VMs are created by *merging* the parent and child VM state in the following way:

- All guest state is directly taken from the child. For example the CPU registers, the event injection, and the exit information are reused unmodified.
- The access rights to the platform resources are calculated by taking the intersection of both VMs' access rights. The two nested page-tables for example are folded and the two I/O permission bit maps are AND-ed together. We thereby guarantee that the parent can isolate its child VMs and that a child has no more rights than its parents.
- The intercept bitmaps of instructions, exceptions and asynchronous events for the shadow VM are the union of what both VMs would use. This makes sure we do not lose any event the parent or the grandparents are interested in.

Please note that the hypervisors can be used unmodifed, as no support is needed from them for creating shadow VMs. Furthermore, merging is independent of the VM nesting depth, since only the parent and child state are involved. A hierarchy of virtual machines is flattened into a single layer by the root hypervisor by applying this process recursively.

Lazy and Eager Merging

Merging can be an expensive operation, as it might touch several megabytes of memory. We therefore optimize this process through *lazy merging*. The nested page-tables are a good example for this. In the beginning the shadow VM starts with an empty nested page-table. Whenever the VM accesses memory that is not present, the root hypervisor will observe a nested page-fault and merges the corresponding page-table entries. In fact all resources can be lazily merged, if their absence generates a trap to the root hypervisor and if the operation that caused the trap can be resumed. This includes I/O permission and MSR bitmaps but excludes the I/O MMU page-tables, because DMA operations are usually not restartable.

Recursive virtualization with an Intel VT interface can also benefit from *eager merging*: Because the root hypervisor observes all updates (`vmwrites`) to the child VM state, it can directly merge the data into the shadow VM. When the child should run, the shadow VM can be activated without the need to merge anything anymore. This drastically reduces the runtime overhead since only the subset of the state that was modified has to be merged.

Forwarding Virtualization Events

The root hypervisor receives all virtualization events and has to forward them to the correct upper-layer hypervisor. If a virtualization event occurs, the root hypervisor inspects the original intercept bitmaps of all parents of the currently running virtual machine. Inspection is done in either top-down or bottom up direction, depending on the type of event. Asynchronous events such as interrupts, and machine check exceptions are delivered bottom-up and synchronous events, such as exceptions and instruction intercepts, are delivered in a top-down manner. The first intermediate hypervisor that has the corresponding bit in the intercept bitmap of its child set will receive the event. The overhead for event delivery scales at most linearly with the nesting

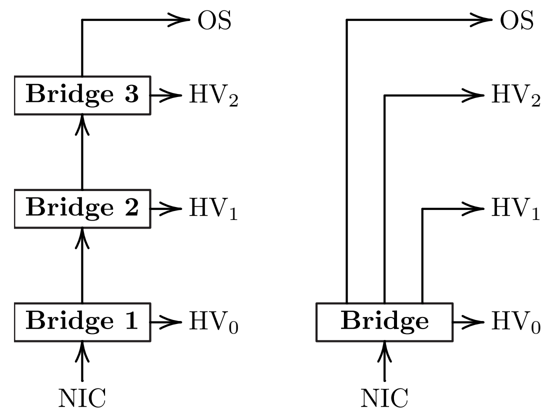

Figure 13.5: Bridging cascade (left) vs. direct assignment of virtual NICs (right). Traditionally every virtualization layer implements a bridge to multiplex its network card with his child VMs. This forces packets through a deep bridging cascade. In our design the root layer can emulate multiple virtual NICs and recursively assign them to upper layer VMs. Thus only a single bridge is involved in network packet handling.

depth, because only the state of the current VM and its parents has to be inspected. Please note that we do not require an interface extension for this, in contrast to previous solutions [PM10].

Event delivery can also be performed in constant time, because the destination for an event can be cached in a per shadow VM lookup table, if the root hypervisor can observe all state updates that invalidate this cache. This holds true for example with Intel VT based VMs, where only `vmwrite` instructions, which are emulated by the root hypervisor, can be used to modify VM state.

13.4.2 I/O Virtualization

The I/O path can have a huge influence on the overall performance of a virtualized system. [YDD+10] has shown that directly assigning a hardware device to a nested VM improves the performance, because hypervisors are no longer involved in most device operations. A register access for example will not cause a trap anymore, but will go directly to the hardware. Furthermore, DMA requests issued by a device are validated with an I/O MMU. This restricts DMA to the memory of the nested VM which drives the device. This allows to put drivers in untrusted VMs while still protecting the hypervisor from DMA attacks. Thus, we also use this technique and directly assign physical devices to recursive VMs.

However, not all VMs can benefit from direct assignment because there are usually more VMs than hardware devices in a system. This holds especially true where saving physical resources is one of the main incentives such as in server consolidation scenarios. Therefore, we also *assign virtual devices* through the VM hierarchy. The root HV, for example, can create multiple virtual NICs, which are recursively given to upper level VMs. In this way a received packet needs to be bridged only once, instead of going to a bridging cascade (See Figure 13.5). Since every bridging usually involves a copy of the packet, the performance gain of this approach will be significant.

Deep Intercepts

Direct assignment of virtual devices allows to implement shortcuts between different layers. Say for example a layer two (L2) hypervisor implements a virtual network card for a L3 VM, that is directly assigned by L3 to L4 and by L4 to L5. If L5 accesses a register on this virtual NIC, it will trap to the root hypervisor which forwards it to L2, thereby removing L3 and L4 from the chain.

This example leads us to the *deep intercept problem* of recursive VMs: In some point in time L2 needs to handle a trap from L5. However, L2 does not known L5 directly, but for handling the trap it needs to access the state of L5. One reason might be to figure out which of the memory it has given to L3 is available to L5. Therefore, L2 has to be able to recover the state of L5 by just looking at its direct child L3.

To handle deep intercepts a HV needs to know whether its child runs a nested VM and where the state of this VM can be accessed. Unfortunately, neither Intel's VT nor AMD's SVM currently provide these two pieces of information. Because we want to support deep intercepts for performance reasons, we extend the virtualization interface in the following way: We introduce a new *VM running* exit reason to decide whether the VM has exited or if it is still running and a nested VM has caused this exit. Furthermore, we publish the VMCS pointer of a VM in its shadow VMCS for Intel VT based VMs. This makes it accessible to its parents via `vmread`. On AMD SVM, we use the RAX register which already points to a child VMCB if the last operation of the VM was to run a child. These two simple extensions are sufficient for a hypervisor to search for the nested VM that caused the trap and handle it.

IRQ Forwarding

We rely on direct assignment and I/O MMUs to speed up the control and data path to a device. This leaves interrupts as the *the biggest cause for the remaining overhead* [YDD+10]. In fact, interrupt (IRQ) delivery is the only operation that still goes through the VM hierarchy. It is therefore desirable to minimize the number of IRQ forwarding steps. Another approach would be to reduce the interrupt frequency through IRQ coalescing. However, this causes a larger latency which might be undesirable for certain applications.

An interrupt can be directly delivered to the leaf VM, if the root HV would knew to which child VM the intermediate hypervisors will forward the IRQ and how they will translate the IRQ vector. Vector translation is important with Message Signaled Interrupts, where a VM can freely choose its destination vector for a device interrupt. Unfortunately, neither AMD nor Intel currently allow to specify this information in their virtualization extensions. Related work [PM10] has proposed a simple redirection bitmap that decides whether an interrupt is forwarded or not. However, this is insufficient as it does not allow vector translation. Thus, to optimize the interrupt path for recursive VMs we developed our own extension.

We extend the VM state with a pointer to an IRQ redirection table. This table is indexed by the interrupt vector and reveals the destination vector and a pointer to a child VM. If the root hypervisor should deliver an IRQ, it traverses the hierarchy of redirection tables until a leaf is reached or a maximum number of forwarding steps is done. Limiting the number of lookups avoids a DoS attack against the root HV, where an untrusted VM creates a loop inside the table or a large number of VMs. At the destination VM the root HV sets the corresponding bit in the virtual APIC IRR, as it cannot directly inject the IRQ, because the VM might be in an IRQ shadow or even not running at all. If the virtualization interface does not provide a virtual IRR register, it can be shadowed by the root HV. This hardware extension can drastically reduce

the number of events even in single HV scenarios. Please note that we do not optimize the handling of end-of-interrupts. In contrast to previous work [YDD+10], the overhead for them is independent of the nesting depth in our design, because we can handle them locally in the parent hypervisor.

13.4.3 Discussion

We solved the two issues mentioned by Ford et al. [FHL+96] that made recursive virtual machines based on a hardware interface infeasible. Our design avoids the exponential overhead caused by recursive trap-and-emulates, because the root hypervisor flattens the hierarchy and handles all virtualization instructions itself. Furthermore, shortcuts in a chain of virtual machines can be implemented by directly assigning virtual devices through the hierarchy.

Our design shows that a hypervisor does not need to provide the very same interface to its virtual machines which it gets from the underlying hardware. This is in-line with the observation that current hypervisors provide only a subset of the hardware functionality to their virtual machines. Especially seldom needed and optional features, for instance CPU power management, or FPU extensions such as 3DNow!, are usually ignored or only partially implemented. A virtualized OS will still work by detecting missing features and adopting accordingly.

However, we can go a step further and extend or partially replace the interface by the one that leads to the best performance for a given scenario. Announcing and supporting, for example, nested paging for child VMs is possible, even if the underlying hardware does not support it. We can also virtualize Intel VT extensions on an AMD CPU and vice versa. This has the nice consequence that all hypervisors running on top of the root hypervisor can be simpler than state-of-the-art implementations. If the root hypervisor provides a richer interface and already supports for example recursive virtualization, nested paging and direct execution in real-mode, there is no need to have support for nested virtualization, a virtual TLB, or a real-mode emulator in the higher layers anymore.

Finally, the root hypervisor can easily benefit from hardware acceleration. This includes our virtualization extensions, merging nested page-tables in hardware and providing a selective `vmwrite` intercept. Our design allows a clear upgrade path, where parts of the root HV functionality can move step-by-step from software into the CPU.

13.5 Advanced Security Mechanisms

By solving the performance problem of hypervisor-based recursive virtualization we opened the doors for the design of advanced security mechanisms. In the following we will shortly discuss the different ideas:

TCB reduction

Recursive virtualization allows the fine-grained decomposition of the virtualization environment into multiple hypervisors. This could even further reduce the TCB compared to [SK10].

Minimal Hypervisors

A minimal hypervisor has to reserve some memory for itself and can give all other resources to a child VM. It can periodically wakeup or trigger on a register access to perform its (security) task. Except for the initial configuration, it does not need to be involved in virtualizing the child

VM. Furthermore, it does not need to virtualize devices, if it just assigns them to its child. This should make a single upper-level virtualization layer very small.

Thin Security Layers

Thin virtualization layers will improve the security of legacy operating systems, without degrading performance. These layers can be independently developed from different parties and later freely combined into a single system, because they implement the very same machine interface.

Defense in Depth

Multiple security layers below the OS can provide in-depth barriers of several kinds, such as firewall-like filters, wrappers, failure and intrusion detectors, etc.

Intrusion and Fault Tolerance

Sophisticated micro-middleware structures could achieve fault and intrusion tolerance of an OS and upper layer hypervisors. Diverse replicas can be used to transparently mask intrusions, and to provide fault-free support [VCNS09]. VMs may also be rejuvenated periodically for proactive recovery [SBC+10] to heal potential intrusions.

Cloud computing

Fast recursive virtualization is especially interesting for a cloud-computing environment as it provides security and flexibility with very little overhead. Supporting it in a cloud would allow a user to freely select its hypervisor. It would improve diversity as multiple management layers can coexist in a single system. Finally, it makes hypervisor updates easier, because it allows to move a hypervisor one layer up and start the new version in parallel. Virtual machines can then migrate locally from the old to the new version.

Further developing these ideas is part of ongoing work.

13.6 Conclusions

In this report, we showed that current approaches for recursive virtualization on x86 are plagued by exponential overheads, and we proposed a new design that can avoid these overheads, scale beyond a handful of layers and minimizes the burden for higher layer hypervisors.

Furthermore, we discussed that fast recursive virtual machines will open the field for new security-related mechanisms. We advanced several ideas that concurred to this objective. We are currently working to finish our implementation and evaluate the performance characteristics of recursive virtual machines on x86.

Bibliography

[Abb] Imad M. Abbadi. Middleware Services at Cloud Application Layer. In *IWTMP2PS '11: Proceedings of Second International Workshop on Trust Management in P2P Systems*.

[Abb11a] Imad M. Abbadi. Clouds' Infrastructure Taxonomy, Properties, and Management Services. In *CloudComp '11: Proceeding of the International workshop on Cloud Computing: Architecture, Algorithms and Applications*, LNCS. Springer-Verlag, July 2011.

[Abb11b] Imad M. Abbadi. Self-Managed Services Conceptual Model in Trustworthy Clouds' Infrastructure. In *Workshop on Cryptography and Security in Clouds*. IBM, Zurich, March 2011. http://www.zurich.ibm.com/ cca/csc2011/program.html.

[Abb11c] Imad M. Abbadi. Toward Trustworthy Clouds' Internet Scale Critical Infrastructure. In *ISPEC '11: in proceedings of the 7th Information Security Practice and Experience Conference*, volume 6672 of *LNCS*, pages 73–84. Springer-Verlag, Berlin, June 2011.

[Acc08] Rafael Accorsi. Automated counterexample-driven audits of authentic system records. 2008.

[Aci07] Onur Aciiƈcmez. Yet another microarchitectural attack: exploiting i-cache. In *CSAW '07: Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pages 11–18. ACM, 2007.

[ADDS91] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens. Transaction security system. *IBM Systems Journal*, 30(2):206–229, 1991.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the Clouds: A Berkeley View of Cloud Computing, 2009. http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[AGrS+08] Frederik Armknecht, Yacine Gasmi, Ahmad reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An efficient implementation of trusted channels based on openssl. In *Computer and Communications Security*, pages 41–50, 2008.

[AGS+08] Frederik Armknecht, Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, Gianluca Ramunno, and Davide Vernizzi. An efficient implementation of Trusted Channels based on OpenSSL. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 41–50, New York, NY, USA, 2008. ACM.

[AGSS10] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. The Evolution of an x86 Virtual Machine Monitor. *SIGOPS Oper. Syst. Rev.*, 44:3–18, December 2010.

[Aim10] M.D. Aime. Trusted distributed log services. In *Wireless Conference (EW), 2010 European*, pages 488 –495, april 2010.

[AL11] Imad M. Abbadi and John Lyle. Challenges for provenance in cloud computing. In *in 3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP '11)*. USENIX Association, 2011.

[All10] Cloud Security Alliance. Top threats to cloud computing. https://cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf, 2010.

[Amaa] Amazon EC2 Pricing. http://aws.amazon.com/ec2/pricing.

[Amab] Amazon Machine Images (AMIs) . http://aws.amazon.com/amis.

[Ama10a] Amazon. Amazon Elastic Compute Cloud (Amazon EC2), 2010. http://aws.amazon.com/ec2/.

[Ama10b] Amazon. Amazon Simple Storage Server (Amazon S3), 2010. https://s3.amazonaws.com/.

[Ama11a] Amazon Elastic Compute Cloud User Guide, 2011. http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf.

[Ama11b] AWS Developer Forums: Email from Amazon EC2 about my AMI being compromised, April 8 2011. https://forums.aws.amazon.com/thread.jspa?messageID=235613.

[Ama11c] Compromised EC2 image includes root access SSH key, 2011. http://pastebin.com/q1VH4rmF.

[App] Apple App Store Review Guidelines. http://developer.apple.com/appstore/guidelines.html.

[ASM09] N. Arvantis, M. Slaviero, and H. Meer. Clobbering the Cloud!, 2009. http://www.sensepost.com/labs/conferences/clobbering_the_cloud.

[AT07] A. Abdelhalim and I. Traore. The impact of google hacking on identity and application fraud. In *Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 240 –244. IEEE, 2007.

[AWSa] All Together Now: Amazon, we need those caps on billing. http://forums.aws.amazon.com/thread.jspa?threadID=50075#jive-message-217130.

[AWSb] Amazon Web Services. http://aws.amazon.com.

[AWSc] Amazon Web Services Documentation. http://aws.amazon.com/documentation.

[AWS10] Amazon Web Services: Overview of Security Processes, August 2010. http://awsmedia.s3.amazonaws.com/pdf/AWS_Security_Whitepaper.pdf.

[BCF+10] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filiatre, Claude Marche, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specification langauge version 1.5. http://frama-c.com/download/acsl_1.5.pdf, 2010.

[BCG+06] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. v. Doorn. vTPM: Virtualizing the Trusted Platform Module. In *USENIX Security Symposium (USENIX'06)*, pages 305–320. USENIX Association, 2006.

[BCG+08] Adam Beautement, Robert Coles, Jonathan Griffin, Christos Ioannidis, Brian Monahan, David Pym, Angela Sasse, and Mike Wonham. Modelling the human and technological costs and benefits of usb memory stick security, 2008.

[BH75] Gerald Belpaire and Nai-Ting Hsu. Hardware Architecture for Recursive Virtual Machines. In *Proceedings of the 1975 annual conference*, ACM '75, pages 14–18, New York, NY, USA, 1975. ACM.

[BPE+10] Spyros Blanas, Jignesh M. Patel, Vuk Ercegovac, Jun Rao, Eugene J. Shekita, and Yuanyuan Tian. A comparison of join algorithms for log processing in MapReduce. In *SIGMOD '10: ACM International Conference on Management of Data*, pages 975–986, Indianapolis, IN, USA, 2010.

[BPN+11] Sven Bugiel, Thomas Pöppelmann, Stefan Nürnberger, Ahmad-Reza Sadeghi, and Thomas Schneider. Amazonia: When elasticity snaps back. In *18th ACM Conference on Computer and Communications Security (CCS'11)*. ACM, Oct 2011.

[BS95] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 1–11, New York, NY, USA, 1995. ACM.

[BS10] F. Baiardi and D. Sgandurra. Securing a community cloud. In *Proceedings of Distributed Computing Systems Workshops (ICDCSW)*, pages 32 –41, june 2010.

[BSP+10] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *ACM Cloud Computing Security Workshop (CCSW'10)*, pages 93–102. ACM, 2010.

[BSSM08] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two–Dimensional Page Walks for Virtualized Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 26–35. ACM, 2008.

[Bun] Amazon EC2: ec2-bundle-vol. http://docs.amazonwebservices.com/AmazonEC2/dg/2006-10-01/CLTRG-ami-bundle-vol.html.

[BY97] Mihir Bellare and Bennet S. Yee. Forward integrity for secure audit logs. Technical report, Dept. of Computer Science & Engineering, University of California at San Diego, 1997.

[CA08] Rod Chapman and Peter Amey. *SPARK 95 - The SPADE Ada 95 Kernel*. Praxis High Integrity Systems, January 2008.

[CCS+11] Giovanni Cabiddu, Emanuele Cesena, Roberto Sassu, Davide Vernizzi, Gianluca Ramunno, and Antonio Lioy. The Trusted Platform Agent. *IEEE Software (Special issue on Software Protection)*, 8(2):35–41, 2011.

[CDE+09] Luigi Catuogno, Alexandra Dmitrienko, Konrad Eriksson, Dirk Kuhlmann, Gianluca Ramunno, Ahmad-Reza Sadeghi, Steffen Schulz, Matthias Schunter, Marcel Winandy, and Jing Zhan. Trusted virtual domains - design, implementation and lessons learned. In *International Conference on Trusted Systems 2009 (INTRUST 2009)*, December 2009.

[CDE+10] Serdar Cabuk, Chris I. Dalton, Konrad Eriksson, Dirk Kuhlmann, HariGovind V. Ramasamy, Gianluca Ramunno, Ahmad-Reza Sadeghi, Matthias Schunter, and Christian StÃ¼ble. Towards automated security policy enforcement in multi-tenant virtual data centers. *Journal of Computer Security*, 18(1):89–121, 2010.

[CGL+08] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R. K. Ports. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, Seattle, WA, USA, March 2008.

[CL99] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proc. of the 3rd Symp. on Operating Systems Design and Implementation*, pages 173–186, 1999.

[CLM+09] Luigi Catuogno, Hans Löhr, Mark Manulis, Ahmad-Reza Sadeghi, and Marcel Winandy. Transparent Mobile Storage Protection in Trusted Virtual Domains. In *23rd USENIX Large Installation Systems Administration Conference (LISA 2009)*, pages 159–172. USENIX Association, 2009.

[CLM+10] Luigi Catuogno, Hans Löhr, Mark Manulis, Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Trusted Virtual Domains: Color Your Network. *Datenschutz und Datensicherheit (DuD) 5/2010*, pages 289–294, 2010.

[Clo] The Cloud Market. EC2 Statistics. http://thecloudmarket.com/stats.

[Clo09a] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing, December 2009.

[Clo09b] Cloud Security Alliance (CSA). Security guidance for critical areas of focus in cloud computing v2.1, December 2009. https://cloudsecurityalliance.org/csaguide.pdf.

[Clo10a] Cloud Computing Use Case Discussion Group. Cloud computing use cases, 2010. http://cloudusecases.org/Cloud_Computing_Use_Cases_Whitepaper4_0.odt.

[Clo10b] Cloud Security Alliance (CSA). Top threats to cloud computing, March 2010. cloudsecurityalliance.org/topthreats/csathreats.v1.0.pdf.

[CLVV09] Miguel Correia, Lau Cheuk Lung, Paulo Verissimo, and Giuliana Santos Veronese. Minimal Byzantine fault tolerance: Algorithm and evaluation. Technical Report 09-15, DI-FCUL, 2009.

[CMSK07] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *Proc. of 21st Symp. on Operating Systems Principles*, pages 189–204, 2007.

[CMW+08] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. BFT: the time is now. In *Proc. of the 2nd Work. on Large-Scale Distributed Systems and Middleware*, pages 1–4, 2008.

[CNV04] Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proc. of the 23rd Intl. Symp. on Reliable Distributed Systems*, pages 174–183, 2004.

[CP03] Cheun N Chong and Zhonghong Peng. Secure audit logging with tamper-resistant hardware. In *18th IFIP International Information Security Conference (IFIPSEC), volume 250 of IFIP Conference Proceedings*, pages 73–84. Kluwer Academic Publishers, 2003.

[CPK10] Y. Chen, V. Paxson, and R. H. Katz. What's new about cloud computing security? Technical Report UCB/EECS-2010-5, Jan 2010.

[CWA+09] Allen Clement, Edmund Wong, Lorenzo Alvisi, Mike Dahlin, and Mirco Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proc. of the 6th Symp. on Networked Systems Design and Implementation*, pages 153–168, 2009.

[CYC+08a] David Challener, Kent Yoder, Ryan Catherman, David Safford, and Leendert Van Doorn. *A Practical Guide to Trusted Computing*. IBM Press, 2008.

[CYC08b] Ionut Constandache, Aydan Yumerefendi, and Jeff Chase. Secure control of portable images in a virtual computing utility. In *ACM workshop on Virtual machine security*, VMSec '08, pages 1–8. ACM, 2008.

[DKC+02] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

[DKP+11] T. Distler, R. Kapitza, I. Popov, H. P. Reiser, and W. Schröder-Preikschat. SPARE: Replicas on hold. In *Proc. of the 18th Network and Distributed System Security Symp.*, pages 407–420, 2011.

[DR06] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.1. RFC 4346, Internet Engineering Task Force, April 2006.

[DY81] Danny Dolev and Andrew C. Yao. On the security of public key protocols. Technical report, 1981.

[EBS] Amazon Elastic Block Store (EBS). http://aws.amazon.com/ebs/.

[EC2] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[EMC11] EMC. EMC, 2011. http://www.emc.com/products/category/storage.htm.

[ENI08] ENISA. Cloud computing risk assessment. http://www.enisa.europa.eu/act/rm/files/deliverables/cloud-computing-risk-assessment, 2008.

[ENI09] ENISA. Cloud Computing Risk Assessment. Technical report, ENISA, 2009.

[Eur09] European Network and Information Security Agency (ENISA). Cloud computing security risk assessment, May 2009. http://www.enisa.europa.eu/pages/02_03_news_2009_05_18_risk_survey.html.

[Fab07] Michael Fabian. Endpoint security: managing usb-based removable devices with the advent of portable applications. In *Proceedings of the 4th annual conference on Information security curriculum development*, InfoSecCD '07, pages 24:1–24:5, New York, NY, USA, 2007. ACM.

[FHL+96] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullmann, Godmar Back, and Stephen Clawson. Microkernel meet recursive virtual machines. In *USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[Gar09] S.L. Garfinkel. Automating disk forensic processing with SleuthKit, XML and Python. In *Workshop on Systematic Approaches to Digital Forensic Engineering*, pages 73–84. IEEE, 2009.

[GKQV10] Rashid Guerraoui, Nikola Kne$\mathcal{V}$zević, Vivien Quéma, and Marko Vukolić. The next 700 BFT protocols. In *Proc. of the EuroSys 2010 Conf.*, pages 363–376, 2010.

[Goo11] An Update on Android Market Security, March 3 2011. http://googlemobile.blogspot.com/2011/03/update-on-android-market-security.html.

[GPC+03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, 2003.

[GPS06] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking remote attestation to secure tunnel endpoints. In *Computer and Communications Security*, pages 21–24, 2006.

[GR03] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[GR05a] T. Garfinkel and M. Rosenblum. When virtual is harder than real: security challenges in virtual machine based computing environments. In *Workshop on Hot Topics in Operating Systems (HotOS'05)*, page 20. USENIX, 2005.

[GR05b] Tal Garfinkel and Mendel Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[GR09] Alexander Graf and Joerg Roedel. Nesting the Virtualized World. Linux Plumbers Conference, September 2009.

[Gro] Trusted Computing Group. Cloud computing and security - a natural match. http://www.trustedcomputinggroup.org/files/resource_files/1F4DEE3D-1A4B-B294-D0AD0742BA449E07/Cloud%20Computing%20and%20Security%20Whitepaper_July29.2010.pdf.

[GS03] S.L. Garfinkel and A. Shelat. Remembrance of data passed: a study of disk sanitization practices. *IEEE Security and Privacy*, 1(1):17 – 27, January 2003.

[Hat04] B. Hatch. SSH Host Key Protection, Oct 2004. http://www.symantec.com/connect/articles/ssh-host-key-protection.

[Hea06] C. Heath. *Symbian OS Platform Security*. John Wiley & Sons, 2006.

[HN08] Jay Heiser and Mark Nicolett. Assessing the security risks of cloud computing. http://www.gartner.com/DisplayDocument?id=685308, 2008.

[IAM] AWS Identity and Access Management (IAM) Documentation. http://aws.amazon.com/documentation/iam/.

[IBM] IBM Cooperation. IBM Smart Cloud. http://www-935.ibm.com/services/us/igs/cloud-development/#tab:details-security.

[IBM01] IBM. Autonomic computing, 2001. http://www.research.ibm.com/autonomic/.

[INR] INRIA. The coq proof assistant. http://coq.inria.fr/.

[JMG+02] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *In Proceedings of the USENIX 2002 Annual Technical Conference*, pages 275–288, 2002.

[JNL10] Keith Jeffery and Burkhard NeideckerLutz. The Future of Cloud Computing — Opportunities For European Cloud Computing Beyond 2010, 2010.

[JSZ03] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the selinux example policy. In *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, pages 5–5, Berkeley, CA, USA, 2003. USENIX Association.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative Byzantine fault tolerance. In *Proc. of the 21th Symp. on Operating Systems Principles*, pages 45–58, 2007.

[Kau07] Bernhard Kauer. OSLO: improving the security of Trusted Computing. In *Proceedings of the 16th USENIX Security Symposium*, pages 1–9, Berkeley, CA, USA, 2007. USENIX Association.

[KD04] Ramakrishna Kotla and Mike Dahlin. High throughput Byzantine fault tolerance. In *Proc. of the 2004 Intl. Conf. on Dependable Systems and Networks*, pages 575–584, 2004.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KGP+11] A. Kurmus, M. Gupta, R. Pletka, C. Cachin, and R. Haas. A comparison of secure multi-tenancy architectures for filesystem storage clouds. In *Proceedings of the 12th International Middleware Conference*, 2011. To appear.

[Kin] KindSoftware. Esc/java2 summary. http://kind.ucd.ie/products/opensource/ESCJava2/.

[KJ10] Manos Kapritsos and Flavio P. Junqueira. Scalable agreement: toward ordering as a service. In *Proceedings of the Sixth international conference on Hot topics in system dependability*, HotDep'10, pages 1–8, Berkeley, CA, USA, 2010. USENIX Association.

[KR09] Petr Kuznetsov and Rodrigo Rodrigues. BFTW3: Why? when? where? Work. on the theory and practice of Byzantine fault tolerance. *SIGACT News*, 40(4):82–86, 2009.

[KS05] S. Kent and K. Seo. Security Architecture for the Internet Protocol. RFC 4301, IETF, December 2005.

[KS09] Ulrich Kühn and Christian Stüble. User-friendly and secure tpm-based hard disk key management. In David Gawrock, Helmut Reimer, Ahmad-Reza Sadeghi, and Claire Vishik, editors, *Future of Trust in Computing*, pages 171–177. Vieweg+Teubner, 2009.

[Kun10] Vivek Kundra. State of Public Sector Cloud Computing, May 2010.

[LDLM09] Dave Levin, John R. Douceur, Jacob R. Lorch, and Thomas Moscibroda. Trinc: small trusted hardware for large distributed systems. In *Proc. of the 6th Symp. on Networked Systems Design and Implementation*, pages 1–14, 2009.

[LLC11] Amazon Web Services LLC. Amazon ec2 instance types. http://aws.amazon.com/ec2/instance-types/, 2011.

[LM04] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proc. of the 2004 Conf. on Dependable Systems and Networks*, pages 307–314, 2004.

[LS01] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*. USENIX Association, June 2001.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[MG09a] Peter Mell and Tim Grance. Effectively and Securely Using the Cloud Computing Paradigm, October 2009.

[MG09b] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2009. http://csrc.nist.gov/groups/SNS/cloud-computing/clouddefv15.doc.

[Mic10] Microsoft. Microsoft System Center IT Infrastructure Server Management Solutions, 2010. http://www.microsoft.com/systemcenter/.

[Mit08] Chris Mitchell, editor. *Trusted Computing*. The Institution of Engineering and Technology, second edition, 2008.

[MJM08] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building efficient replicated state machine for WANs. In *Proc. of the 8th Symp. on Operating Systems Design and Implementation*, pages 369–384, 2008.

[MLQ+10a] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *IEEE Symposium on Security and Privacy*, pages 143–158, 2010.

[MLQ+10b] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil D. Gligor, and Adrian Perrig. TrustVisor: Efficient TCB Reduction and Attestation. In *Proceedings of IEEE Symposium on Security and Privacy (Oakland 2010)*, May 2010.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. 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, 2008. ACM.

[MS10] D. Molnar and S. Schechter. Self hosting vs. cloud hosting: Accounting for the security impact of hosting in the cloud. In *Workshop on the Economics of Information Security (WEIS 2010)*, June 2010.

[MSMW03] Rich MacDonald, Sean Smith, John Marchesini, and Omen Wild. Bear: An open-source virtual secure coprocessor based on TCPA. Technical Report TR2003-471, Department of Computer Science, Dartmouth College, Hanover, NH, USA, 2003.

[MSW+04a] John Marchesini, Sean W. Smith, Omen Wild, Josh Stabiner, and Alex Barsamian. Open-source applications of tcpa hardware. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 294–303, Washington, DC, USA, 2004. IEEE Computer Society.

[MSW+04b] John Marchesini, Sean W. Smith, Omen Wild, Josh Stabiner, and Alex Barsamian. Open-source applications of TCPA hardware. In *20th Annual Computer Security Applications Conference*, pages 294–303, Washington, DC, USA, December 2004. ACM, IEEE Computer Society.

[MSWM03] John Marchesini, Sean W. Smith, Omen Wild, and Rich MacDonald. Experimenting with TCPA/TCG hardware, or: How I learned to stop worrying and love the bear. Technical Report TR2003-476, Department of Computer Science, Dartmouth College, 2003.

[MT09] Di Ma and Gene Tsudik. A new approach to secure logging. *Trans. Storage*, 5:2:1–2:21, March 2009.

[Mur08] J. Murty. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, 2008.

[NE08] EUROPEAN NETWORK and INFORMATION SECURITY AGENCY (ENISA). Secure usb flash drives, June 2008.

[NIS11a] NIST. Guidelines on security and privacy in public cloud computing. 2011. Special Publication 800-144.

[NIS11b] NIST. The NIST Definition of Cloud Computing (Draft). 2011. Special Publication 800-145 (Draft).

[OCJ08] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Exploiting Live Virtual Machine Migration. In *BlackHat DC Briefings*, Washington DC, February 2008.

[Ope10a] OpenSource. OpenNebula, 2010. http://www.opennebula.org/.

[Ope10b] OpenSource. OpenStack, 2010. http://www.openstack.org/.

[Ora10] Oracle. Solaris Cluster 3.2 Documentation, 2010. http://www.oracle.com/technetwork/documentation/solariscluster32-193000.html.

[Ora11a] Oracle. Oracle Real Application Clusters (RAC), 2011. http://www.oracle.com/technetwork/database/clustering/overview/index.html.

[Ora11b] Oracle. System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones, 2011. http://download.oracle.com/docs/cd/E1979701/8171592/index.html.

[Pea02] Siani Pearson. *Trusted Computing Platforms: TCPA Technology in Context*. Prentice Hall PTR, 2002.

[Per05] Colin Percival. Cache missing for fun and profit. http://www.daemonology.net/papers/htt.pdf, May 2005.

[PG74] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Commun. ACM*, 17:412–421, July 1974.

[PK00] O. Pirttikoski and Y. Kortesniemi. Local Key and Certificate Storage in JDK 1.3. *Proceedings of the NordSec2000, Reykjavk, Iceland*, 2000.

[PKM08] Simon Edward Parkin, Rouaa Yassin Kassab, and Aad P. A. Van Moorsel. The impact of unavailability on the effectiveness of enterprise information security technologies. In *International Service Availability Symposium*, pages 43–58, 2008.

[PM10] Wing-Chi Poon and Alyosius K. Moon. Bounding the Running Time of Interrupt and Exception Forwarding in Recursive Virtualization for the x86 Architecture. Technical Report TR-2010-003, Vmware Inc., Palo Alto, CA, USA, October 2010.

[PMP10] Bryan Parno, Jonathan M. McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *S&P '10: Proceedings of the IEEE Symposium on Security and Privacy*, pages 414–429, Oakland, CA, USA, 2010. IEEE Press.

[Pow92] D. Powell. Fault assumptions and assumption coverage. In *Proceedings of the 22nd IEEE International Symposium of Fault-Tolerant Computing*, July 1992.

[RC11] Francisco Rocha and Miguel Correia. 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 (DCDV, with DSN'11)*, June 2011.

[RI11] Chris Rohlf and Yan Ivnitskiy. Attacking clientside jit compilers. http://www.matasano.com/research/Attacking_Clientside_JIT_Compilers_Paper.pdf, 2011.

[RK07] H. P. Reiser and R. Kapitza. Hypervisor-based efficient proactive recovery. In *Proc. of the 26th Intl. Symp. on Reliable Distributed Systems*, pages 83–92, 2007.

[RTA+08] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: using semantic information to combat virtual machine image sprawl. In *SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'08)*, pages 111–120. ACM, 2008.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 199–212, New York, NY, USA, 2009. ACM.

[RY10] T. Ristenpart and S. Yilek. Randomness goes bad: Virtual machine reset vulnerabilities and hedging deployed cryptography. In *Network and Distributed Security Symposium (NDSS'10)*. ACM, 2010.

[S3] Amazon Simple Storage Service (S3). http://aws.amazon.com/s3/.

[SBC+10] Paulo Sousa, Alysson Neves Bessani, Miguel Correia, Nuno Ferreira Neves, and Paulo Verissimo. Highly available intrusion-tolerant services with proactive-reactive recovery. *IEEE Transactions on Parallel and Distributed Systems*, 21(4):452–465, 2010.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Survey*, 22(4):299–319, 1990.

[SDB] Amazon SimpleDB. http://aws.amazon.com/simpledb.

[SFKE08] Frederic Stumpf, Andreas Fuchs, Stefan Katzenbeisser, and Claudia Eckert. Improving the scalability of platform attestation. In *STC '08: Proceedings of the 3rd ACM workshop on Scalable Trusted Computing*, pages 1–10, New York, NY, USA, 2008. ACM.

[SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the 2009 conference on Hot topics in cloud computing*, HotCloud'09, pages 3–3, Berkeley, CA, USA, 2009. USENIX Association.

[SHJ+11] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jorg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All Your Clouds are Belong to us - Security Analysis of Cloud Management Interfaces. In *2011 ACM workshop on Cloud computing security workshop*. ACM, 2011.

[SK99] Bruce Schneier and John Kelsey. Secure audit logs to support computer forensics. *ACM Transactions on Information Systems*, 2:159–176, 1999.

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems*. ACM, 2010.

[SLQP07] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. *SIGOPS Oper. Syst. Rev.*, 41:335–350, October 2007.

[SRA+10] M. Satyanarayanan, W. Richter, G. Ammons, J. Harkes, and A. Goode. The case for content search of vm clouds. In *Computer Software and Applications Conference Workshops (COMPSACW'10)*, pages 382 –387. IEEE, 2010.

[SS75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer system. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[SS08] Mario Strasser and Heiko Stamer. A software-based Trusted Platform Module emulator. In *Proceedings of the 1st International Conference on Trust and Trustworthy Computing*, volume 4968 of *LNCS*, pages 33–47, Berlin, Germany, 2008. Springer.

[SU98] Morten Heine B. Sorensen and Pawel Urzyczyn. Lectures on the curry-howard isomorphism, 1998.

[SV10] N. Smart and F. Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. *Public Key Cryptography (PKC'10)*, pages 420–443, 2010.

[SW99] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks: The Intl. Journal of Computer and Telecommunications Networking*, 31(9):831–860, 1999.

[Swi09] S. Swidler. How to Keep Your AWS Credentials on an EC2 Instance Securely, August 2009. http://shlomoswidler.com/2009/08/how-to-keep-your-aws-credentials-on-ec2.html.

[SZ10] Christian Stüble and Anoosheh Zaerin. μTSS – A simplified trusted software stack. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, volume 6101 of *LNCS*, pages 124–140, Berlin, Germany, 2010. Springer.

[Sze] M. Szeredi. SSH filesystem. http://fuse.sourceforge.net/sshfs.html.

[SZJvD04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and implementation of a TCG-based Integrity Measurement Architecture. In *Proceedings of the 13th USENIX Security Symposium*, pages 223–238, Berkeley, CA, USA, 2004. USENIX Association.

[Tat06] E.I. Tatlı. Google Reveals Cryptographic Secrets. In *Kryptowochenende 2006 – Workshop über Kryptographie. Universität Mannheim*, page 33, 2006.

[TBEH04] Mads Tofte, Lars Birkedal, Martin Elsman, and Niels Hallenberg. A retrospective on region-based memory management. *Higher-Order and Symbolic Computation*, 17:245–265, 2004.

[TCG] Trusted Computing Group. https://www.trustedcomputinggroup.org.

[TJA10] H. Takabi, J. B.D. Joshi, and G.-J. Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security and Privacy*, 8:24–31, 2010.

[tro] TrouSerS for Windows. http://security.polito.it/tc/trouserswin.

[Tru07a] Trusted Computing Group. TCG Software Stack (TSS) Specification Version 1.2, Errata A. https://www.trustedcomputinggroup.org, 2007.

[Tru07b] Trusted Computing Group. TPM Main Specification, Version 1.2, Revision 103. https://www.trustedcomputinggroup.org, 2007.

[TWNH09] Ronald Toegl, Thomas Winkler, Mohammad Nauman, and Theodore Hong. Towards platform-independent Trusted Computing. In *STC '09: Proceedings of the 4th ACM workshop on Scalable Trusted Computing*, pages 61–66, New York, NY, USA, 2009. ACM.

[UNR+05] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, 2005.

[UT] University of Cambridge and Technische Universit"at M"unchen. Isabelle. http://isabelle.in.tum.de/.

[Var11] J. Varia. Architecting for the cloud: Best practices, January 2011. http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf.

[VCBL09a] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Highly-resilient services for critical infrastructures. In *Proc. of the Work. on Embedded Systems and Communications Security*, pages 1–5, 2009.

[VCBL09b] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, and Lau Cheuk Lung. Spin one's wheels? Byzantine fault tolerance with a spinning primary. In *Proc. of the 28th Intl. Symp. on Reliable Distributed Systems*, pages 135–144, 2009.

[VCNS09] Paulo Verissimo, Miguel Correia, Nuno Ferreira Neves, and Paulo Sousa. *Intrusion-Resilient Middleware Design and Validation*, pages 615–678. Emerald, 2009.

[VDJ10] Marten Van Dijk and Ari Juels. 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, 2010. USENIX Association.

[VMw10] VMware. VMware vCenter Server, 2010. http://www.vmware.com/products/vcenter-server/.

[VNC00] P. Verissimo, N. F. Neves, and M. Correia. The middleware architecture of MAFTIA: A blueprint. In *Proceedings of the Third IEEE Information Survivability Workshop*, October 2000.

[WAP08] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX 2008 Annual Technical Conference*, pages 321–334, Berkeley, CA, USA, 2008. USENIX Association.

[web] Memcached - a distributed memory object caching system. http://memcached.org/.

[Wik10a] Wikipedia. Network-Attached Storage (NAS), 2010. http://en.wikipedia.org/wiki/Network-attached_storage.

[Wik10b] Wikipedia. Storage Area Network (SAN), 2010. http://en.wikipedia.org/wiki/Storage_area_network.

[Woj08] Rafal Wojtczuk. Adventures with a certain Xen vulnerability (in the PVFB backend). http://invisiblethingslab.com/pub/xenfb-adventures-10.pdf, October 2008.

[WR09] Rafal Wojtczuk and Joanna Rutkowska. Attacking smm memory via intelÂ® cpu cache poisoning. Technical report, Invisible Things Lab, 2009.

[WSV+11] Timothy Wood, Rahul Singh, Arun Venkataramani, Prashant Shenoy, and Emmanuel Cecchet. ZZ and the art of practical BFT execution. In *Proc. of the EuroSys 2011 Conf.*, pages 123–138, 2011.

[WZA+09] J. Wei, X. Zhang, G. Ammons, V. Bala, and P. Ning. Managing security of virtual machine images in a cloud environment. In *ACM Cloud Computing Security Workshop (CCSW'09)*, pages 91–96. ACM, 2009.

[Xi04] Hongwei Xi. Applied type system. In *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer Berlin / Heidelberg, 2004.

[YBAG04] J. Yan, A. Blackwell, R. Anderson, and A. Grant. Password memorability and security: empirical results. *IEEE Security and Privacy*, 2(5):25 –31, 2004.

[YBDS08] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *GCE'08: Grid Computing Environments Workshop*, pages 1–10, Austin, TX, USA, 2008.

[YBS08] L. Youseff, M. Butrico, and D. Da Silva. Toward a unified ontology of cloud computing. In *Proceedings of Grid Computing Environments Workshop*, pages 1–10. IEEE, 2008.

[YDD+10] Muli B. Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[YL06] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Protocol Architecture. RFC 4251 (Proposed Standard), January 2006.

[YMV+03] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proc. of the 19th Symp. on Operating Systems Principles*, pages 253–267, 2003.

[YN09] A.A. Yavuz and Peng Ning. Baf: An efficient publicly verifiable secure audit logging scheme for distributed systems. In *Computer Security Applications Conference, 2009. ACSAC '09. Annual*, pages 219 –228, dec. 2009.

[Zei95] Stephen F. Zeigler. Comparing development costs for c and ada. 1995.

[ZNZ+10] W. Zhou, P. Ning, X. Zhang, G. Ammons, R. Wang, and V. Bala. Always up-to-date: scalable offline patching of vm images in a compute cloud. In *Annual Computer Security Applications Conference (ACSAC'10)*, pages 377–386. ACM, 2010.

List of Acronyms

Attestation Identity Key (AIK)

AVI Composite Fault Model (AVI)

Amazon Web Services (AWSEntry)

Certificate Authority (CA)

Central Processing Unit (CPU)

Core Root of Trust for Measurements (CRTM)

Denial Of Service (DoS)

Endorsement Key (EK)

European Network and Information Security Agency (ENISA)

Fully Homomorphic Encryption (FHE)

Garbled Circuits (GC)

Graphical User Interface (GUI)

Hardware Security Module (HSM)

Infrastructure as a Service (IaaS)

Identity and Access Management (IAM)

Identity Management (IdM)

Integrity Measurement Architecture (IMA)

Internet Protocol (IP)

Internet Protocol Security (IPsec)

Information Rights Management (IRM)

Local Area Network (LAN)

Network Access Control (NAC)

Order Preserving Encryption (OPE)

Platform as a Service (PaaS)

Platform Configuration Register (PCR)

Provable Data Posession (PDP)

Perfect Forward Secrecy (PFS)

Public Key Infrastructure (PKI)

Proof of Retrievability (POR)

Random-access memory (RAM)

Role-Based Access Control (RBAC)

Root of Trust for Measurements (RTM)

Software as a Service (SaaS)

Storage Root Key (SRK)

Trusted Computing (TC)

Trusted Computing Base (TCB)

Trusted Computing Group (TCG)

TCG Core Services (TCS)

TCG Device Driver Library (TDDL)

Transport Layer Security (TLS)

Trusted Management Channel (TMC)

TrustedObjects Manager (TOM)

Trusted Platform Module (TPM)

TCG Service Provider (TSP)

TCG Software Stack (TSS)

Trusted Virtual Domain (TVD)

Virtual Local Area Network (VLAN)

Virtual Machine (VM)

Virtual Machine Monitor (VMM)

Virtual Private Network (VPN)

Glossary

Access Control

Access control describes a method which enables an authority to control access to areas and resources in a computer-based information system. Access Control requires Authentication to identify subjects that request access to an object.

Amazon Web Services

A collection of cloud services offered by the company Amazon.com, launched in July 2006, allowing their customers to rent entire virtual infrastructures (including computing, storage, network, and maintenance services). It is one of the major providers of Cloud computing today and focuses primarily on IaaS services. The two best known and fundamental services are the Elastic Compute Cloud (ECtwo) computing service and the Simple Storage Service (Sthree) storage service.

Arbitrary Failure

Is the encompassing category of all failure modes: failures that can be omissive, assertive or both (see Failure Mode). It often used synonymously with Byzantine failure (see Byzantine Fault).

Arbitrary Failure Models

Failure modes that specify no qualitative or quantitative bounds on failures of the components. However, bounds on the number of components of the system which can fail are inevitable, e.g., "only less than one third of the components can fail". Arbitrary failure assumptions are usually costly to handle, both in terms of performance and algorithm complexity.

Attack

An *attack* is a malicious interaction with the system, performed with the objective of exploiting one or more vulnerabilities. Examples are port scans and hacker attempts to guess passwords.

Attestation

Attestation can be used to check the status of a computer system. Based on attestation the 1) authenticity of computer system and 2) the trustworthiness of the system's hardware and software can be determined.

Attestation Identity Key

When a user requires an Attestation Identity Key (AIK) the user wants its key to be certified by a CA. Therefor the user sends three credentials through a TPM: a public key credential, a platform credential, and a conformance credential.

Authenticated Boot

During the boot procedure of a Trusted Computing Platform, a Chain of Trust is build, starting with the Core Root of Trust for Measurements as anchor, to measure the platform state after the bootstrap procedure. The measurement, in form of a cryptographic hash, of the next component during the bootstrap, i.e. the next link in the chain, is taken before this component is executed. The measurements are stored inside the Platform Configuration Register of Trusted Platform Module. The chain include the BIOS, the bootloader, and the Operating System kernel. In contrast to Secure Boot, the boot procedure will not be aborted and a remote party must determine the trust it puts into the booted system based on the measurements taken and Remote Attestation. On an Integrity Measurement Architecture the chain is further extended into the userspace.

Authentication

Authentication is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the subject are true. This might involve confirming the identity of a person, program, or source etc. Identity can be proven by three factors:

- Knowledge factor – Something the subject knows (e.g. password)
- Inherence factor – Something the has (e.g. ID card)
- Ownershop factor – Something the user is (e.g. fingerprint)

Authenticity

The property of en entity being "genuine". For example, if a document or a network message identifies its author/sender, it is authentic if it was truly authored/sent by that entity. Authenticity is often tied to certain Authentication tests. When they have been passed successfully, an entity is called "authenticated".

Availability

Availability applies to assets. An asset is available if it is accessible and usable when needed by an authorized entity. In other words availability is the readiness of the system to provide a correct service. Attacks against availability are often designated DoS attacks.

AVI Composite Fault Model

The AVI composite fault model is a specialization of the fault → error → failure sequence [VNC00]. It describes the mechanism of intrusion in terms of three kinds of faults and the mechanism that can lead to failure: vulnerability + attack → intrusion → error → failure. The model gives insight into the mechanisms that can be used to tolerate these faults avoiding failure, i.e., on how to ensure the security properties.

Byzantine Fault

The term is used in two senses. Byzantine faults can be a particular kind of arbitrary faults studied in a classical paper by Lamport et al. [LSP82]. Today, the term is typically used as a synonym of arbitrary faults.

Certificate Authority

An entity that issues digital certificates which certify the ownership of a public key by the named subject of the certificate. This allows others to rely upon signatures or assertions made by the private key that corresponds to the public key. CAs are characteristic for Public Key Infrastructure.

Chain of Trust

It is build by iteratively verifying the trustworthiness of the next link in a chain starting a Root of Trust. The trust perimeter is automatically extended to the next trustworthy link, based on transitive trust relationships established by a successful verification. Authenticated Boot or Secure Boot are used to build a Chain of Trust at boot time of a system, but also certificate chains build a Chain of Trust.

Confidentiality

Confidentiality applies to information. Confidentiality is the fact that it is ensured that information is protected and that it is not made available or disclosed to unauthorized entities. In this context, entities include both individuals and processes. Disclosure of information is hard to detect so it is usually prevented/tolerated using cryptography.

Controlled Failure Models

Failure modes that specify qualitative and quantitative bounds on component failures. For example, the fault model can specify that there are only crash failures (e.g., crash of hosts) or omission failures (e.g., loss of packets in the network). This approach can be realistic since it represents well how common systems work under the presence of accidental faults only. However, it is very difficult to model the behavior of an attacker, so specifying bounds on component failures in the presence of malicious faults can create a problem of assumption coverage.

Core Root of Trust for Measurements

An immutable part of the system Firmware (i.e. BIOS) that acts as the Root of Trust for an Authenticated Boot or Secure Boot.

Correct Service

A system or a component provides a service that is *correct* if it performs the intended function. Conversely, a failure happens when a service offered by the system no longer behaves as intended.

Covert Channels

Not to be confused with *Covered Channels*. Covert Channels are channels, that are used to transport any kind of information between to entities that are not allowed to communicate. This is done by e.g. exploiting certain physical characteristics or 'abusing' meta data.

Denial Of Service

A denial-of-service attack (DoS attack) or distributed denial-of-service attack (DDoS attack) is an attempt to make a computer resource unavailable to its intended users. This is usually done by accessing the resource through normal means, but in an extensive amount.

Descretionary Access Control

Is a technique to restrict access to objects based on the identity of subjects. DAC refers to one category of access controls that subjects can transfer among each other, and MAC refers to a second category of access controls that imposes constraints upon the first.

Direct Anonymous Attestation

A cryptographic protocol, enabling the Remote Attestation of a Trusted Computing Platform whilst preserving the user's privacy.

Endorsement Key

The key used in the process for the issuance of AIK credentials and to establish a platform owner

Error

An *error* means that the system (or parts thereof) entered a state that deviates from the state that would've been required to provide a correct service. Such a condition isn't necessarily observable from a user of the system and may not even have any noticeable influence on the behaviour of the systems' services during its lifetime at all. However, if the error becomes visible at the system interface and affects its behaviour, a system failure occurred.

Error Processing

Techniques and mechanisms that performed with error detection and recovery (e.g., intrusion detection and countermeasures) or with error compensation (e.g., error masking using majority voting).

Failure

A behavior of a system that deviates from the intended function. The failure is a direct consequence of an Error, but does not necessarily happen immediately after the error occurred. If the error never influenced the subsequent system behaviour, it will also never come to a failure. Moreover, a failure does not impose any special kind of deviation of intended behaviour, like, e.g. a crash. Dependability aims at avoiding system failures.

Failure Mode

The way in which a component fails [Pow92]. Failures can be omissive (the component does not do an action when it was supposed to do) or assertive (the component does an action in a manner not specified). *Omissive failures* can be classified in timing failures (late or early interactions), omission failures (missing interactions), and crash failures (stop interacting). *Assertive failures* can be syntactic (incorrect format) or semantic (incorrect meaning).

Fault Forecasting

Techniques and mechanisms that involve the evaluation of the history of fault occurrence and activation in the system. This evaluation can be qualitative (identification of the faults, components where they occur, environmental causes) and quantitative (probability, how dependability attributes are affected).

Fault Prevention

Techniques and mechanisms that aim to impede the occurrence or introduction of faults in the system. Quality control techniques in the design of hardware and software are typical fault prevention examples.

Fault Removal

Techniques and mechanisms used during the system development. They involve verification of dependability properties (including validation of the specification) and the diagnosis and correction of problems. During the operational life, fault removal is made in the context of system maintenance.

Fault Tolerance

Techniques and mechanisms that intend to make a system continue to deliver a correct service despite the actual presence of faults. Fault tolerance involves both error processing and fault treatment.

Fault Treatment

Techniques and mechanisms that that aim to avoid that faults are reactivated. It encompasses fault diagnosis (identifying the cause) and fault isolation (exclusion of faulty components).

Fully Homomorphic Encryption

Fully Homomorphic Encryption is an encryption scheme, that supports operations in the cipher text space to be equivalent to arithmetic operations in plain text space. That means that for a FHE scheme ε the following holds: $\varepsilon(x) \boxplus \varepsilon(y) = \varepsilon(x + y)$ where $\boxplus$ represents an algebraic operation and $+$ its plaintext counterpart. Moreover, a FHE scheme provides two algebraic operations $(+, \cdot)$ so that operations form a ring structure. However, these concepts are still quite theoretical as their practical implementation remains far too complex and exhibits poor performance. They are related to Garbled Circuits.

Garbled Circuits

Garbled Circuits allow the computation of an algorithm at an untrusted party while preserving the confidentiality of the input to the algorithm, the computations performed thereon and its output. The performance of GCs is quite decent. However, the generation of a GC is a complex process and its input can only be used once. Then, a new GC has to be created to prevent replay-attacks. GCs are related to Fully Homomorphic Encryption.

Hardware Security Module

These modules are physical devices with a secure cryptoprocessor targeted at managing digital keys, accelerating cryptoprocesses in terms of digital signings and for providing strong authentication to access critical keys. HSMs protect high-value cryptographic keys trough (a) onboard secure generation, (b) onboard secure storage, (c) use of cryptographic and sensitive data material, (d) offloading application servers for complete asymmetric and symmetric cryptography. Such modules provide both logical and physical protection of these materials from non-authorized use and potential adversaries. E.g. HSM in a PKI environment: CA HSMs.

Hybrid Failure Models

Failure modes that combine controlled and arbitrary failure assumptions. The idea is to allocate different assumptions to different subsets of components of the system. With hybrid assumptions some parts of the system can be assumed to exhibit fail-controlled behavior, whilst the remainder of the system is still allowed to fail arbitrarily. This is advantageous in modular and/or distributed system architectures subjected to malicious faults. However, this approach is only feasible when the fault model is well substantiated, that is, when the behavior assumed for every single subset of the system can be modeled and/or enforced with high coverage. As a matter of fact, a system normally fails by its weakest link, and naive assumptions about a component's behavior can be an easy target to attackers.

Hypervisor

See Virtual Machine Monitor.

Identity and Access Management

Techniques and mechanisms that attempt to realize management of digital identities, authentication and authorization (e.g. User Management and Role Management) for Access Control, provisioning, etc. E.g., Cloud specific AWS Identity and Access Management (IAM) from Amazon.com as part of Amazon Web Services (AWSEntry).

Identity Management

See IAM.

Information Rights Management

A technology which protects sensitive information from unauthorized access.

Infrastructure as a Service

Provides virtual machines and other abstracted hardware and operating systems which may be controlled through a service API. Examples include Amazon EC2 and Simple Storage Service, Terremark Enterprise Cloud, Windows Live Skydrive and Rackspace Cloud. (European Network and Information Security Agency (ENISA), 2009)

Integrity

The absence of invalid system state alterations. The expression "system state" should be taken very generically. It may include data and code in a host, messages in the network, and hardware configuration. Invalid alteration can be due to accidental faults (e.g., electromagnetic noise corrupting a network packet) or malicious faults (e.g., a hacker corrupting the packet). The Integrity of information is preserved when the accuracy and completeness is of it is left untouched and was not tampered with. This term is related to Authenticity.

Integrity Measurement Architecture

An enhancement of Linux by a Trusted Platform Module based Linux Security Module that is currently being developed by IBM research in close collaboration with various product groups and the IBM Linux Technology Center. Its main goal is to generate verifiable representative information about the software stack running on a Linux system.

This information can be used by remote parties to determine the integrity of the execution environment.

Internet Protocol Security

A protocol suite for securing IP communications by authenticating and encrypting each IP packet of a communication session.

Intrusion

The event of an attack managing to exploit a vulnerability is called an *intrusion*, a malicious operational fault. An intrusion happens, for example, when an attacker managed to guess a privileged account password and then penetrates the system. This intrusion causes an *error* (unauthorized person gains privileges) which can later cause the system *failure* (e.g., the system delivers modified web pages).

Local Area Network

A computer network that connects devices in a limited geographical area.

Maintainability

The property of a system being able to undergo repair and reconfiguration.

Mandatory Access Control

Mandatory Access Control is a technique that constrains subject's access to objects. It is often seen as the opposite of Descretionary Access Control. DAC refers to one category of access controls that subjects can transfer among each other, and MAC refers to a second category of access controls that imposes constraints upon the first.

MapReduce

It is a software framework introduced by Google in 2004 to support distributed computing on large data sets on clusters of computers. The framework is inspired by the map and reduce functions commonly used in functional programming, although their purpose in the MapReduce framework is not the same as their original forms.

Network Access Control

Techniques and mechanisms that attempt to realize Access Control at the network layer, especially to unify endpoint security technology (such as anti-virus tools), user or system Authentication and network security enforcement. An open architecture for Network Access Control is Trusted Network Connect (TNC).

Nonrepudiability

The property of the author/sender of a piece of data not being able to deny that he authored/sent it.

Order Preserving Encryption

Order Preserving Encryption is an encryption scheme, that allows the data (despite being encrypted) to be partially and totally ordered, i.e. standard numeric relations ($<, >, =$) to be applied.

Owner

In the context of ISO 27001 and ISO 27002, an owner is a person or entity that has been given formal responsibility for the security of an asset or asset category. It does not mean that the asset belongs to the owner in a legal sense. Asset owners are only formally responsible.

Perfect Forward Secrecy

The property that ensures that a session key derived from a set of long-term public and private keys will not be compromised if one of the private keys is compromised in the future. It is an optional feature in IPsec.

Platform as a Service

Allows customers to develop new applications using APIs deployed and configurable remotely. The platforms offered include development tools, configuration management, and deployment platforms. Examples are Microsoft Azure, Force and Google App engine. (ENISA, 2009)

Platform Configuration Register

A 160-bit shielded storage location inside the Trusted Platform Module, of which an TPM in the current version 1.2 contains exactly 24, used to store the measurements taken, e.g., during Authenticated Boot.

Policy

A policy statement defines a general commitment, direction, or intention. An information security policy statement expresses management's commitment to the implementation, maintenance, and improvement of its information security management system.

Process

Is the loaded instance of a Program in computer's memory. It is usually designed to (but not necessarily) operate on input data and to produce output data or side effects. Only a process is a live system, the program is just its (offline) definition. When it is loaded, it becomes a process.

Program

A (computer) program is a piece of software that consists of machine instructions and a fixed set of data and is designed to extend the capabilities of the computer hardware. In contrast to a program, a Process is the loaded instance in computer's memory of a program which operates to do its intended purpose. Hence, a program might have several process instances.

Proof of Retrievability

A Proof of Retrievability is a proof that a third party issues to the challenger in order to guarantee that it possesses certain data. The data is not private, but usually belongs (and is known) to the challenger. The purpose of the proof is, that the proof is much shorter than sending the actual data in question. Please note, that a cryptographic hash is NOT a POR as the hash value could simply be stored without possessing the data. Therefore, a Nonce and HMAC is usually used. See also Provable Data Posession.

Provable Data Posession

Is the same as Proof of Retrievability.

Public Key Infrastructure

A set of software, hardware, people, policies, and procedures needed to create, manage, store, distribute, use, and revoke digital certificates. A CA issues those digital certificates, see Certificate Authority.

Reliability

The property of the service delivered being correct.

Remote Attestation

The verification of the trustworthiness of a computer system by a remote verifier, i.e. that the checked Trusted Computing Platform is in known and trusted state.

Role-Based Access Control

An approach to restricting system access to authorized users, based on grouping permissions and assigning them to roles. Subjects (users, automated agents/systems) can exercise a permission only if they have been assigned to one or multiple role(s). A subject's active role must be authorized for the subject. A subject can execute a permission only if the permission is authorized for the subject's active role.

Root of Trust

A component that must always behave in the expected manner, i.e. it has to be axiomatically trusted, because its misbehavior cannot be detected. It works as a trust anchor in the Chain of Trust, from which trust to other components is assured based on the (deemed) trustworthiness of the Root Of Trust.

Root of Trust for Measurements

A computing engine of the Trusted Computing Platform capable of making inherently reliable integrity measurements.

Safety

The absence of catastrophic consequences of a failure of the system to its users or environment.

Secure Boot

Identical to Authenticated Boot with the only difference, that the measurements taken during the bootstrap are compared to a trusted reference value and in case that these values differ the boot procedure is aborted. Thus, in contrast to Authenticated Boot, Secure Boot ensures that a Trusted Computing Platform can not be booted into an untrusted state.

Secure Function Evaluation

Is a protocol that allows a multi-party computation with n users and their respective inputs $x_1, x_2, \ldots, x_n$. After the protocol run, every user knows the outcome of the function $F(x_1, x_2, \ldots, x_n)$ but the input of the others remain unknown.

Security Hazard

Security hazards are normally classified in three broad categories –vulnerability, attack and intrusion– which are related to the notions of fault-error-failure through the AVI Composite Fault Model (AVI) composite fault model, developed in the MAFTIA project [VNC00].

Security Policy

The concepts of error and failure are related to the notion of *security policy*. The security policy of a system is a set of specific security requirements imposed on a system. Examples of items in a security policy are the minimum length of passwords and the machines that can be remotely accessed by certain user groups. The notion of security policy can be refined in terms of security goals and security rules. *Security goals* are high-level statements of the security properties that the system must guarantee. The violation of a security goal is a failure of the system. *Security rules* are lower level statements of constrains that the system should satisfy so that the security goals are not violated, therefore the rules are usually stronger than the goals. The violation of a security rule is an error, which can directly or indirectly lead to the failure of the system.

Simple Storage Service

A central service of AWSEntry, providing very high availability, key-value based online storage. In particular, the provided storage is designed for “BLOBs” (Binary Large Objects) and can be accessed via well-established interface protocols like SOAP or REST.

Software as a Service

Software offered by a third party provider, available on demand, usually via the Internet configurable remotely. Examples include online word processing and spreadsheet tools, CRM services and web content delivery services (Salesforce CRM, Google Docs, etc). (ENISA, 2009)

Storage Root Key

The root key embedded in the Trusted Platform Module, used to wrap others TPM keys.

TCG Core Services

The layer of the TCG Software Stack which provides a common set of services per platform for all services providers.

TCG Device Driver Library

This module provides the user mode interface for the TCS to communicate with the TPM Device Driver.

TCG Service Provider

The module of the TCG Software Stack which provides TCG services to applications.

TCG Software Stack

The TCG specifications of the software designated to communicate with the Trusted Platform Module device.

Third party

In the context of a specific issue, a third party is any person or body that is recognized as independent of the people directly involved with the issue in question.

Threat

A threat is the existence, but not the occurence of a potentially negative event. When a threat turns into an actual event, it may cause negative impact.

Token

A token, also referred to as *security token*, is a special, tamper-resistant hardware component. It is, e.g., used to hold secrets for Authentication in different scenarios, such as online banking. Advanced security tokens further provide a cryptographic co-processor and a trusted execution environment.

Transport Layer Security

A cryptographic protocol that is used for establishing a secure connection between a client and a server. TLS and its predecessor, Secure Sockets Layer (SSL), encrypt the segments of network connections above the Transport Layer (OSI Layer 4). TLS is using asymmetric cryptography for privacy and a keyed message authentication code for reliability. See RFC 5246 for details.

Trust

The expectation that a device will behave in an expected manner for a specific purpose.

Trusted Computing

The interaction between a hardware security anchor and upper layer protocols and applications. (Definition of the TCG) See also Trusted Computing Base.

Trusted Computing Base

The hardware and software components that are critical for the security of a computer system. Vulnerabilities or errors in the Trusted Computing Base jeopardize the security of the entire system.

Trusted Computing Group

An initiative started by AMD, Hewlett-Packard, IBM, Intel, and Microsoft to implement Trusted Computing and successor to the Trusted Computing Platform Alliance (TCPA). The TCG's original goal was the development of a Trusted Platform Module (TPM). The TCG also realised the Trusted Network Connect (TNC) protocol specification.

Trusted Computing Platform

A computing platform that can be trusted to report its properties. ”Trusted” is used in the sense of Trust. Components outside of the Trusted Computing Base are not critical. Their Trust is usually established by means that are incorporated inside the Trusted Computing Base.

Trusted Management Channel

Allows to securely connect the TOM with TrustedServers to set-up, start and stop VM instances, and to load configuration and policies. It is part of the overall security concept of a Trusted Infrastructure and also could be used to interconnect TOMs. It provides an confidential, integer, and authentic management channel between management console and server using TPM based remote attestation. Its primary goal is to provide a communication channel between a client and a server that provides all required communication features while satisfying certain security aspects.

Trusted Network Connect

Trusted Network Connect is an open architecture and a growing set of standards for endpoint integrity, especially for Network Access Control (NAC), published by the Trusted Network Connect Work Group (TNC-WG) of the Trusted Computing Group (TCG).

Trusted Platform Module

The Trusted Platform Module is an opt-in hardware device, usually mounted on the motherboard of a computer platform. It is defined by the Trusted Computing Group such that it implements security design goals like the secure reporting of the environment, the provision of secure storage, or the attestation of the platform status to remote parties (Remote Attestation). For that reason, the TPM is equipped with a number of essential components. To name the major components: a) an execution engine for the specified commands; b) a True Random Number Generator; c) Non-Volatile Memory; d) a cryptographic co-processor; e) Platform Configuration Registers (Platform Configuration Register); and f) a small number of monotonic counters.

Trusted Virtual Domain

A Trusted Virtual Domain is a coalition of virtual and/or physical machines that can trust each other based on a security policy that is uniformly enforced independently of the boundaries of physical computing resources. It leverages the combination of Trusted Computing and virtualization techniques in order to provide confinement boundaries for an isolated execution environment - a domain - hosted by several physical platforms.
Source: Löhr et al. - Trusted Privacy Domains – Challenges for Trusted Computing in Privacy-Protecting Information Sharing.

TrustedObjects Manager

The central management component of a Trusted Infrastructure. The TOM manages the physical infrastructure including networks, services and appliances (physical platforms). Since appliances remotely enforce a subset of the overall security policy, a permanent trusted channel between the TOM and dedicated appliances is used for client authentication, to check their software configuration using attestation, and to upload policy changes and software updates.

TrustedServer

The central security platform in a Trusted Infrastructure to run the VM instances (also called compartments) on. It is based on the TURAYA ™SecurityKernel and provides several built in security services. In order to provide strict isolation of compartments, operating systems and data compartments are linked to Trusted Virtual Domains (TVDs).

Domain specific transparent encryption is applied to prohibit information flows between TVDs.

Virtual Local Area Network

A group of hosts with a common set of requirements that communicate as if they were attached to the same broadcast domain, regardless of their physical location. It has the same attributes as a physical LAN, but allows end stations to be grouped together even if they are not located on the same network switch.

Virtual Machine

A VM is a software implementation of a machine or a computer that behaves like a physical machine from the operating systems perspective. Virtual Machines need the presence of a software layer (VMM) in order to access the multiplexed physical hardware.

Virtual Machine Monitor

The virtual machine monitor (also called Hypervisor) generates and manages virtual machines (VM). Hardware resources are shared to allow execution of multiple operating systems on one host.

Vulnerability

A vulnerability is a weakness in an asset or group of assets. An asset's weakness could allow it to be exploited and harmed by one or more threats. A vulnerability is the potential to be exploited, not the actual exploitation. It the chain 'Fault, Error, Failure' it is part of the Fault. A vulnerability can be a design fault (created during the system development) or an operational fault (introduced during system execution). Typical examples are software bugs (e.g. a missing array boundary test, which can be exploited to gain full control of the system), or a privileged account with a easily guessable password. These faults are usually accidental but can also be malicious, like in the case of a hidden backdoor to circumvent security measures.