D2.3.2
Components and Architecture of Security Configuration and Privacy Management

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 / D2.3.2 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.3
Due date:	September 2012 – M24
Actual submission date:	28th September, 2012

Responsible organisation:	IBM
Editor:	Sören Bleikertz
Dissemination level:	Public
Revision:	1.0

Abstract:	This report presents components and architectures for distributed security management, management of large-scale and complex cloud systems, and establishment of trust in cloud services.
Keywords:	Security management, isolation, models and languages, verification, quota enforcement, public-key infrastructures, trust anchors.

Editor

Sören Bleikertz (IBM)

Contributors

Sören Bleikertz, Thomas Groß (IBM)

Alexander Bürger, Michael Gröne, Norbert Schirmer (SRX)

Imad M. Abbadi, Andrew Martin, Anbang Ruan (OXFD)

Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia (POL)

Johannes Behl, Rüdiger Kapitza, Klaus Stengel (TUBS)

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.

Executive Summary

Cloud computing today is shaped by the involvement of a large number of entities with diverse trust relationships and by large-scale, complex systems. Security management, in particular in a distributed way, is essential for achieving a trusted cloud service that fulfills the requirements of high availability, fault tolerance, and scalability.

In this deliverable, we consolidate and analyze the requirements for distributed security management, such as high-availability and federation among multiple entities in clouds, and requirements for the adaption of security management in large-scale cloud infrastructures. We propose a set of components and architectures that tackle the challenges of distributed security management, managing large-scale and complex systems, and establishing trust among different entities in cloud services. We can categorize these components and architectures into the following high-level topics:

1. Security properties formalization and automated analysis
2. Distributed security management
3. Integrity assurance and local trust management

Security properties formalization and automated analysis Cloud computing and virtualized infrastructures are often accompanied by complex configurations and topologies. Dynamic scaling, rapid virtual machine deployment, and open multi-tenant architectures create an environment, in which local misconfiguration can create subtle security risks for the entire infrastructure. This situation calls for automated deployment as well as analysis mechanisms, which in turn require a cloud assurance policy language to express security goals for such environments. Where possible, configuration changes should be statically checked against the policy prior to implementation on the infrastructure.

We study security requirements of virtualized infrastructures and propose a practical tool-independent policy language for security assurance. Our policy proposal has a formal foundation, and still allows for efficient specification of a variety of security goals, such as isolation. In addition, we offer language provisions to compare a desired state against an actual state, discovered in the configuration, and thus allow for a differential analysis. The language is well-suited for automated deduction, be it by model checking or theorem proving.

Furthermore, we aim to introduce the *Trusted Virtual Domain* model in the Cloud architecture, in order to provide customers with a protected environment for their Virtual Machines (*VMs*) where the containment and trust properties are guaranteed by the infrastructure and, at the same time, to address the need for a strong isolation between tenants.

Distributed security management We approach the security and management concerns of Cloud providers as well as users that stem from the dynamic nature of clouds. For example how can Cloud providers assure users that: (a.) dependent applications running on different VMs are hosted within physical proximity (for performance reasons); (b.) mutually exclusive VMs are not hosted at the same physical server (e.g. for availability and security reasons); and (c.) when

migrating VMs the new allocated physical servers satisfy users' application requirements and security and privacy criteria. We propose a framework which at this foundation stage focuses on providing secure environment for the management of Clouds' virtual layer. It also helps in establishing trust in Cloud's operational management.

Furthermore, we discuss the logical separation of security and cloud management, while providing the necessary and required integration. We propose an architecture that integrates Public-Key-Infrastructure (PKI) management with OpenStack.

Finally, we present *DQMP*, a decentralized, fault-tolerant, and scalable quota-enforcement protocol. It allows, for instance, customers to buy a fixed amount of resources (e. g., CPU cycles) that can be used flexibly within the cloud, and cloud providers to control resource usage of customers on a cloud-wide scale.

Integrity assurance and local trust management Managing the allocation of Clouds virtual machines at physical resources is a key requirement for the success of Clouds. Current implementations of Cloud schedulers do not consider the entire Cloud infrastructure neither they consider the overall user and infrastructure properties. This results in major security, privacy and resilience concerns. We propose a novel Cloud scheduler which considers both users' requirements and infrastructure properties. We focus on assuring users that their virtual resources are hosted using physical resources that match their properties without getting users involved into understanding the details of the Cloud infrastructure.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 TClouds — Trustworthy Clouds

TClouds aims to develop *trustworthy* Internet-scale cloud services, providing computing, network, and storage resources over the Internet. Existing cloud computing services are generally not trusted for running *critical infrastructure*, which may range from business-critical tasks of large companies to mission-critical tasks for the society as a whole. The latter includes water, electricity, fuel, and food supply chains. TClouds focuses on power grids and electricity management and on patient-centric health-care systems as its main applications.

The TClouds project identifies and addresses legal implications and business opportunities of using infrastructure clouds, assesses security, privacy, and resilience aspects of cloud computing and contributes to building a regulatory framework enabling resilient and privacy-enhanced cloud infrastructure.

The main body of work in TClouds defines an architecture and prototype systems for securing infrastructure clouds through security enhancements that can harden commodity infrastructure clouds and by assessing the resilience, privacy, and security extensions of existing clouds.

Furthermore, TClouds provides resilient middleware for adaptive security using a cloud-of-clouds, which is not dependent on any single cloud provider. This feature of the TClouds platform will provide tolerance and adaptability to mitigate security incidents and unstable operating conditions for a range of applications running on a clouds-of-clouds.

1.2 Activity 2 — Trustworthy Internet-scale Computing Platform

Activity 2 (A2) carries out research and builds the actual TClouds platform, which delivers trustworthy resilient cloud-computing services. The TClouds platform contains trustworthy cloud components that operate inside the infrastructure of a cloud provider; this goal is specifically addressed by Workpackage 2.1 (WP2.1). The purpose of the components developed for the infrastructure is to achieve higher security and better resilience than current cloud computing services may provide.

The TClouds platform also links cloud services from multiple providers together, specifically in WP2.2, in order to realize a comprehensive service that is more resilient and gains higher security than what can ever be achieved by consuming the service of an individual cloud provider alone. The approach involves simultaneous access to resources of multiple commodity clouds, introduction of resilient cloud service mediators that act as added-value cloud providers, and client-side strategies to construct a resilient service from such a cloud-of-clouds.

WP2.3 introduces the definition of languages and models for the formalization of user- and

application-level security requirements, involves the development of management operations for security-critical components, such as "trust anchors" based on trusted computing technology (e.g., TPM hardware), and it exploits automated analysis of deployed cloud infrastructures with respect to high-level security requirements.

Furthermore, A2 will provide an integrated prototype implementation of the trustworthy cloud architecture that forms the basis for the application scenarios of Activity 3. Formulation and development of an integrated platform is the subject of WP2.4.

These generic objectives of A2 can be broken down to technical requirements and designs for trustworthy cloud-computing components (e.g., virtual machines, storage components, network services) and to novel security and resilience mechanisms and protocols, which realize trustworthy and privacy-aware cloud-of-clouds services. They are described in the deliverables of WP2.1–WP2.3, and WP2.4 describes the implementation of an integrated platform.

1.3 Workpackage 2.3 — Cross-layer Security and Privacy Management

The overall objective of WP2.3 is to provide mechanisms to manage the privacy-enhanced resilience of the TClouds platform. The work package has three phases that span the three years of the project. The goal during the first project year was to collect component requirements for management operations and to explore the interaction between the various technologies and the demonstration in WP2.4. Furthermore, several concepts and systems for selected management tasks have been developed.

In the second year, the requirements for large-scale and distributed security management have been consolidated. The components and the architecture have been developed further and partially finalized. These components are mostly documented in the current deliverable, and they show how the security objectives are can be implemented and managed on all different layers concerned by the TClouds platform.

1.4 Deliverable 2.3.2 — Components and Architecture of Security Configuration and Privacy Management

Overview. This deliverable consolidates and analyzes the requirements for distributed security management, such as high-availability and federation among multiple entities in clouds, and requirements for the adaption of security management in large-scale cloud infrastructures. Furthermore, we present a set of components and architectures that tackle the challenges of distributed security management, managing large-scale and complex systems, and establishing trust among different entities in cloud services. These components and architectures can be categorized into the following high-level topics:

1. Security properties formalization and automated analysis
2. Distributed security management
3. Integrity assurance and local trust management

Topic 1 deals with the management of complex cloud infrastructures and establishes languages for the specification of security policies, models for expressing security domains, and automated tools for analyzing cloud infrastructures with respect to such security policies.

In Topic 2, this deliverable presents components and architectures for empowering cloud users with more management capabilities and security assurance, for logically separating security management and cloud management, and for enforcing resource quotas in a distributed, scalable and fault-tolerant way. Thereby fulfilling the consolidated requirements for federated and highly available security management.

Finally, Topic 3 incorporates trust anchors, such as Trusted Platform Modules, in the management tasks for cloud infrastructure. In this deliverable, we present an architecture for a new workload scheduler that matches the security requirements of the cloud users with the attested security properties of the cloud infrastructure resources.

Structure. The deliverable is structured in the following way: Chapter 2 presents the consolidated requirements for distributed security management and for the adaption in large-scale cloud infrastructures. In Chapter 3, we introduce new languages and models for the expression of security policies and domains, and automated analysis of such policies is presented in Chapter 4. These chapters partially appeared also as [BG11, BGM11]. Architectures and components for distributed security management are aggregated in Chapter 5. This chapter partially appeared also as [AAM11, BDK12]. Finally, Chapter 6 presents an architecture for the integration of trust anchors in management tasks.

Deviation from Workplan. This deliverable conforms to the DoW/Annex I, Version 2.

Target Audience. This deliverable aims at researchers and developers of security and management systems for cloud-computing platforms. The deliverable assumes graduate-level background knowledge in computer science technology, specifically, in virtual-machine technology, operating system concepts, security policy and models, and formal languages.

Relation to Other Deliverables. Figure 1.1 illustrates WP2.3 and its relations to other workpackages according to the DoW/Annex I (specifically, this figure reflects the structure after the change of WP2.3 made for Annex I, Version 2).

The present deliverable, D2.3.2, relates many of the technologies addressed by WP2.3 to the trustworthy cloud infrastructure (WP2.1), specifically for managing security in a trustworthy manner. The management of secure virtual layers and trusted virtual domains as described in Chapters 3 and 5, as well as the trust anchors of Chapter 6, are key technologies in this respect.

D2.3.2 also provides mechanisms for assessing the security of the cloud-of-clouds middleware (WP2.2), specifically through the formal assurance languages and automated validation methods of Chapters 3 and 4, respectively. Selected components are developed as proof-of-concept prototypes and integrated into the TClouds platform of WP2.4. They are described in the corresponding deliverable D2.4.2.

D2.3.2 contributes the components and the architecture for cloud-computing security management to the development of the two application scenarios in WP3.1 and WP3.2. Furthermore, the components are validated in the context of WP3.3 (“Validation and Evaluation of the TClouds Platform”, not shown in the figure).

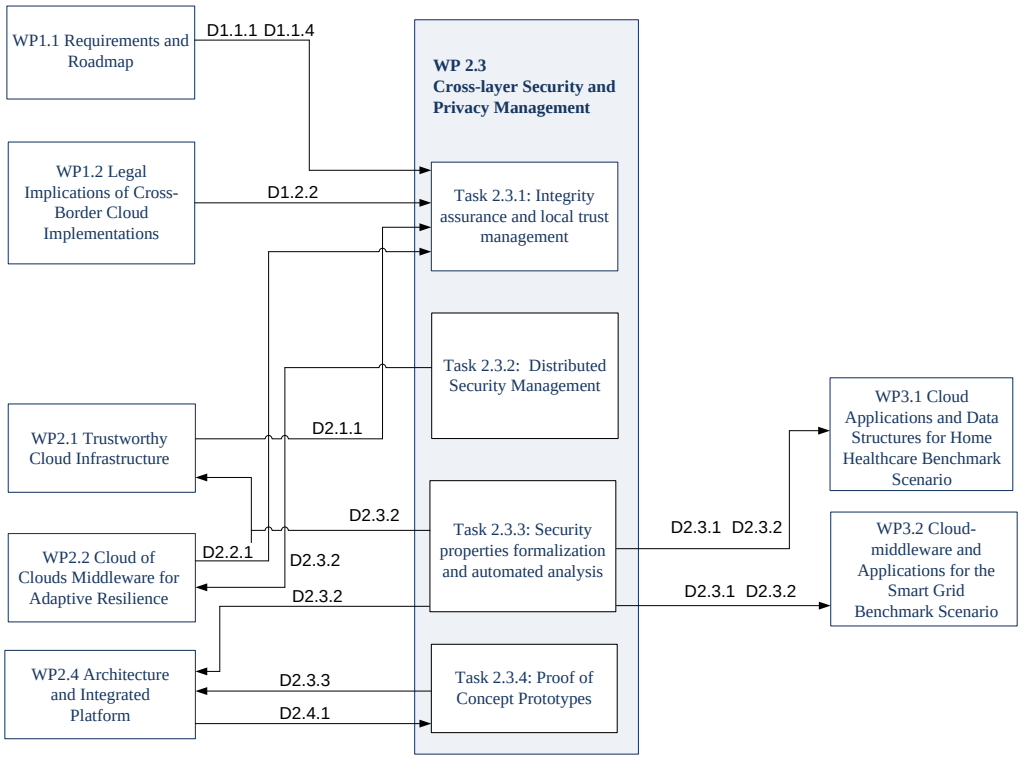


Figure 1.1: Graphical structure of WP2.3 and relations to other workpackages.

Chapter 2

Requirements and Analysis of Security Management

Chapter Authors:
Alexander Bürger, Michael Gröne, Norbert Schirmer (SRX),
Sven Bugiel, Stefan Nürnberger (TUDA)

In this chapter, we consolidate and analyze the requirements for the adaption of security management in large-scale cloud infrastructures and requirements of federation in the management of TrustedInfrastructure clouds.

2.1 Adaption for Large-Scale Infrastructures

2.1.1 Introduction

The cloud is composed of many concepts, like virtualization, distributed storage, data bases and so on. Some of those concepts may even have existed in a similar way in the pre-cloud era, but their combination forms the cloud. Many enhancements for these basic concepts exist in terms of security, trustworthiness, reliability or ease of use. Unfortunately, many of them that do work in the scenario of that single concept cease to work when it comes to their large-scale application, that is the dynamicity and flexibility of thousands of these instances working together. The reasons are manifold and typically stem from assumptions that are no longer true in the cloud, i.e. physical presence or a single machine. The cloud is designed to be as flexible as possible in order to fulfill the requirements for tailored resources that are billed on a pay-as-you-go model. For this to work every concept and component has to be flexible because together they form the infrastructure and services that the cloud offers. However, and this tight connection is limited by its weakest link in the chain of inter-operating components. We will look at exemplary security enhancements for the cloud that are not yet deployed because their large-scale adaption is currently infeasible.

2.1.2 Background

During the analysis conducted here, we assume an arbitrary component that already exists as a small-scale prototype and adaptation to a large-scale component is planned. For the sake of generality we do not assume a specific component but the following gives some examples in order to support imagination of such a component. The component is further assumed to work with respect to its functional requirements. However, its design might have to be adapted to

work in large-scale (i.e., the cloud) in order to deliver the same functionality in a distributed, large-scale infrastructure.

2.1.2.1 Mode of Operation

This component can operate as a **central service** for the whole cloud or as a **local service** for individual physical or virtual machines. Which mode of operation is applicable highly depends on the component itself.

Central Service An example of a central service is a service that needs to execute in an atomic fashion, e.g. counting a variable or keeping track of used cloud services. Such a service could be the cloud scheduler which is responsible for distributing load across the cloud infrastructure and hence needs to have a complete overview of the cloud.

Local Service An example of a local service or component is a data base that is only attached to one VM. It does not need to be central and even benefits from better performance due to good locality to the running VM.

2.1.2.2 Example Components

Logging Logs of an operating system are designed to gather information about events that happened in the past and might reveal an insight into why something happened (e.g. misconfiguration, attack traces etc.). However, as there is no physical access to a machine in the cloud, accessing these logs from outside the VM is not possible. Consequently, logs can no longer be used to identify misconfiguration errors that prevent a machine from booting. Moreover, consolidation of several logs in one file needs coordination in order to leave the logs in a consistent state[1]. The multi-tenancy of clouds makes it also necessary to authenticate log messages which in turn require a trust anchor for writing authenticated messages to the log.

Trusted Computing A key feature of Trusted Computing is attesting a hardware and/or software configuration, e.g. to entrust the machine with secrets. Trusted Computing completely builds upon the fact that a machines integrity is vouched for by the Trusted Platform Module (TPM), a chip local to every physical computer. Cloud computing on the other hand virtualizes the work load and the physical machines where the VMs run on are selected by the cloud providers scheduler in a way opaque to the customer. This black box behaviour is diametrically opposed to the identification requirements of Trusted Computing.

Transactional Memory & Data bases Transactional memory or transactional data bases ensure that operations happen in an atomic manner, i.e. the operation either succeeded and is visible to all subsequent operations or it failed completely and did not leave anything in an inconsistent state. This behaviour needs a lot of communication overhead due to coordination and hence imposes a significant slowdown in a distributed system like a cloud.

[1] Readers-Writers Problem http://en.wikipedia.org/wiki/Readers-writers_problem

Random Numbers One of the key benefits of the cloud is the ability to clone machines as they are no longer physical but virtual. This enables enormous scalability effects that are available at the touch of a button since a service can be multiplied by literally thousands of machines instantaneously. As all machines are derived from a single instance in this scenario, the complete state of all inherited clones is identical to begin with because they share the same past. Their future deviation solely varies with their input. This can lead to unwanted and potentially unsafe behaviour when it comes to random number generators. When all instances are cloned from a single source they are very likely to produce the same series of random numbers as they machine they were cloned from.

2.1.3 Cloud Provider Requirements

For the solutions to be actually adapted to a large-scale cloud, they have to climb certain obstacles. First of all, cloud providers want to deliver a satisfying customer experience, which means that new components need to be thoroughly tested before they can be used by potentially millions of customers analogously to software development and testing life-cycles. This process usually starts as a proof-of-concept and ripens until a satisfactory level of readiness for large-scale deployment has been reached.

Generally, new concepts are only adapted to the cloud, when each of the following criteria is met:

2.1.3.1 No negative impact

A newly introduced component must not affect the existing cloud in a negative way, compared to status quo. Obvious surfaces of negative impact are:

Security The introduction of a new component could tear a hole in the security system already in place at the cloud infrastructure.

Performance The introduction of a new component could slow down the throughput of the network, could congest the computational power of the virtual machines or even the cloud infrastructure-provided services or could consume valuable memory that could otherwise be used to run virtual machines.

Latency The introduction of a new component could also affect the response times for virtual machine start-up, network latency, resource availability and the like.

Reliability The introduction of a new component could affect the redundancy. Hence, it could be a single point of failure or even unintentionally imede the availability of existing services in the infrastructure due to errors or vulnerabilities in the new component.

2.1.3.2 Demand

New components will only be adapted, when there is

1. need on the customer side or
2. a law regulation forcing the providers to do so.

Otherwise, there is no motivation in doing so. First, the development of a component is costly and second, this is followed by maintenance costs. If the customers do not demand for the component, the cost-benefit ratio has a negative impact on the cloud provider. Sometimes, cloud providers are forced to implement components or processes that are required by law. Usually, this is due to the fact that the cloud customer is the **data owner**, but not the **data processor** (cloud service provider).

2.1.3.3 Risk & Market Penetration

When introducing a new service (component), there is a certain risk that it is not accepted by the customers. The benefit of this service has to be as clear as possible to the customer in order for him to estimate the benefit and hence to be willing to pay a potentially higher fee. This is why cloud service providers are somewhat reluctant to adapt new services (which are usually entangled with new costs, see above) until the competing cloud providers will also introduce that new service. Otherwise, if the benefit of the new service is unclear to the customer, they might only see the increased costs that are tied to its introduction.

2.1.4 Requirements Analysis for Transfer to Large-Scale Scenarios

Now that we have discussed the technical pitfalls in the previous sections, we analyse which requirements are needed during the design phase of an algorithm or software so that these cloud hurdles can be gracefully overcome.

Algorithm must be scalable - If an algorithm is parallizable, it is not necessarily sufficient to be used in a large-scale multi-tenant environment, where every customer wants to have the same performance experience regardless of the load of other systems. If an algorithm is parallelizable but does not scale linearly, it effectively slows down when used by several parties.

Algorithm must be elastic - Elasticity means that an algorithm must not be based on fixed assumptions about the parallel work it can distribute, as the cloud is a highly dynamic infrastructure that constantly changes. If calculation nodes are added and removed as a consequence of load balancing or hardware failures, optimizations for a fixed assumption of n nodes will fail if n changes.

2.1.4.1 Performance/scalability

During the design, attention should be paid whether an algorithm is parallelizable in general. Some algorithms cannot be parallelized because executions or results depend on each other or it would distort results. However, MapReduce [DG04] that was first introduced by Google, can be applied to parallelize the processing of huge data sets. MapReduce works on so-called embarrassingly parallel problems, that are workloads for which little to no effort is required to divide them in sub-problems. In MapReduce, the *Map* phase splits the problem and dataset in smaller chunks that can still be solved individually and distributes them on nodes that work in parallel. These nodes may in turn again use a *Map* phase in order to divide their problem. Then, the *Reduce* phase combines the delivered results so that the original question or problem performed on the large dataset can be answered.

2.1.4.2 Elasticity of the solutions

Scalability is a necessary step towards a large-scale cloud adaptation, but not a sufficient one. An algorithm can work splendidly on a fixed number of n nodes, but cannot deliver adequate performance if n changes. A cloud, however, is growing and shrinking rapidly every day. This is due to failures and lack of resources, new hardware being added or power saving being applied. The static assumptions of grid computing and clusters do not apply to a highly dynamic environment such as the cloud.

2.1.4.3 Trust Establishment and Privacy Issues

Establishing trust in the cloud provider is always based on gaining knowledge about the provider's processes and infrastructure. Trusted Computing for example can be used to attest that a known-good software is running on a remote, untrusted machine. The large-scale characteristic of the cloud, however, renders this easy and secure process infeasible because it has two drawbacks:

1. Virtual machines are scheduled on physical hosts in a way not transparent to the user. Hence, every physical host a VM runs on has to be attested to the customer.

2. This leads to identifiability of the physical hosts and scheduling algorithm or corporate secrets in general and gives customers an insight that cloud providers are probably not willing to share.

2.2 Federated Security Management for the TrustedInfrastructure Cloud

We analyse both the technical requirements and security requirements of federation in the management of the TrustedInfrastructure Cloud, and especially Trusted Virtual Domains (TVD), which were both introduced in Deliverable D2.1.1 Chapter 13. In short, the TrustedInfrastructure Cloud is based on Trusted Computing technology, which ensures and measures integrity of all components (management and servers) and provide Trusted Virtual Domains, which are isolated virtual computing and networking resources, secured by encryption. With federation we mean the cooperation of the management components of different stakeholders which collaborate or provide infrastructure as a service to other stakeholders.

2.2.1 Requirements Analysis

The TrustedInfrastructure Cloud as used in the TClouds project was introduced in Deliverable D2.1.1 Chapter 13, and the key concepts of their management are described in Deliverable D2.3.1 Chapter 6. Therefore, in this section we only shortly summarize some of the core notions here.

In the TrustedInfrastructure Cloud, a management component, called TrustedObjects Manager (TOM), manages a set of appliances, in case of TClouds these are the TrustedServers (cf. Figure 2.1).

Central to Trusted Infrastructures is the concept of security zones called Trusted Virtual Domain (TVD) [CLM+10, CDE+10], which allows to deploy isolated virtual infrastructures upon shared physical computing and networking resources. By default, different TVDs are strongly isolated from each other.

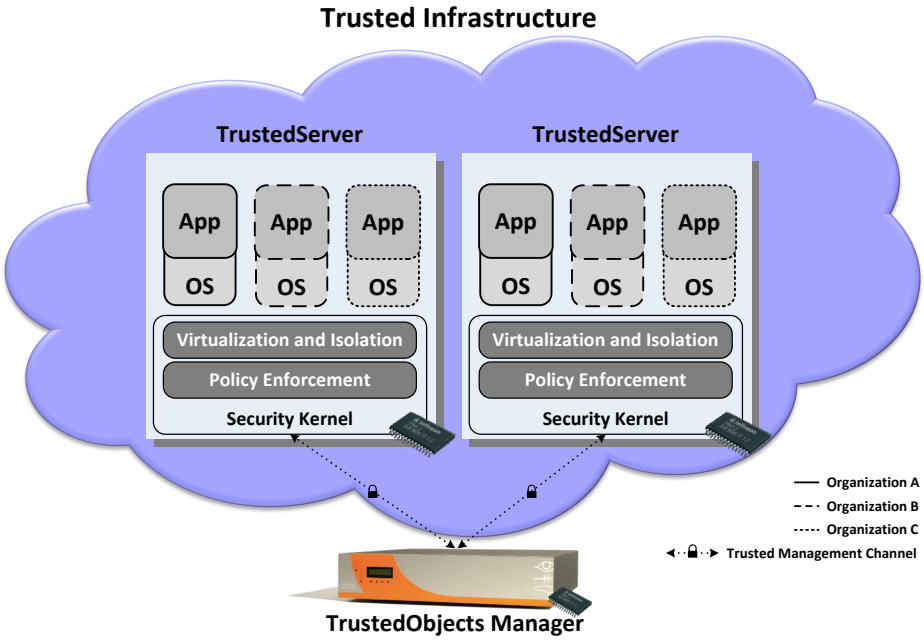


Figure 2.1: Schematic Trusted Infrastructure - TrustedServers managed by TrustedObjects Manager.

As a public cloud typically hosts various tenants, multi-tenancy of the management component is required. Therefore, the default behavior of the TrustedInfrastructure Cloud is to strongly isolate the TVDs of different tenants. However, in practice different organizations may want to collaborate which requires to weaken the isolation in a controlled way.

2.2.1.1 Application Scenarios

We describe several application scenarios that motivate the requirements and use-cases for federated security management. The technology and structure just described for the TrustedInfrastructure Cloud is not only applicable for cloud computing, which basically means management of servers. In fact, our work is based on previous work focused on trustworthy desktop systems and their management. In the following use cases this shows up as we also consider on premise systems as part of the 'TrustedInfrastructure'. For example an organisation wants to securely connect on premise systems (e.g. a desktop computer) with cloud systems in the same TVD. that The scenarios are based on the following actors:

- *Organisation* : A party (e.g. company or federal authority) that may want to extend their infrastructure for data and computation beyond their premise boundaries, e.g., use cloud resources or collaborate with another party.
- *Provider* : An organisation providing infrastructure, such as a cloud provider.

/AS 10/ Provider manages both cloud and organisation infrastructure

This is a very basic setup that is especially attractive to small companies that want to completely outsource the management of their infrastructure (see Figure 2.2). The provider manages both

the on premise infrastructure of the organisation (e.g., desktops, laptops, mobile devices) as well as cloud resources such as storage and servers. In this scenario, either the provider needs a management component that can both manage the organisations infrastructure as well as the cloud infrastructure, or the provider may delegate the infrastructure management to its customers, e.g., by offering them a management interface. Multi-tenancy of the management component is a crucial requirement here, as the provider needs to work on behalf of different organisations. This includes the infrastructure management as well as the security management.

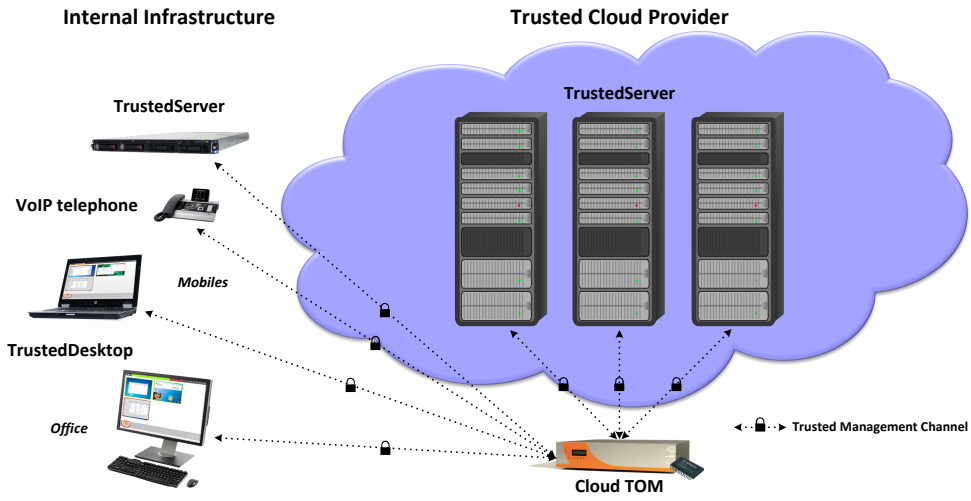


Figure 2.2: Scenario 1: Single infrastructure spanning internal and cloud resources

/AS 20/ Provider manages cloud and organisation manages own infrastructure

In this scenario the organisation manages their own on premise infrastructure and only extends its infrastructure into the cloud managed by the provider (see Figure 2.3). Resources in the providers infrastructure are then allocated on behalf of the organisation.

Since the organization's infrastructure and security policy is defined by the management component of the organisation, it conceptually acts as the master, while the management component of the provider acts as a slave. Therefore, the provider's management component retrieves management information and deploys it in its cloud infrastructure.

/AS 30/ Two organisations collaborate

In this scenario two organisations host their own infrastructure, independent of each other. They decide to collaborate and want to be able to exchange information (see Figure 2.4). Each organisation also has their own TVD-based security policy managed by their own management component. Exchange of information is governed by a shared TVD which has to be established between the two management components. In contrast to Scenario /AS 20/, there is no conceptual master / slave relationship between the organisations. This also includes that the organisation does not allocate infrastructure resources on the other organisations infrastructure.

/AS 40/ Two organisations collaborate via third party cloud

This scenario combines Scenario /AS 20/ and Scenario /AS 30/. As in Scenario /AS 30/ two organisations decide to collaborate and exchange information via a shared TVD. In contrast

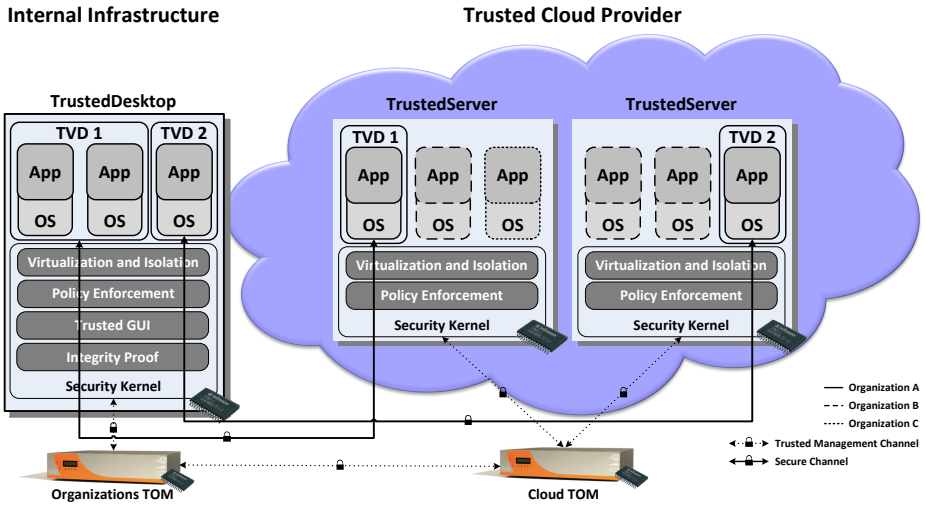


Figure 2.3: Scenario 2: Extend infrastructure into cloud

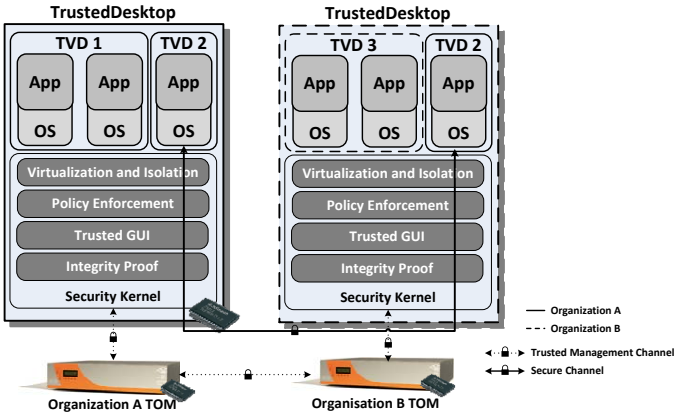


Figure 2.4: Scenario 3: Direct collaboration between two organisations

to Scenario /AS 30/, this shared TVD is deployed on the infrastructure of a cloud provider to establish shared storage or shared computation resources (see Figure 2.5).

2.2.2 Security Requirements

This section describes the security environment, including assumptions, assets, the threat model, and the security objectives related to the technical requirements described in subsection 2.2.1

2.2.2.1 Assumptions

We now describe the assumption on the security aspects of the environment in which the federated security management will be used. This includes

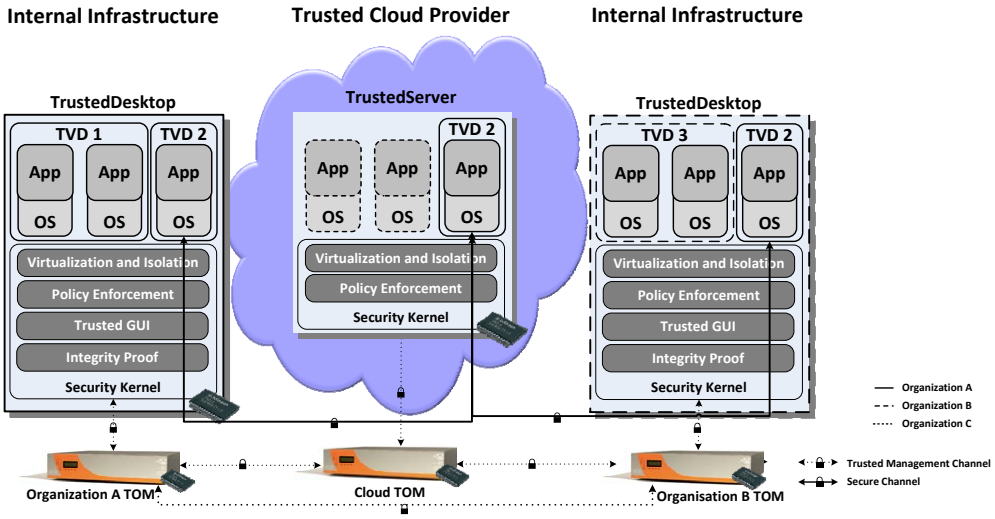


Figure 2.5: Scenario 4: Collaboration between two organisations via cloud resources

- information about the intended usage of the management component, including such aspects as the intended application, potential asset value, and possible limitations of use, and
- information about the environment of use, including physical, personnel, and connectivity aspects.

In particular, this includes the following assumptions:

/A 10/ Trusted Platform

The Trusted Infrastructure components (TOM and TrustedServer) are built on top of a trusted platform, employing trusted computing technology (e.g. TPM and remote attestation) and a security kernel.

/A 20/ Isolation of Organisations

Within a single Trusted Infrastructure, organisations are isolated from each other.

/A 30/ Inner Organisation Information Flow

Within an organisation, TVDs are isolated by default and an information flow is only granted according to the organisations security policies, in particular the information flow rules.

2.2.2.2 Assets

The assets describe the primary information under control of the security management. In general, we distinguish *data* from *information*. Data contains information, but this information might be encrypted, e.g. within a TVD. So data may be exchanged without revealing the information by means of cryptography. The assets that might be compromised by adversaries and should be protected by security objectives are:

/D 10/ TVD Information

The primary asset is the information stored within a TVD. This is the organisations primary data which may be mission critical for their business.

/D 20/ TVD Policy

The secondary asset is the TVD policy itself, especially the information flow policies which determine the admissible information flows.

2.2.2.3 Threat Model

We now identify possible threats that may compromise the assets.

/T 10/ Unauthorized TVD Sharing

An organization may get unauthorized access to TVD information of another organization by connecting one TVD of its own environment to one TVD of the other organization without authorization.

/T 20/ Modified Hypervisor

A malicious administrator of a cloud provider may get unauthorized access to TVD information of a hosted organization by modifying the used hypervisor such that it, e.g., copies the TVD information in regular intervals.

/T 30/ Modified Security Policy

A cloud provider may get unauthorized access to TVD information of a hosted organization by modifying the security policy of its hosted infrastructure of that organization.

/T 40/ Unauthorized Storage Access

A malicious administrator of a cloud provider may get unauthorized access to TVD information by reading the TVD information from persistent storage such as hard disks of the cloud infrastructure.

/T 50/ Unauthorized Network Access

A cloud provider may get unauthorized access to TVD information by reading the TVD information from its network.

2.2.2.4 Security Objectives

Now we describe the security objectives the federated management component should guarantee.

/O 10/ Inter Organisation Information Flow

Access to TVD information by an external organisation is permitted only according to the inter-organisation information flow policy. After an organisation disconnect a shared TVD, no new information should be accessible for other organisations.

/O 20/ Security Policy Integrity

The provider must not be able to modify the security policy enforced on behalf of a managed organization.

/O 30/ Trusted Hypervisor

Only authorized hypervisor configurations running on top of the of the provider's infrastructure should have access to TVD information of hosted organizations.

/O 40/ Secure TVD Information

The integrity and confidentiality of TVD information under control of the cloud provider's infrastructure, including computation, network, and persistent storage has to be guaranteed.

/O 50/ Authorized Sharing of TVDs

Sharing of two TVDs of two organizations is only allowed if both involved organizations agree and if the security policy of the used TVDs do not contradict.

2.2.3 Use Cases

From the scenarios above we have extracted the use cases that have to be provided by the federated management component.

USE CASE UNIQUE ID	/UC 10/ (Add organisation)
DESCRIPTION	A cloud provider adds a new organisation to the infrastructure
ACTORS	Provider
PRECONDITIONS	Organisation not yet part of infrastructure
POSTCONDITIONS	Organisation is ready to use resources of infrastructure
NORMAL FLOW	1. An administrator of the cloud provider adds the organisation in the management console

USE CASE UNIQUE ID	/UC 20/ (Register provider)
DESCRIPTION	Register cloud provider within internal infrastructure and specify which TVDs may be deployed in cloud
ACTORS	Organisation
PRECONDITIONS	
POSTCONDITIONS	Organisation can deploy resources within the providers infrastructure
NORMAL FLOW	1. An administrator of the organisation adds a provider in the management console and selects TVDs which may be deployed

USE CASE UNIQUE ID	/UC 30/ (Allocate provider resources)
DESCRIPTION	An organisation allocates resources (virtual machines or storage) in the providers infrastructure
ACTORS	Organisation
PRECONDITIONS	Provider is registered and TVD is selected to be deployed
POSTCONDITIONS	Organisation can access the resources from within the same TVD
NORMAL FLOW	1. An administrator of the organisation allocates resources from his management console

USE CASE UNIQUE ID	/UC 40/ (Deregister from provider)
DESCRIPTION	An organisation deregisters from using providers infrastructure
ACTORS	Organisation
PRECONDITIONS	Organisation is registered with the provider
POSTCONDITIONS	Organisation can no longer access resources at the provider
NORMAL FLOW	1. An administrator of the organisation unregisters via his management console

USE CASE UNIQUE ID	/UC 50/ (Register other organisation for collaboration)
DESCRIPTION	An organisation registers an other organisation for collaboration and specifies which TVDs are available for the other organisation to share information
ACTORS	Organisation
PRECONDITIONS	
POSTCONDITIONS	The other organisation can select a TVD to share information
NORMAL FLOW	1. An administrator of the organisation selects the other organisation and the TVDs

USE CASE UNIQUE ID	**/UC 60/ (Select a shared TVD from other organisation)**
DESCRIPTION	An organisation selects a shared TVD from an other organisation
ACTORS	Organisation
PRECONDITIONS	The TVD of the other organisation selected as shared TVD
POSTCONDITIONS	Both organisations can now use the shared TVD to exchange information
NORMAL FLOW	1. An administrator of the organisation selects the other organisation shared TVD

Chapter 3

Models and Languages for Security Requirements

Chapter Authors:
Sören Bleikertz, Thomas Groß (IBM),
Gianluca Ramunno, Roberto Sassu, Paolo Smiraglia (POL)

Cloud computing and virtualized infrastructures are often accompanied by complex configurations and topologies. Dynamic scaling, rapid virtual machine deployment, and open multi-tenant architectures create an environment, in which local misconfiguration can create subtle security risks for the entire infrastructure. This situation calls for automated deployment as well as analysis mechanisms, which in turn require policy languages and models to express security goals for such environments.

In Section 3.1 of this chapter, we propose a practical tool-independent policy language for security assurance. Our policy proposal has a formal foundation, and still allows for efficient specification of a variety of security goals, such as isolation. In addition, we offer language provisions to compare a desired state against an actual state, discovered in the configuration, and thus allow for a differential analysis. The language is well-suited for automated deduction, be it by model checking or theorem proving.

In Section 3.2, we aim to introduce the Trusted Virtual Domains (*TVD*) model in the Cloud architecture in order to provide customers with a protected environment for their *VMs* where the containment and trust properties are guaranteed by the infrastructure and, at the same time, to address the need for a strong isolation between tenants. This model is suitable for automated deployment and an implementation based on *libvirt* is described.

3.1 A Virtualization Assurance Language for Isolation and Deployment

3.1.1 Introduction

Cloud and large-scale virtualized infrastructures give rise to complex configurations. This complexity is a side-effect of highly dynamic scaling, rapid machine deployment and open multi-tenant systems. The complexity renders clouds challenging to administrate with respect to achieving all security requirements. This holds for cloud providers as well as their subscribers. The side effects of configuration changes to high-level security goals, such as isolation of tenants, are non-trivial at best. This is particularly true when performing local low-level configuration changes.

Indeed, research has established that configuration problems in complex environments are likely to result in security problems. A study of problems in large-scale Internet services by Oppenheimer et al. [OGP03] highlights configuration problems as the major source of security issues. We conjecture that these results also apply to virtualized infrastructures and clouds, because these infrastructures are large-scale, interconnected, heterogeneous systems, as well. In addition, studies by Berger et al. [BCP+08] point out a complexity increase introduced by the virtualization of the infrastructure.

Given the impact of configuration problems on large-scale infrastructures, we suggest that global high-level security properties must be verified complementary to local low-level ones. This is because local security properties do not compose gracefully to fulfill goals for the entire topology. Let us exemplify this rationale in the case of isolation for a multi-tenant virtualized infrastructure. Even if an administrator configures all resources well with regard to local policy decisions, such as firewall policies for virtual machines or access control policies for virtual storage, there still may be information flow through the connections of the topology, be it by covert channels between virtual machines on the same hypervisor, inter-zone VLAN traffic, or shared physical storage areas.

The complexity of cloud configuration with respect to assuring high-level security goals is tantalizing. It calls either for infrastructure-wide access control and deployment mechanisms to enforce the security goals automatically or for verification mechanisms to check for breaches of the goals. In any case, we need a specification language for high-level assurance goals. Such a language plays a different role in the three cases mentioned: *First* in the access enforcement case, the security assurance language is an auxiliary input to the policy decision engine that has in turn the function to ensure that the high-level assurance goals are preserved by access requests. *Second* in the automated deployment case, the deployment mechanism establishes deployment patterns that maintain the high-level security goals. Best practices and deployment templates that incorporate some security targets are insufficient to fulfill high-level security goals for the entire topology, because a series of local configuration transitions, which fulfill a local-view security property, may still breach a topology-level security goal in a global view. *Third* in the verification case, the high-level security goals constitute the verification target against which the actual infrastructure is evaluated.

There already exist specification languages for virtualized environments. These languages aim at provisioning (cf. [DMT10, MGHW09]), or network and reachability properties, e.g., firewall topology or distributed network access control [DDLS01]. In the former case, the specification languages are restricted to single resources, notably virtual machines, however do not have provisions for statements over the topology. In the latter case, the languages have provisions to model the topology and properties thereof, however they do not provide language primitives for expressing diverse security statements as needed in virtualized infrastructures.

We derived the following three categories of interesting security statements for virtualized infrastructures from existing research literature such as [BCP+08, OGP03, RTSS09a]: operational correctness, failure resilience, and isolation. *First*, operational correctness ensures that services are correctly deployed and that their dependencies are reachable. *Second*, failure resilience ensures that the effects of single component failures cannot cascade and affect many entities. *Third*, isolation ensures that different security zones are properly separated and that traffic between security zones is only routed through trusted guardians.

The goal of this work is to study such high-level security properties of virtualized infrastructures and propose a policy language to express these as goals. We call the resulting language Virtualization Assurance Language for Isolation and Deployment (*VALID*).

3.1.1.1 Contribution

We contribute the first formal security assurance language for virtualized infrastructure topologies. More precisely, we model such an assurance language in the tool-independent Intermediate Format IF [AVI03], which is well suited for automated reasoning. We lay the language's formal foundations in a set-rewriting approach, commonly used in automated analysis of security protocols, with access to graph analysis functions. In addition, we propose language primitives for a comparison of desired and actual states. As a language aiming at expressing topology-level requirements, it can express management and security requirements as promoted by [DDLS01]. Management requirements in the cloud context are, for instance, provisioning and de-provisioning of machines or establishing dependencies. Security requirements are, for instance, sufficient redundancy or isolation of tenants. To test soundness and expressibility of our proposal, we model typical high-level security goals for virtualized infrastructures. We study the areas deployment correctness, failure resilience, and isolation, and propose exemplary definitions for respective security requirements in *VALID*.

3.1.1.2 Outline

We structure this work in a top-down way. We first propose infrastructure-level assurance goals for virtualized systems in Section 3.1.2. These goals are a diverse sample of the language scope. In Section 3.1.3, we specify our requirements on the cloud assurance language on a meta-level. We lay the language's formal foundations in Section 3.1.4, that is, we introduce its roots in the Intermediate Format IF [AVI03] and our cloud-specific language primitives and syntax. In Section 3.1.5, we propose formal specifications of checkable attack states for the assurance goals defined in Section 3.1.2. Thereby, we exemplify the use of *VALID* in its application domain. We briefly discuss a virtualization assurance tool that would incorporate *VALID* in Section 3.1.6. We compare our cloud assurance language proposal to other policy language and virtualization security efforts in Section 3.1.7.

3.1.2 Virtualized Systems Security Goals

We distilled three categories of virtualized systems security goals based on common problems described in existing research literature: *Operational Correctness*, *Failure Resilience*, and *Isolation*. Furthermore, for each of these categories we identified specific goals that our language should be capable of capture and express efficiently. Figure 3.1 depicts a simple virtualized system example that we will use to illustrate the different security goals.

3.1.2.1 Operational Correctness

Operational correctness describes that a service is both correctly deployed and reachable. It bears some similarity to the *Liveness* property introduced in [AS86, Lam77] and informally states that "good things" will eventually happen for a service. Configuration mistakes often lead to unavailability of services in traditional data center environments (cf. [OGP03]) and is only intensified in virtualized environments due to their increasing complexity (cf. [BCP+08]).

Deployment correctness means that an entity is deployed in correct operational conditions, which includes multiple factors: i. The geographic location of the host system can have legal and technical consequences, e.g., conflicts with privacy laws, or long end-to-end delay due to geographic disparity. ii. Properties of the host system such as capabilities and reliability can

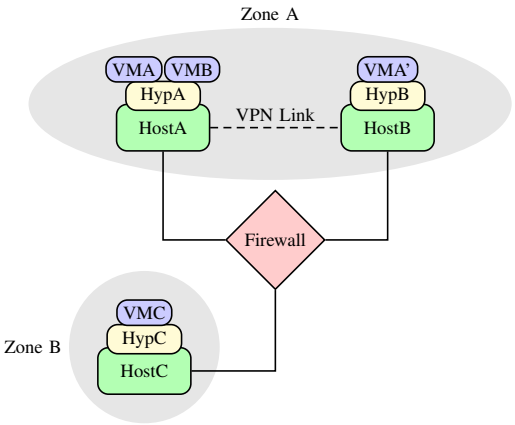


Figure 3.1: Virtualized System Example

have a significant impact on the service. iii. Furthermore, the configuration of the host system and service has to be correct that the service can actually be run on the host.

Reachability means that an entity is connected to all its operational dependencies. On one hand, these dependencies can be network reachability, i.e., the VM and physical host are actually reachable over the network from the client-side. On the other hand, these dependencies can be resource dependencies in general, e.g., that a VM is able to access services on other nodes. All such dependencies have to be fulfilled in order that the operational correctness of the service is given.

3.1.2.2 Failure Resilience

Failures of components in a computing environment are unavoidable, but a resulting failure of services, which are visible to the end users, can be mitigated. Such containment of component failures are pointed out in [OGP03] and can be summarized as: failure compartmentalization due to *Independent failure*, and prevention of cascading failures and limitation of failure impact due to the *Redundancy*.

Independent failure means that failures of an entity are well-contained and that dependencies of entities with the same function will fail independent from each other. This goal nurtures a diversity of the components deployed in the computing environment. A typical software stack in a virtualized system consists of a hypervisor, management operating system, virtual machine system, and the service application. A diversity in this stack, such as using different hypervisors from different vendors, will have an isolated failure in case of faults in one of these hypervisor implementations. Independent failure can be satisfied in the example scenario, in case the hypervisors *HypA* and *HypB* hosting the service VMs are provided by different vendors.

Redundancy means that sufficient replication enforces that individual component failures will leave overall service availability unharmed. The necessary level of redundancy depends on the desired failure resilience for a service, which also depends on its criticality. Sufficient redundancy implies the absence of a single point of failure (SPoF). A SPoF exists in a system, if a dependency of a service is only satisfied by one entity in the whole system. The absence of such a SPoF entity will increase the failure resilience due to the limitation of a cascading

failure effect on dependent services. In our example, the service running in *VMA* is replicated in *VMA'*, both running on different physical machine and interconnected with two independent network links.

3.1.2.3 Isolation

In virtualized environments, such as public infrastructure clouds, we see multi-tenancy in order to increase the utilization of the system. Isolation compares to *Safety* [AS86, Lam77] that undesired information flow do not happen. In [RTSS09a], the problem of undesired information flow in public infrastructure clouds was exposed.

Isolation of zones means that specified security zones are isolated from each other, either by correct association of machines to zones or by enforcement of flow isolation between any entity of different zones. A security zone can be any set of entities in the virtualized environment. For example, a zone in the case of tenant isolation is the set of resources used by a tenant, and zone isolation is given if the tenants do not have access to common resources. In the illustrated example, we have defined two security zones *Zone A* and *Zone B* that are disjoint, i.e., isolated of each other.

Guardian mediation means that information flow between zones is allowed if, and only if, mediated by a trusted guardian. In case information flow is allowed between the two security zones defined in our example case, the *Firewall* guardian has to mediate the traffic between the zones.

Chinese wall The *Chinese wall* policy [BN89] in the context of virtualization security describes that a physical host is not serving VMs of conflicting tenants. Such a policy can be implemented using the sHype [SVJ+05] hypervisor. *VMA* and *VMC* in our example case are virtual machines of conflicting tenants, therefore they can not be hosted on the same physical host with regard to the Chinese wall policy.

Secure channels The goal of secure channels describes that certain information flow is only permitted over secure channels, such as provided by VLAN or VPN in terms of network resources. A VPN link is established between *HostA* and *HostB* in our example that acts as a secure channel.

3.1.3 Requirements

3.1.3.1 Formal Foundations

Virtualized environments can gain complexity beyond human oversight and therefore require tool-supported deployment and analysis. Thus, we expect the security assurance language to have formal rigor and be suitable for automated reasoning. This requirement implies a simple, mathematical structure with controllable state space.

3.1.3.2 Expressibility

There are many different security requirements imposed on virtualized infrastructures. Therefore, we require that the security assurance language needs to be able to efficiently express a

wide range of security properties as discussed in Section 3.1.2. *First*, the language needs to have *three expression layers*: i. statements about properties of resources, e.g., their IP address or functional classification, ii. set operations, such as membership in security zones, iii. graph operations, such as existence of an information flow or dependency path in a graph model of the topology. *Second*, the language needs to be *reflexive* and *self-contained*, that is, one can define new security goals with the existing terms of the grammar and without the need of auxiliary grammar.

As a corollary of this requirement, we propose that the security assurance language shall express *attack states*, that is, states in which a security property is violated, as well as *ideal states*, that is, states that assure a correct system behavior. Whereas the first approach is suitable for more efficient security analysis (model checking) without complete state exploration, the second approach is suitable for complete verification (theorem proving).

3.1.3.3 Tool and Standard Independence

Virtualized environments are still a young field without settled predominant standards. Therefore, we require the specification language to be independent from a specific vendor's tool or a specific standard.

3.1.3.4 Desired State Comparison

The validation of security properties of virtualized environments provides two different views on the state of such a virtualized infrastructure: a *desired* state or the *ideal world*, as specified in the policy, and an *actual* state or *real world*, i.e., the current configuration of the virtualized infrastructure. One specific goal of our assurance language is to express comparisons of a desired state and an actual state discovered in a configuration. Sometimes it is necessary to make statements about ideal elements as well as real elements in the very same policy statement. Consider the example that a VM should be hosted on a specific host. Or in other words, the goal is breached if the VM is hosted on a different machine than specified. This breach can be efficiently captured using both elements from the ideal and real world in one policy statement. We specify that we have an ideal machine hosting the VM and also a real machine hosting the same VM. In order to describe the placement breach, we say that these two machines do not correspond to each other, i.e., the real machine is not the same as the ideal one in terms of the given properties. Therefore, if such a statement holds, we observed a placement breach.

3.1.4 Language Syntax

We propose a specification and reasoning language for security properties of virtualized environments based on set-rewriting and conditions over states.

VALID uses a subset of the AVISPA Intermediate Format IF [AVI03] as its basis, a meta-language for automated deduction based on set manipulation and conditions over state expressions. We chose IF as the basis for our work because of its capability to efficiently express goals as stated in Section 3.1.2, its natural extensibility to state transition formulations, its tool-independence, and its close relation to general-purpose automated deduction, which is given due to the strong formal foundation of IF, and its support by model checkers and theorem provers.

Table 3.1: Basic type constants for virtualized infrastructures.

Type Symbol	Description
node	denotes the superclass of types in $\mathbb{T}_N$.
machine	denotes a virtual machine.
hypervisor	denotes a hypervisor on a host or VM.
host	denotes a physical host.
machineOS	denotes an operating system of a virtual machine.
hostOS	denotes an operating system of a physical host.
network	denotes a network component.
zone	denotes an isolation zone of an infrastructure.
class	denotes a functional class of similar components.

3.1.4.1 Term Algebra and Atomic Terms

We start from *atomic terms*, that is constants and variables. The value of a constant is fixed, e.g., the symbol for the type machine. We call the set of all constant terms *signature*. A variable can be matched against any value (of matching type). Atomic terms with different symbols have different values.

Definition 1 (Term Algebra) *We define a term algebra over a signature Σ and a variable set $\mathcal{V}$. Constants and variables are disjoint alphanumeric identifiers: constants start with a lower-case letter; variables start with an upper-case letter. We typeset IF elements in* sans−serif.

The signature Σ contains a countable number of constant symbols that represent resource names, numbers and strings.

The atomic terms are typed (see Table 3.1):

Definition 2 (Type System) *We have a set of basic types:*

$$\mathbb{T} := \{\mathsf{node}, \mathsf{machine}, \mathsf{host}, \mathsf{hypervisor}, \mathsf{machineOS}, \mathsf{hostOS}, \mathsf{network}, \mathsf{zone}, \mathsf{class}\}$$

We write $t : \tau$ for a term t having type τ. Variables can be untyped or typed. If a variable has a basic type, it can generally only be matched against a constant with matching type. The type symbol node *represents a super-type: variables of type* node *can match against types in the sub-set:*

$$\mathbb{T}_N := \{\mathsf{machine}, \mathsf{host}, \mathsf{hypervisor}, \mathsf{machineOS}, \mathsf{hostOS}, \mathsf{network}\}$$

To analyze topologies, we model virtualized infrastructure configurations as graphs. Whereas the basic graph, called realization, is a unification of vendor-specific elements into abstract nodes, we introduce further graph transformations to model information flow and dependencies.

Definition 3 (Graph Types) *A* graph type $G \in \{\mathsf{real}, \mathsf{info}, \mathsf{depend}\}$ *is a constant identifier for a type of a graph model:*

- real *denotes a realization graph unification of resources and connections thereof.*
- info *denotes a realization graph augmented with colorings modeling topology information flow.*
- depend *denotes a realization graph augmented with colorings modeling sufficient connections to fulfill a resource's dependencies.*

3.1.4.2 Function Symbols and Dependent Terms

Definition 4 (Function Symbols) Σ *contains a finite set of fixed* function symbols.

- $\mathsf{pair}(A, B)$ *denotes a pair.*
- $\mathsf{contains}(S, E)$ *denotes a untyped set membership relationship of a set S and element E.*
- $\mathsf{matches}(I, R)$ *denotes the correspondence between an element of the ideal world I and the real world R. Both elements I and R must have the same type.*
- $\mathsf{edge}([G : \mathsf{real}];\ A, B)$ *is a predicate, which denotes the existence of a single edge between A and B with respect to an (optional) graph type G.*
- $\mathsf{connected}([G : \mathsf{real}];\ A, B)$ *is a predicate, denotes existence of a path between A and B, respect to an (optional) graph type G.*
- $\mathsf{paths}([G : \mathsf{real}];\ A, B)$ *denotes the complete search of all paths between A and B, with respect to an optional graph type G. The resulting type of the function is a set of edge pair sets.*

The notation $[A : v]$ *denotes an optional argument A with default constant value v.*

Observe that the graph functions allow an optional graph type argument G (Definition 3), which specifies the graph type the function is applied to.

We introduce the notion of *dependent terms* to model access to resource properties, such as IP address $\mathsf{ipadr}(M)$ or image type $\mathsf{imagetype}(M)$ of a machine M.

Definition 5 (Dependent Term Function Symbols) *A* dependent term *is a function symbol denoting the mapping of constant values to atomic terms.* Σ *includes a fixed set of constant symbols for dependent terms.*

3.1.4.3 Facts, State and Conditions

VALID aims at reasoning over secure and insecure states of a cloud topology, which we model as a set of known facts.

Definition 6 (Facts and State) *A* Fact *represents a Boolean piece of knowledge: it can be either* true *or* false. *A state is a set of ground facts. We express such sets by a dot-operator ("."), that is, a commutative, associative, idempotent operator, which joins all elements of a state.*

Conditions restrict state terms with auxiliary predicates:

Definition 7 (Condition) *A condition is an inequalities predicate over terms. We define the condition function symbols for equality* equal(*Term*, *Term*) *and less-or-equal* leq(*Term*, *Term*) *over terms as well as negation* not(*Condition*) *and conjunction operator* & *Condition over conditions with their natural semantics.*

3.1.4.4 State Transitions

In general, an IF specification consists of an initial state and a finite set of *transition rules*, defining a transition relation.

Definition 8 (Transition Rules) *Transition rules have form*

$$PF.NF \;\; C \quad =[V]\Rightarrow \quad RF$$

where

- PF *and* RF *are sets of facts,* NF *is a set of* negated facts *of the form not*(f) *where* f *is a fact,*
- *C is a set of conditions and*
- *V is a set of variables.*

We distinguish the left-hand side (LHS) defining the preceding state and the right-hand side (RHS) defining the result state. The variables V are existentially quantified in the rule to introduce fresh variables during transitions. RF defines the resulting facts. The variables of RF must be a subset of the variables of the positive facts PF and the existentially quantified variables V.[1]

Note that transition definition does not enforce transition determinism, that is, that result states are unambiguously defined from the preceding state. IF, being a formal language for model checking, focuses on exploring the state space and determining reachability of attack states, possibly following multiple routes.

We focus on specification of security goals over static states and will only specify initial and goal states. We leave analysis of dynamic systems to future work.

3.1.4.5 Goals

We define *goals* by specifying an abstract state which constitutes attaining the goal. For an analysis we *pattern-match* a Fact set modeling the goals constrained by a conditions list against the actual analysis state.

Definition 9 (Goal) *A* goal state *is a set of positive and negative facts constrained by a (potentially empty) condition list. It is specified with a unique identifier, an optional graph type G and a variable list as interface. It has the form:*

$$\begin{array}{l}\textsf{goal} \;\; \textit{Identifier}\;([G:\textsf{real}];\; \textit{VariableList}) := \\ \qquad PF.NF \;\; C\end{array}$$

where PF *and* NF *are positive and negative fact sets and C a condition list. The graph type G determines the graph type of unparametrized graph functions used in the goal.*

[1] Note that this excludes the variables only occurring in negative facts and conditions.

Example 1 (Goal) *Let us consider a simple isolation breach attack state, which matches against a state, in which disjoint zones* ZA *and* ZB *contain machines* MA *and* MB *respectively, and in which there exists an information flow path between these two machines. It is determined as information flow goal by the graph type* info. *Observe that the goal is defined over variables and can match against any state with constant zones and machines fulfilling this relation and that the matching values must be different.*

```
goal isolation_breach (info; ZA,ZB,MA,MB) :=
  contains(ZA,MA).contains(ZB,MB).
    connected(MA,MB)
```

3.1.4.6 Structured Specifications

Specification of our language consists of distinct sections: The *TypesSection* introduces all atomic terms that will be used throughout the analysis. The type section may have two subsections for real and ideal type declarations. The *InitsSection* specifies initial knowledge on entities. For instance, here one would specify properties of machines that can be used for identifying the machine, such as the machine's IP address as Condition over machine properties. Knowledge specified here can be about ideal and real entities. The *RulesSection* specifies the knowledge on the structure of the virtualized infrastructure. For instance, it specifies which machine elements are associated with which isolation zones. Note that the topology specified in this section is particularly important to model the system's ideal state. Finally, the *GoalsSection* defines attack and assurance states which are matched against analysis results.

3.1.4.7 Dual Type System

We introduce the declaration of ideal and real types, that is a *dual type system*.

Definition 10 (Dual Types) *For each constant or variable symbol, we explicitly declare symbol to be either universal or restricted to the ideal or real model. A declaration in the top-level of the TypesSection means universal, a declaration in the subsections* idealTypes *and* realTypes *restricts the declaration to the respective model. The* matches$(\cdot,\cdot)$ *fact denotes that two symbols of ideal and real world have a correspondence with each other.*

3.1.5 Attack State Definition

We model the security goals from Section 3.1.2 as abstract attack states. In case the state is reached, a tool will alert that the corresponding goal has been breached. This approach aims at security analysis by, for instance, model checking.

To facilitate an actual security analysis, one complements these abstract goals with specifications of the ideal state of the system in two areas: *First*, one defines the initial knowledge on entities (*InitsSection*), that is, properties modeled as dependent terms, such as IP address. *Second*, one defines the knowledge of the ideal structure of the topology (*RulesSection*) as initial state, that is, facts known on contains, matches or edge relations.

3.1.5.1 Operational Correctness

For the operational correctness from Section 3.1.2.1, we model deployment breach as exemplary attack state.

Deployment Breach: Deployment breach considers in how far VMs are placed on an incorrect hypervisor or physical machine.

Definition 11 (Deployment Breach) *A* deployment breach *is an attack state over some virtual machine* M *and two different hosts (*HA*,* HB*), in which* edge(HA, M)*, i.e.,* M *is hosted on* HA*, is a specified fact, but* edge(HB, M) *was observed.*

```
section types:
  M                          : machine
   subsection idealTypes:
     HA                      : host
   subsection realTypes:
     HB                      : host

section goals:
goal deploymentBreach (real; HA,HB,M) :=
  not(matches(HA,HB)).edge(HA,M).edge(HB,M)
```

After declaring Fact that HA does not match HB, the left-side of the statement contains the matched facts of the ideal world, that is, edge(HA, M), the right side of the statement the observed fact of the real world edge(HB, M).

Unreachability: Unreachability is an attack state that there does not exist a path between a machine and a dependent resource in the dependency graph.

Definition 12 (Unreachability) *A* unreachability *is an attack state over some machine* M *and a resource set* {RA, . . . , RN}*, on which* M *depends. The attack state is triggered if no dependency path between* M *and at least one of the needed resources* RI *exists.*

3.1.5.2 Failure Resilience

Single Point of Failure:

Definition 13 (Single Point of Failure) *A* single point of failure *is an attack state over any machine* M *and any two different resources (*RA*,* RB*) with equivalent function. A single point of failure exists if only* path(M, RA) *holds, but* not(path(M, RB)) *for any* RB*.*

In general, a single point of failure exists if there is only one dependency path between a resource and its dependencies. This requires knowledge what the dependencies of a certain resource (type) are and which other resources can fulfill the same function. For instance, for a network single point of failure, one may consider all network switches that connect to the Internet, independently from the ones connecting to the Intranet. We therefore define different attack state goals for different resource types and model the goals with functional classes of resources fulfilling the same purpose.

```
section types:
  M                          : machine
  NA, NB                     : network
  C                          : class

section goals:
goal singlePoF_Net (depend; NA,NB,M,C) :=
  contains(C,NA).contains(C,NB).connected(M,NA).
  not(connected(M,NB))
```

Interdependent Failure Behavior:

Definition 14 (Interdependent Failure Behavior) Interdependent failure behavior *is an attack state over two different machines* $(\mathsf{MA}, \mathsf{MB})$ *with the same functional class* C *and k pairs of resource and associated class, i.e., a specific implementation, such as:*

$$(\{\mathsf{RA1}, \ldots, \mathsf{RAN}\}, \mathsf{CRA}), \ldots, (\{\mathsf{RK1}, \ldots, \mathsf{RKN}\}, \mathsf{CRK})$$

We have an attack if for any two machines (MA*,* MB*) of class* C*, there exists a resource of the same class they both have in their stack.*

3.1.5.3 Isolation

Zoning & Isolation Breach: We specify a simple isolation analysis over machines and zones. Machines can be recognized by their properties, for instance an IP or MAC address. By the contains rule, we express that a machine is associated with zone (i.e., that the zone contains the machine).

Definition 15 (Zoning Breach) *A* zoning breach *is an attack state over a pair of machines (*MA*,* MB*) and zones (*ZA*,* ZB*), where either* MA *is declared to be in* ZA *and not present, or* MB *is declared not to be in* ZB*, but was found there in the real state.*

```
section types:
  MA, MB                    : machine
   subsection idealTypes:
     ZA, ZB                 : zone
   subsection realTypes:
     ZA0, ZB0               : zone

section goals:
goal zoningBreach_Missing (info; ZA,ZA0,MA) :=
  matches(ZA,ZA0).contains(ZA,MA).
    not(contains(ZA0,MA))
goal zoningBreach_Unknown (info; ZB,ZB0,MB) :=
  matches(ZB,ZB0).not(contains(ZB,MB)).
    contains(ZB0,MB)
```

Isolation breach is more complex as it incorporates the existence of information flow paths between zones.

Definition 16 (Isolation Breach) *An* isolation breach *is an attack state over any pair-wise different variable machines* $(\mathsf{MA}, \mathsf{MB})$ *and zones* $(\mathsf{ZA}, \mathsf{ZB})$*,* MA *in* ZA *and* MB *in* ZB*, in which there exists a path between* MA *and* MB*.*

```
section types:
  MA, MB                    : machine
  ZA, ZB                    : zone

section goals:
goal isolationBreach (info; ZA,ZB,MA,MB) :=
  contains(ZA,MA).contains(ZB,MB).
    connected(MA,MB)
```

Guardian Circumvention: Guardian Circumvention is an attack state corresponding to Guardian Mediation from Section 3.1.2. It means that there exist paths between machines that are not controlled by a trusted guardian.

Definition 17 (Guardian Circumvention) Guardian circumvention *is an attack state over any pair-wise different variable machines* (MA, MB)*, guardian* G *and zones* (ZA, ZB)*,* MA *in* ZA *and* MB *in* ZB*, in which there exists a path between* MA *and* MB*, which does not contain the guardian* G*. The attack state naturally extends to a set of multiple guardians.*

```
section types:
  G                              : guardian
  MA, MB                         : machine
  ZA, ZB                         : zone
  N                              : node

section goals:
  goal guardianCircumvention (info; ZA,ZB,MA,MB) :=
    contains(ZA,MA).contains(ZB,MB).connected(MA,MB).
    contains(paths(MA,MB),X).not(contains(X, (G,N))
```

3.1.6 Virtualization Assurance Tool

We report that we have implemented a virtualized systems assurance tool, which is able to discover heterogeneous virtualized infrastructures, such as ones based on Xen and VMware, and build up a unified graph representation thereof. The tool provides mechanisms for graph operations and information flow analysis. *VALID* needs to be integrated into such a tool for diagnosis purposes, that is, matching the security goals against the currently deployed system. We built a parser for our language grammar using ANTLR[2], a Java parser generator. We aim at integrating the parser as well as *VALID*-specific analysis capabilities into the assurance tool as future work. To project a *VALID* policy onto native IF as well as use it with an existing IF tool, our assurance tool has to resolve graph function symbols (edge, connected, paths) into the set of all valid graph assignments. In addition, the tool needs to translate knowledge about the discovered real state into policy statements about real entity properties and topology.

3.1.7 Related Work

Automated network infrastructure analysis Narain et al. [NCPT06] analyze network infrastructures with regard to *single point of failure* using a formal modeling language. In contrast, our approach focuses on providing a generic language to express a variety of high-level security goals, among them the absence of single point of failure. Previous work has also analyzed network reachability in an automated way, for example, [XZM+04] for IP networks, [KSS+09] for VLANs, and [BSP+10a] for cloud configurations. In terms of network manageability and configuration management, Ballani and Francis [BF07] propose a deployment language that overcomes the complexity of the low-level configuration. It allows the specification of high-level configuration goals to improve the manageability and was applied to network tunnels. Narain [Nar05] proposes modeling a network configuration using a formal language and do automated reasoning on this formal model.

[2] www.antlr.org

Formal languages for security policies and modeling *Ponder* [DDLS01] is an object-oriented formal specification language for access control policies and role management in distributed systems. However, it does not aim at expressing high-level security goals for virtualized infrastructure topologies. Kagal et al. [KFJ03] present a policy language for pervasive computing, which is similar to cloud computing environments with regard to their dynamic behavior. It is to express entitlements on actions, services, or conversations of an entity, such as an agent or user. Their implementation is based on Prolog. *Alloy* [Jac02] is a first-order logic modeling language, which is used, among other things, in network infrastructure modeling and analysis [Nar05, NCPT06]. Alloy can express structural properties as relations between objects as well as temporal aspects as dynamic models with states and allowed transitions. It has potential as suitable basis for our cloud assurance language, however we opted for IF as basis because of two reasons. *First*, IF has a strong formal foundation. *Second*, IF is supported by model checkers and theorem provers, such as AVISPA [ABB+05], SATMC [AC04], and OFMC [BMV05a] in combination with a fix-point evaluation exportable to Isabelle [Pau94].

Virtualized systems specification languages The *Open Virtualization Format* (OVF) [DMT10] is a standardized specification language for the packaging and distribution of virtual machines. OVF is used to describe general information and virtual resource usage for an individual virtual machine or a virtual appliance consisting of multiple VMs, but not for an entire virtualized infrastructure as in our approach. *Virtual Machine Contracts* [MGHW09] are a policy specifications based on OVF that govern the security requirements of a virtual machine, e.g., to specify firewall rules. Similar to OVF, the objective of this language is linked to provisioning rather than expressing high-level security goals on the topological level. On the hypervisor level, *sHype* [SVJ+05] is an implementation of access and isolation control for virtual machines, which uses a XML-based access control policy[3]. Again, the policy only applies to one entity in the virtualized system, i.e., the hypervisor hosting virtual machines.

3.1.8 Conclusion and Future Work

We studied virtualized systems security goals in the categories operational correctness, failure resilience, and isolation. We proposed a formal language to express such high-level security goals, which, unlike previous work, covers topological aspects rather than just individual virtual machines. We chose the *Intermediate Format* (IF) as formal foundation of our language because of its support by existing general-purpose model checkers and theorem provers. We demonstrated the ability of our language to efficiently express a diverse set of virtualized systems security goals by giving concrete specifications for a subset of the studied goals.

Further potential future work is to study dynamic models of virtualized infrastructures, in order to capture and analyze configuration changes and state transitions in general.

3.2 Deploying TVDs in the Cloud

3.2.1 Introduction

Cloud computing is one of the most promising technologies in these days since it allows a user to access a potentially unlimited set of virtualized resources to offer a service over Internet without buying the required infrastructure, thus avoiding the maintenance costs.

[3] Xen User Manual, Section 10.3.

In order to deploy his service, the user has only to determine the resources needed to run it in the cloud environment: the number of running instances, the amount of virtual storage space and the quantity of data that will be exchanged with the outside network. Then, he can deploy the service on the assigned *VM*s by configuring the operating system, the network and the needed software.

However, cloud computing raises new security issues that are not present in the case of ad-hoc infrastructure. In particular, from the Cloud providers perspective, the effort is concentrated in ensuring the isolation between tenants whose virtualized resources may have been allocated on the same physical nodes. Thus, as already stated in the Section 3.1.1, the configuration of the infrastructure must be validated against the security goals to avoid undesired information flows between different customers.

Besides, also for Cloud users there are security concerns that should be properly addressed. If an user wants to deploy a service using multiple instances, he must ensure that all his instances are configured correctly and the communication between each other is adequately protected. Otherwise, there may be the situation where an external *VM* may interfere with those executing the user service, so that the latter will not behave correctly.

One solution to address these issues is to consider a groups of *VM*s as an unique entity on which a security policy must be coherently enforced. A model that has been developed for this purpose is the Trusted Virtual Domain.

3.2.1.1 Background

Trusted Virtual Domains (*TVD*s) is a model for deploying business and IT services in complex and heterogeneous distributed systems. It aims to provide an operating environment where a secure operational policy is uniformly enforced over all entities and that has some verifiable properties, containment, trust and simplification.

While these properties can be in general ensured using existing technologies, the novelty of the *TVD* concept consists in the definition, in a consistent way, of high level operations required for managing the life cycle of this protected environment that are then mapped in component-specific instructions, relieving the burden for users to configure the underlying software and hardware.

A *TVD* consists of an *Execution Environment* (*EE*), (e.g. a virtual machine) and an abstract communication channel which allows the former to securely communicate with other *EE*s and can be implemented, for instance, using *VLAN*s and *IPsec*. Further, an *EE* is executed by a *physical node* through one or more *software container* (i.e. the hypervisor), which ensures the isolation from other running instances.

When a new *TVD* is defined, it must be specified the secure operational policy to be enforced among all *TVD* members. In particular, it is possible to specify functional requirements (i.e. limitations on what members can do), quality requirements (i.e. the adherence of a component to a specific standard) and implementation/mechanism mappings. The above requirements must be verified each time a new *EE* joins a *TVD*.

3.2.1.2 Our contribution

In this work we aim to introduce the *TVD* model in the Cloud architecture in order to provide customers with a protected environment for their *VM*s where the containment and trust properties are guaranteed by the infrastructure and, at the same time, to address the need for a strong isolation between tenants as already mentioned in the Section 3.2.1.

In particular, we study how a generic Cloud infrastructure must be enhanced to support *TVD*s by defining requirements for the physical nodes and for the Cloud Controller. Further, we analyze the language of *libvirt* in order to determine whether already defined elements allow to describe the *TVD* configuration and we introduce new elements to cover the missing parts. Among possible *TVD* implementations proposed in the design document [BGJ+05], we choose to describe the solution identified by *Case 1*.

3.2.2 Benefits of TVDs to the Cloud

The *TVD* model could give significant benefits to Cloud Computing in that it allows to create an environment from a group of *VM*s where well-defined security properties and the enforcement of a security policy are guaranteed by the underlying Cloud infrastructure. In particular, through the containment and trust properties it is possible to achieve the following security goals:

- **Secure communication among a group of *VM*s.** The containment property ensures that a *TVD* member can communicate only with other members without interception and interference from external entities. A customer may create a *TVD* to ensure that communication happens only among their *VM*s so that they cannot be attacked by instances of other tenants. This may eliminate the need of configuring the firewall to properly filter the network traffic.

- **Allow join to *TVD* only *VM*s with the desired trust level.** The trust property guarantees that a *VM* can join a *TVD* only if its level of trust is that expected. In particular, the level of trust depends of the security mechanisms implemented in the physical node and it must be verifiable by other *TVD* members. Using this property it could be possible to restrict the access to a *TVD* only to *VM*s running on physical nodes that have been verified through the TCG remote attestation.

- **Limit the damage from a compromised host.** Using the *TVD* model allows to limit the damage caused by a compromised host that tries to attack other *VM*s by sending malicious packets over the Cloud network. Indeed, an important assumption derived by the containment property is that a *TVD* can be considered isolated from other *TVD*s. This means that, if physical nodes are certified to properly handle these packets without being corrupted, a compromised host is able to attack only *TVD*s for which it is allowed to run an instance, leaving the others unaffected.

3.2.3 Enhancing the Cloud infrastructure to support *TVD*s

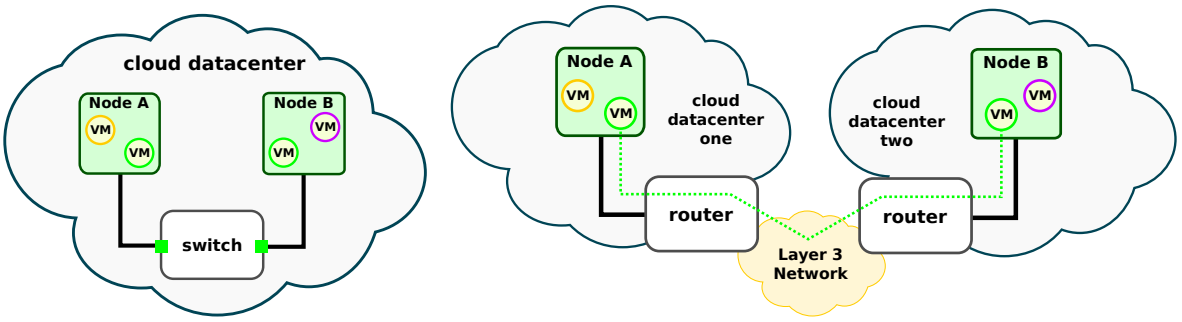
In order to provide support for *TVD*s in the Cloud, the infrastructure must be enhanced to properly implement all *TVD* operations and to ensure that the defined secure operational policy is uniformly enforced over all their members. This work is focused on evaluating existent Cloud components in order to determine the changes to the configuration model needed to describe *TVD*s and to guarantee the containment property.

Since the *TVD* is an abstract model that can be implemented using different technologies, we chose a specific solution among those proposed in [BGJ+05] to give the reader a concrete proposal about the requirements that must be satisfied to achieve our goal. In particular, we chose to implement the *Case 1*, which requires an hypervisor for the isolation, TCG-based attestation for the verification, and *VLAN + IPsec* for the channels.

In the following, we analyze required configuration changes by splitting them in three areas: management of *EE*s, communication channel between *EE*s and *TVD* management.

Management of *EE*s From the *TVD* model specifications, the *software container* in the node is expected to handle creation and destruction of *EE*s, resources provisioning and auditing operations requested by the *Domain Controller*. These functions can be implemented by the Cloud physical nodes that must receive from the *Cloud Controller*, the component that manages the Cloud infrastructure, the list of defined *TVD*s together with the *TVD* membership information at the time the latter requests the creation of a new *VM*.

Communication channel between *EE*s While there are several studies in the literature [Dmi10, CDRS07, BCP+08] that implement a complete solution covering all possible interactions among *VM*s, in this document we will describe the configuration of physical nodes and other network components related to two typical network topologies. In the former case depicted in the Figure 3.2(a), we consider a layer 2 network as it is a realistic option for connecting nodes in the same Cloud data centre. In the latter, we analyze the scenario represented in the Figure 3.2(b) where some nodes are connected at layer 3, as it can happen if a customer instantiates its *VM*s in different locations of the same Cloud.



(a) L2 network on a cloud datacenter

(b) L3 network on different cloud datacenters

Figure 3.2: Typical Cloud network topologies

The *VLAN* technology, specified in the implementation choice, allows to separate the communication of different *TVD*s through L2-adjacent physical nodes by associating them to a different *VLAN* tag. This way, a *VM* believes to be connected to an isolated network composed only by members of the same *TVD*, while the underlying infrastructure is responsible to deliver only data with the associated tag.

When two *TVD* members are running on physical nodes that are not L2 adjacent (assuming we want to achieve L2 connectivity), the infrastructure must create an IP tunnel (e.g with the Generic Routing Encapsulation protocol - GRE) on which the tagged Ethernet frames are encapsulated. In this case, the traffic exchanged between the two nodes must be protected by an *IPsec* connection to ensure confidentiality and data source authentication.

The configuration of physical nodes must be set so that they can properly label Ethernet frames coming from *VM*s, with the tag associated to the *TVD* they belong to, and to unlabel incoming data before delivering them to the correct *VM*s. The advantages of performing these operations at physical node level are that there is no need for additional configuration in the *VM*s and that, even if a *VM* gets compromised, it cannot attack other *TVD*s by sending Ethernet frames with a different tag.

In order to perform the above operation, a physical node, as already mentioned for the management of *EE*s, must have the list of defined *TVD*s and, in addition, must know the VLAN tag associated to each *TVD*. Then, it can properly configure the network interface of *VM*s according to the TVD membership information provided by the *Cloud Controller*. Finally, if an IP tunnel is required for the communication among *TVD* members, it receives from the *Cloud Controller* the related configuration, such as the remote IP address and the parameters for the *IPsec* connection.

In the end, we want to mention that also network components of the Cloud network must be properly configured. In particular, the switches that connect physical nodes must support *VLAN*s and their ports must be configured as trunk.

***TVD* management** The *Cloud Controller* is the central component of a Cloud infrastructure as it executes operations requested by customers (i.e. the creation of new instances or the allocation of virtual storage) and performs the management of physical nodes and network components. This makes the *Cloud Controller* the right candidate also to manage *TVD*s.

This component must hold the overall configuration for defined *TVD*s (e.g. the *VLAN* tags) and must distribute it to the physical nodes. Through its interface it should allow customers to define which *TVD* a new *VM* will be member of, information that will be transmitted to the physical node designed to run the instance.

Another critical task of the *TVD* management component is to determine whether all members of a *TVD* are L2 adjacent to each other. Otherwise, it must generate the configuration for creating a secure tunnel and send it to the involved physical nodes.

Finally, it is responsibility of this component to define for which *TVD*s a physical node is allowed to run new instances. In particular, the benefit of this restriction is that it could be possible to deploy mission-critical *TVD*s only on those physical nodes that meet specific security requirements. To achieve this goal, the *TVD* management component must define a set of filtering rules that physical nodes and network components will enforce so that they are protected from a compromised host that sends packets to *TVD*s for which it is not authorized. More details about these filtering rules will be explained in the Section 3.2.4

3.2.4 Describing *TVD*s using *libvirt*

In the *TVD* model one important aspect covered is the definition of the APIs. More in detail, APIs must be defined for *EE*s management, communication and code execution and for the *software container* management. In particular the latter API must allow the *Domain Controller* to perform *TVD* management operations like attestation, monitoring and member registration.

The model of a generic Cloud infrastructure is very similar in that there is one *Cloud Controller* that perform operations requested by customers and management tasks by invoking an API defined on physical nodes to execute virtualization-specific functions. In common Cloud software (i.e. OpenStack[4] and OpenNebula[5]) this API is currently implemented by *libvirt* [6].

This software allows to perform virtualization functions on a physical node regardless of the specific hypervisor in execution and to model virtualized resources using XML standard language, which can be flexible enough to describe the *TVD* configuration.

In this section we will verify whether requirements stated in the Section 3.2.3 can be satisfied using *libvirt*. In particular, we will analyze in the following whether elements already

[4]OpenStack - http://www.openstack.com
[5]OpenNebula - http://www.opennebula.org
[6]Libvirt - http://libvirt.org

defined in the language can be used to describe the *TVD* configuration and we will propose new items to cover the missing parts. Since this work is focused on defining a data model to be used by the Cloud infrastructure to properly implement *TVD*s, we assume that the *Cloud Controller* can determine, depending on the network topology, the defined *TVD*s and others security requirements, the correct configuration parameters to ensure the containment property for running *VM*s.

At the end of this section, an example of a global TVD configuration will be depicted in the Figure 3.3. This example includes both the already existent *libvirt* configuration elements and the new ones we are proposing.

3.2.4.1 State of the art

Since version 0.9.4 *libvirt* introduces in the virtual network definition a new element called portgroup. Using portgroups, administrators are capable to put *VM* connections to a virtual network into different classes, with each class potentially having different level or type of service.

As specified in the official documentation, for each network multiple portgroups can be defined and one of them, optionally, can be set as the default portgroup. Each class of connection, a portgroup, is identified by the name attribute and the level or the type of service that it represents is expressed using the subelements `<virtualport>` and `<bandwidth>`. The former represents a port of IEEE 802.1Qbh capable bridge where the network interface of an instantiated *VM* is plugged, the latter enables administrators to set Quality of Service parameters. In the `<virtualport>` element an administrator may set some attributes of the subelement `<parameters>` in order to associate it to a certain *VLAN*.

In the *VM* description, a virtual network interface is defined by using the element `<interface>` with the attribute `type` set to the value `network`. A network interface may be associated to a certain portgroup by specifying its name as attribute in the subelement `<source>`. This way, if the portgroup is associated to a *VLAN* through the `<virtualport>` subelement, the domain can be considered member of this *VLAN*.

```
# Network definition which includes two portgroups definition.

<network>
  <name>network_alpha</name>
  ...
  <portgroup name="alpha" default="yes">
    <virtualport type="802.1Qbh">
      <parameters profileid="vlan_alpha"/>
    </virtualport>
  </portgroup>
  <portgroup name="beta">
    <virtualport type="802.1Qbh">
      <parameters profileid="vlan_beta"/>
    </virtualport>
  </portgroup>
  ...
</network>


# Domain definition which includes a network interface associated
# to a portgroup alpha
<domain>
  <interface type="network">
    <source network="network_alpha" portgroup="vlan_alpa">
```

```
      ...
  </interface>
</domain>
```

3.2.4.2 Missing features

New forwarding mode In order to make the frame tagging process transparent for the virtual domain, we propose to delegate this procedure to the host bridge. This approach may be considered as a new forward mode called *vlanbridge* which can be specified in the virtual network definition. To apply this new mode, the presence of a virtual switch 802.1Q capable is required. A possible solution may be the substitution of the standard Linux bridge driver with that provided by Open vSwitch[7].

```
<network>
    <name>mynet</name>
    <forward mode="vlanbridge">
        <interface dev="eth0" />
        <tunnel name="mytun" />
    </forward>
</network>
```

Tunnel element definition In routed networks the Ethernet frames could be encapsulated in L3 packets to create a L2 adjacency between endpoints. The encapsulation can be executed by establishing a tunnel between endpoints of a communication. In *libvirt* a configuration element capable to describe a tunnel is not available. We propose to define an element called `<tunnel>` to get *libvirt* able to manage the encapsulation by establishing a tunnel. Below is depicted an example of the possible definition of a GRE tunnel.

```
<tunnel type="gre">
  <name>mytun</name>
  <uuid>0adcaee0-6aba-402e-88fd-5d05384a0515</uuid>
  <device>gre0</device>
  <remoteip>130.192.12.88</remoteip>
  ...
</tunnel>
```

As proposed, the tunnel element does not include any security feature. Considering the definition of the *TVD* channel using the *libvirt* configuration language, the tunnel element could be a good place where to put security features like an *IPsec* profile.

IPSec profile definition Following the guidelines of already existent configuration elements in *libvirt*, an *IPsec* profile could be defined as an independent element which can be optionally attached to another one, for example to a tunnel. A first draft of the `<ipsecprofile>` element structure includes subelements required for external references (`name` and `uuid`) and subelements to describe Security Associations and Security Policies.

```
<ipsecprofile>
  <name>myipsecprofile</name>
  <uuid>d8e0d396-8e90-47ec-991d-f739e9611442</uuid>
```

[7] Open vSwitch - http://www.openvswitch.org

```
  <sa>...</sa>
  <sp>...</sp>
</ipsecprofile>
```

To apply *IPsec* settings to the tunnel definition, an *IPsec* profile could be specified as subelement of the <tunnel> element.

Filtering capabilities In order to enhance the filtering process, we define an element called <accesslist> which includes an IP address and a list of portgroups. While the IP address identify the remote endpoint of a channel (i.e. the remote host of a tunnel), the portgroup list is a list of enabled *VLAN*s. By adopting the <accesslist> elements it is possible to verify if an Ethernet frame received by a remote host is tagged with a tag corresponding to an allowed *VLAN*. In this way, if an host is compromised the attack effects would be confined to the *TVD*s enabled for the host.

```
<accesslist>
    <name>list_alpha</name>
    <uuid>c167ac3d-6e81-440c-b5b4-e9738969460a</uuid>
    <ip>2.156.16.31</ip>
    <portgroup name="service_alpha" />
    <portgroup name="service_beta" />
    ...
</accesslist>
```

In addition to the enhancement of the security level, <accesslist> may simplify the management of the *TVD* structure. In fact, by applying a modification only to a portgroup list in the <accesslist>, the logical topology of the *TVD* among the Cloud nodes may be changed. Moreover, in vision to introduce the usage of virtual switches (e.g. Open vSwitch), the information collected in <accesslist> elements may be used to generate the configuration to be injected to the host bridge.

3.2.5 Conclusions

The *TVD* model allows to address some of the security issues raised in Cloud environments by defining a new entity for which it is possible to guarantee properties like containment and trust. In this work, instead of proposing a specific implementation of *TVD*s, we focused on modeling the configuration of a generic Cloud infrastructure at high level using *libvirt*, which is already used by common Cloud solutions for the configuration of physical nodes.

From the analysis of this software, we discovered that it already defines some configuration elements to describe the mechanisms chosen for providing the containment property of *TVD*s, such as the portgrougs for *VLAN*s. However, some work must be done to map this description on component-specific instructions for *VLAN*-aware bridges because, actually, only the direct forwarding of Ethernet frames through the physical network interface is supported.

Another area of work is represented by the enhancement of the *Cloud Controller*, which must determine the *libvirt* configuration of physical nodes to ensure the isolation between *TVD*s. We envision a great opportunity of integrating our work with *VALID* and the related validation tool [BGM11] to express the *TVD* isolation as security goal and to verify that the overall infrastructure configuration does not violate this goal.

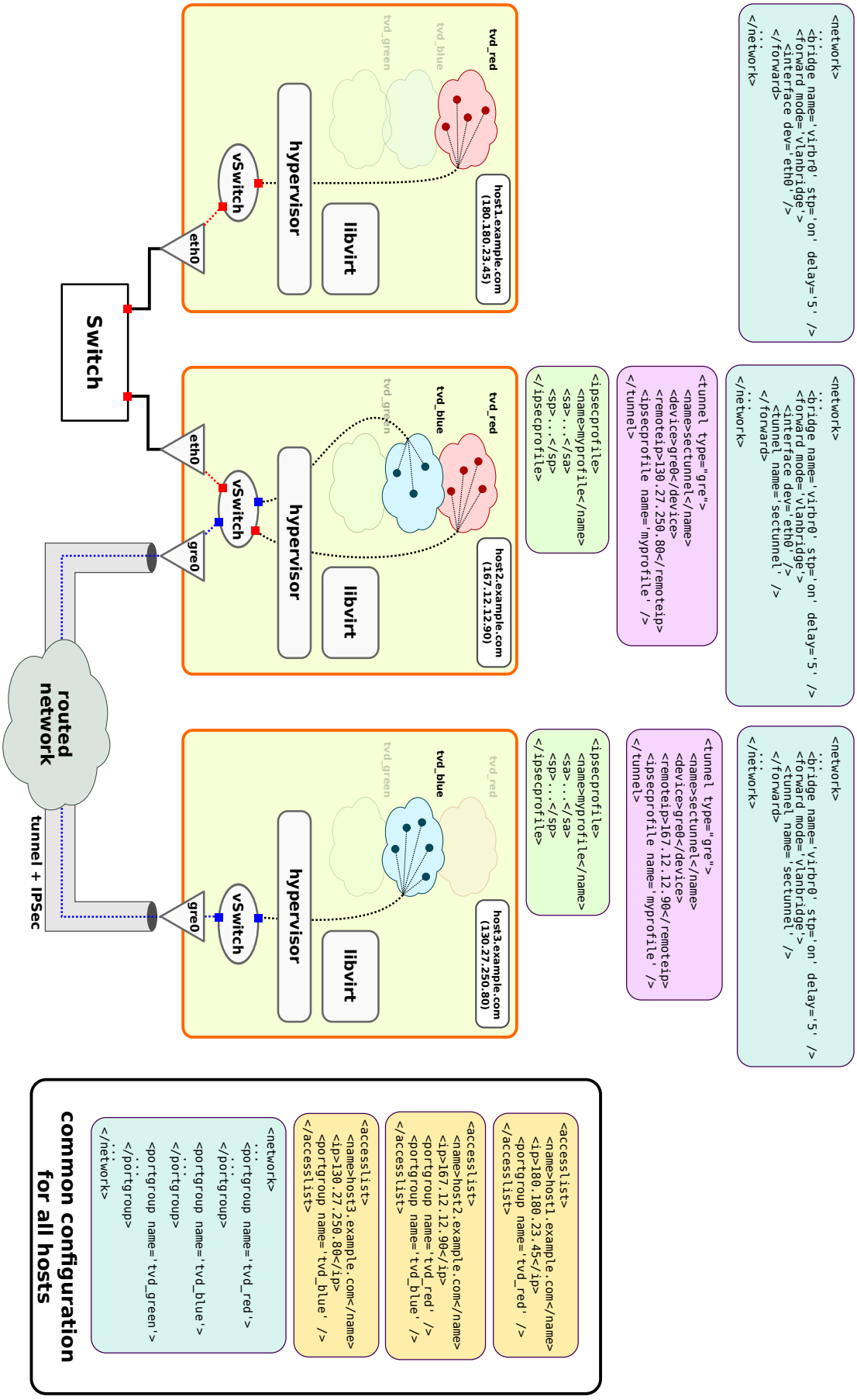


Figure 3.3: TVD configuration in L2/L3 networks

Chapter 4

Automated Verification of Virtualized Infrastructures

Chapter Authors:
Sören Bleikertz, Thomas Groß (IBM)

Virtualized infrastructures and clouds present new challenges for security analysis and formal verification: they are complex environments that continuously change their shape, and that give rise to non-trivial security goals such as isolation and failure resilience requirements. We present a platform that connects declarative and expressive description languages with state-of-the-art verification methods. The languages integrate homogeneously descriptions of virtualized infrastructures, their transformations, their desired goals, and evaluation strategies. The different verification tools range from model checking to theorem proving; this allows us to exploit the complementary strengths of methods, and also to understand how to best represent the analysis problems in different contexts. We consider first the static case where the topology of the virtual infrastructure is fixed and demonstrate that our platform allows for the declarative specification of a large class of properties. Even though tools that are specialized to checking particular properties perform better than our generic approach, we show with a real-world case study that our approach is practically feasible. We finally consider also the dynamic case where the intruder can actively change the topology (by migrating machines). The combination of a complex topology and changes to it by an intruder is a problem that lies beyond the scope of previous analysis tools and to which we can give first positive verification results.

4.1 Introduction

Virtualized infrastructures and clouds form complex and rapidly evolving environments that can be impacted by a variety of security problems. Manual configuration as well as security analysis often capitulate in face of these ever-changing complex systems. The need for automated security assurance analysis is immediate. Given the volatility of virtualized infrastructure configurations as well as the diversity of desired security goals, specialized analysis tools—even though having performance advantages—have limited benefits.

As a general approach, we propose to first specify abstract security goals as *desired state* for a virtualized infrastructure in a formal language. For instance, goals can be in the areas *operational correctness* (e.g., "Are all VMs deployed on their intended clusters?"), *failure resilience* (e.g., "Does the infrastructure provide enough redundancy for critical components?") or *isolation* (e.g., "Are VMs of different security zones isolated from each other?"). Second, we employ a generic analysis tool to evaluate the *actual state*, i.e., the virtualized infrastructure

configuration, against this desired state. Thus, we obtain an automated analysis mechanism that can check the configuration—and configuration changes—against a high-level security policy.

Such an automated analysis can cover two scopes: in the *static* case, we analyze a single state of a virtualized infrastructure against the desired properties. In the *dynamic* case, we consider the actual configuration as a start state and consider transitions that can change this configuration. In our example, we consider particularly changes that an intruder can make to the network (within the limits of his access rights), e.g., by migrating VMs to other security zones. The question is whether we can reach an attack state in this way, i.e., a current configuration of the system that violates the required security properties. The dynamic case is a generalization of the static case that can only be handled by the model-checking tools.

From engagements with customers running large-scale virtualized infrastructures, we learned that they are interested in a broad range of security goals. Specialized tools can be applied to a subset of these security goals, as we already demonstrated in previous research (cf. [BGSE11]) for security zone isolation. However, a general approach is desired that can cover this broad range of security requirements.

Our goal is to establish general-purpose verification methods as an automated tool for security assurance of virtualized infrastructures. We present a platform that connects declarative and expressive description languages with state-of-the-art verification methods. With such a platform, we can benefit from the variety of existing methods and recent advances such as those in the field of SMT solving. As desired state specification, we take security assurance goals in the formal language *VALID* (cf. Chapter 3.1) as inputs. As actual state, we lift the configuration of a heterogeneous virtualized infrastructure to a unified graph model. For this, we employ a security assurance analysis tool called SAVE [BGSE11], which also computes graph coloring overlays, that model, e.g., information flow. We develop a translator that connects these descriptions with the various state-of-the-art verification tools. The translation involves adapting the verification problem to the domain of the respective tool, and property-preserving simplifications and abstractions to support the verification. In particular, the translation does not add false positives or false negatives to the model.

In this work we demonstrate that model-checking of cloud infrastructures is in general possible, and we exemplify our approach by studying three examples: *zone isolation*, *secure migration*, and *absence of single point of failure* on the network level. The first example is a static case, which asks whether machines from different security zones are somehow connected in an information flow graph. The relevancy of this case was confirmed in a case study with a financial institution. The second example is of dynamic nature, and asks whether an intruder with rights to migrate VMs can reach an attack state, either by migrating the machine through an insecure network (thereby modifying the VM) or to a physical machine he controls. Secure migration as an example is used to show our first result in verifying dynamic problems, which are in our future interest. The last example belongs to the static case, and we consider that between certain machines we must have redundant network links. Studying a broader range of scenarios using our proposed general approach is left as future work.

4.1.1 Contributions

We are the first to apply general-purpose model-checking for the analysis of general security properties of virtualized infrastructures. We propose the first analysis machinery that can check the actual state of arbitrary heterogeneous infrastructure clouds against abstract security goals specified in a formal language. Our approach covers static analysis as well as dynamic analysis and uses a versatile portfolio of problem solver back-ends to benefit from their different solution

strategies (fix-point evaluation, resolution, etc.).

We believe that our experiments with different analysis strategies (Horn clauses, transition rules) are of independent interest, because the problem instances for security assurance of virtualized infrastructures are structured differently than traditional application domains of model checkers, notably security protocols. In addition, we gained some insights on the complexity relations of different problem classes.

As a case study, we successfully model checked a sizable[1] production infrastructure of a global financial institution against the zone isolation goal. We have previously analyzed this infrastructure extensively with specialized tools and found the same problems with this generic approach. We report that our different optimizations allowed us to improve the performance by several orders of magnitude: whereas the unoptimized problem instances did not terminate within several hours, the optimized problem instances completed the analysis in the order of seconds.

This work build upon the following results: Bleikertz et al. [BGSE11] introduced an analysis system for virtualized infrastructures, called *SAVE*, which models configurations in a graph representation and runs graph-coloring based information flow analysis on this representation. Bleikertz and Groß introduced a domain-specific language, called *VALID* (cf. Chapter 3.1), which allows us to specify security goals for virtualized infrastructures in formal terms. This work makes distinct new contributions by introducing analysis based on general-purpose model checking on SAVE's graph representation against VALID specifications. The system presented in this work goes far beyond information flow analysis of [BGSE11] by enabling validation of a wide range of security goals.

4.1.2 Architecture

We aim at the evaluation of an actual state against a desired state, for which we employ a tool architecture illustrated in Figure 4.1. To specify a *desired state*, we formulate general security goals in *VALID* [BG11], a language for security assurance of virtualized infrastructures.

To obtain the *actual state* of a virtualized infrastructure, we employ a tool for *assurance analysis of virtualized infrastructures*, called SAVE [BGSE11]. It comes with discovery probes for heterogeneous clouds such as VMware, Xen, pHyp, etc. and takes their proprietary configuration data as inputs. It lifts the configuration data to a unified graph representation of the virtualized infrastructure (the *realization model*) and computes transitive closures over a graph coloring model for information flow tracing. SAVE outputs its graph representations as actual state basis of our analysis.

We use and compare several state-of-the-art tools for automated verification. The first is the AVANTSSAR platform[2]; it consists of three verification backends, OFMC [BMV05b], CL-Atse [Tur06], and SAT-MC [AC08], that all have the common input language ASLan (AVANTSSAR Specification Language). We have focused here on OFMC and made initial experiments with the other two; due to lack of source code availability and lack of support of Horn clauses in current SAT-MC, we could not run CL-AtSe and SAT-MC on the large scale case study through their web interface. The particular strength of AVANTSSAR is that we can model a dynamic network and check whether a property holds in all reachable states of the network. For the simpler case of analyzing a static network, a broader range of tools is applicable as we can express verification as deducibility problems in first-order Horn clauses. We con-

[1] approximately 1,300 VMs, 25,0000 nodes and 30,000 edges

[2] http://www.avantssar.eu/

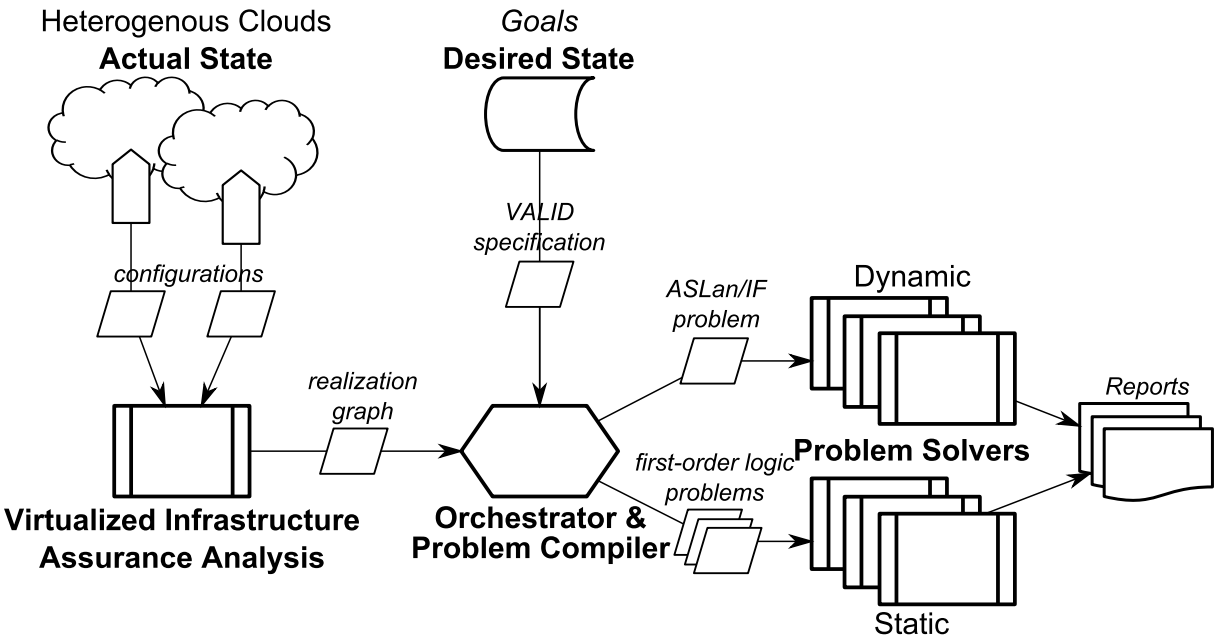


Figure 4.1: Architecture for model checking of general security properties of virtualized infrastructures.

sider here the automatic first-order theorem prover SPASS [WDF+09][3] and the protocol verifier ProVerif [Bla01][4]. We also made initial experiments with the SuccinctSolver [NNS02][5]. In general, our hope is that tools based on different methods can have complementary strengths and connecting them allows us to benefit from all advances of the tools.

As the key component for the actual/desired state analysis, we develop a *compiler* that takes the graph representation of the actual state and the desired state specification in *VALID* as inputs and compiles problem instances for the solver back-ends. It refines the graph representation (e.g., by abstracting from nodes that cannot affect the analysis goal or by introducing "fast lanes"), compiles a term algebra from it, and enhances this problem instance with an analysis strategy (such as, Horn clauses or intruder transition rules) and the goal specified in *VALID* [6].

4.2 Information Flow Analysis Tool Preliminaries

We use an analysis tool for information flow analysis of virtualized infrastructures [BGSE11] to discover the actual configuration of a virtualized infrastructure, abstract it into a unified graph model and determine potential information flow by transitive closure over graph coloring. This tool will provide the bases for the term algebra to describe the actual state. It proceeds in the following phases[7]: The first phase of building a graph model is realized using a discovery step that extracts configuration information from heterogeneous virtualized systems, and a translation step that unifies the configuration aspects in one graph model. For the subsequent analysis, we apply the graph coloring algorithm defined in [BGSE11] parametrized by a set of traversal

[3] http://www.spass-prover.org/
[4] http://www.proverif.ens.fr/
[5] http://www.imm.dtu.dk/cs_SuccinctSolver/
[6] The AVANTSSAR tools accept *VALID* goals out of the box, we translate the goals for the other tools
[7] For further details about this process we refer the reader to [BGSE11].

rules and a zone definition. The resulting colored graph model is the actual state input for the compiler to be verified against the desired state security polices.

Discovery The goal of the discovery phase is to retrieve sufficient information about the configuration of the target virtualized infrastructure. For this matter, platform-specific data is obtained through APIs such as VMware VI, XenAPI, or libVirt, and then aggregated in one discovery XML file. It contains information, among others, about the virtual machines, virtual networks, and storage in a platform-specific representation. The target virtualized infrastructure, for which we will discover its configuration, is specified either as a set of individual physical machines and their IP addresses, or as one management host that is responsible for the infrastructure. Additionally, associated API or login credentials need to be specified. For each physical or management host given in the infrastructure specification, we will employ a set of discover probes that are able to gather different aspects of the configuration.

We illustrate the discovery procedure with VMVare as example. Here, the discovery probe connects to *vCenter* to extract all configuration information of the managed resources. It does so by querying the VMware API with the retrieveAllTheManagedObjectReferences() call, which provides a complete iteration of all instances of ExtensibleManagedObject, a base class from which other managed objects are derived. We ensure completeness by fully serializing the entire object iteration into the discovery XML file, including all attributes.

Transformation into a Graph Model We translate the discovered platform-specific configuration into a unified graph representation of the virtualization infrastructure, the *realization model* (cf. [BGSE11] for the formal specification of the graph model). It expresses the detailed configuration of the various virtualization systems and includes the physical machine, virtual machine, storage, and network details as vertices. We generate the realization model by a translation of the platform-specific discovery data. This is done by so-called *mapping rules* that obtain platform-specific configuration data and output elements of our cross-platform realization model. Our tool then stitches these fragments from different probes into a unified model that embodies the fabric of the entire virtualization infrastructure and configuration.

Again, we illustrate this process for a VMware discovery. Each mapping rule embodies knowledge of VMware's ontology of virtualized resources to configuration names, for instance, that VMware calls storage configuration entries storageDevice. We have a mapping rule that maps VMware-specific configuration entries to the unified type and, therefore, establishes a node in the realization model graph. We obtain a complete iteration of elements of these types as graph nodes. The mapping rules also establish the edges in the realization model. In the VMware case, the edges are encoded implicitly by XML hierarchy (for instance, that a VM is part of a physical host) as well as explicitly by Managed Object References (MOR). The mapping rules establish edges in the realization model for all hierarchy-links and for all MOR-links between configuration entries for realization model types.

This approach obtains a complete graph with respect to realization model types. Observe that configuration entries that are not related to realization model types are not represented in the graph. This may introduce false negatives if there exist unknown devices that yield further information flow edges. To test this, we can introduce a default mapping rule to include all unrecognized configuration entries as dummy node and all respective MOR links as edges.

Coloring Through Graph Traversal The graph traversal phase obtains a realization model and a set of information source vertices with their designated colors as input. The graph col-

oring outputs a colored realization model, where a color is added to a node if permitted by an appropriate traversal rule. We apply a first-matching algorithm to select the appropriate traversal rule for a given pair of vertices.

We use the following three type of traversal rules that are stored in a ordered list. *Flow rules* model the knowledge that information can flow from one type of node to another if an edge exists. E.g., a VM can send information onto a connected network. We represent these rules by "follow". *Isolation rules* model the knowledge that certain edges between trusted nodes do not allow information flow. E.g., a trusted firewall is known to isolate, i.e., information does not flow from the network into the firewall. We represent these rules by "stop". *Default rule* Whenever two types are not covered by an isolation or flow rule, then we default to "follow". In order to be on the safe side, we assume that flow is possible along this unknown type of edges. An example of a set of traversal rules can be found in [BGSE11].

Output Actual State As actual state formulation for our subsequent analysis, the tool outputs different kinds of unified graph models of the infrastructure. We call the graph with the topology of the entire infrastructure realization model; as we will see in Section 4.3, this models the graph type real of the desired state specification in *VALID*. In addition, we obtain subgraphs of the topology that are reachable by a color in the information flown analysis; they model the graph type info of the desired state specification.

4.3 Language Preliminaries

We briefly introduce the concepts of the languages *VALID* and ASLan (cf. Chapter 3) which are at the core of our formal models. ASLan stands for *AVANTSSAR Specification Language* [AVA10], a set-rewriting based formalism for specification of infinite state transition systems dedicated to security of distributed systems. ASLan is an extension of the AVISPA Intermediate Format [AVI03]; one of the key extensions of ASLan is the integration of Horn clauses that allow for complex evaluations within every state of the transition system.

The Virtualization Assurance Language for Isolation and Deployment (*VALID*) [BG11] is a formal language building upon ASLan/IF for specifying security assurance goals of virtualized infrastructures. It is a domain-specific extension and customization of ASLan, in particular introducing a typing system tailored to the needs of virtualized infrastructures and graph-based analysis. Being close to ASLan, it is relatively well-suited for the connection with the model-checking tools of the AVANTSSAR platform.

We describe the two languages along-side (as their structure and meaning is very similar) and highlight the differences.

Term Algebra At the core of ASLan and *VALID* is a term algebra over a signature Σ and variable symbols $\mathcal{V}$. In concrete syntax, all constant and function symbols of Σ are alphanumeric identifiers that start with a lower-case letter, while variable symbols of $\mathcal{V}$ start with upper-case letters. We typeset ASLan/*VALID* elements in sans−serif. We interpret terms in the *free algebra*, i.e., syntactically different terms represent different values. In particular, two different constants always represent different entities. We use standard notions of terms such as *ground* (terms without variables), *substitution*, and *matching*.

In ASLan, terms represent usually (cryptographic) messages where constants can be the identifiers of participants, cryptographic keys, etc., and functions represent cryptographic operations like symmetric encryption. In *VALID*, in contrast, the constants denote the elements of

the virtualized infrastructure such as virtual and physical machines, switches, and zones. We rarely deal with composed terms, and use only constants and variables.

***VALID*'s Type System** All constants in *VALID* have a type, according to the following type system:

Definition 18 (Type System) *We have a finite set of* type symbols *(disjoint from variable and constant symbols) that include for this work the following:*

$$\mathbb{T} := \{\mathsf{node}, \mathsf{machine}, \mathsf{host}, \mathsf{network}, \mathsf{zone}\}$$

We also have an acyclic subtype relation between types; here, all the types machine, host, *and* network *are subtypes of type* node.

A valid specification must contain one type declaration *for every used constant symbol, and at most one for every used variable. All variables without type declaration are* untyped. *Compound terms are always untyped. A typed variable can only be matched against a constant of the same type or of a subtype. A type declaration that term t has type τ is denoted by $t : \tau$.*

To analyze topologies, we model virtualized infrastructure configurations as graphs. Whereas the basic graph, called realization, is a unification of vendor-specific elements into abstract nodes, we introduce further graph transformations to model information flow and dependencies.

Definition 19 (Graph Types) *A* graph type
$G \in \{\mathsf{real}, \mathsf{info}, \mathsf{depend}, \mathsf{net}\}$ *is a constant identifier for a type of a graph model:*

- real *denotes a realization graph unification of resources and connections thereof.*
- info *denotes a realization graph augmented with colorings modeling topology information flow.*
- depend *denotes a realization graph augmented with colorings modeling sufficient connections to fulfill a resource's dependencies.*
- net *denotes a realization graph augmented with colorings modeling network topology information flow.*

Facts and States The next layer of ASLan and *VALID* are *facts* (aka *predicates*) expressing relationships between terms. We use in this work the following (untyped) signature of facts symbols (disjoint from constant, variable, and type symbols) with their intuitive meaning:

- $\mathsf{contains}(Z, M)$ where typically Z and M are constant or variable symbols of type zone and machine, respectively. This denotes that machine M belongs to zone Z.
- $\mathsf{edge}([G : \mathsf{real}];\ A, B)$ is a predicate, which denotes the existence of a single edge between A and B with respect to an (optional) graph type G.
- $\mathsf{connected}([G : \mathsf{real}];\ A, B)$ is a predicate, which denotes existence of a path between A and B, respect to an (optional) graph type G.

The notation $[A : v]$ denotes an optional argument A with default constant value v.

There are further predicate symbols used in *VALID* that we do not discuss here for brevity, such as paths used to iterate over all paths, and matches used to relate ideal and real nodes.

A *state* is a finite set of ground facts that hold true in the state (and all other facts are false). We denote states using the an enumeration of facts separated by "." (which technically can be regarded as a commutative, associative, and idempotent operator).

Rules and Goals An ASLan specification consists of an *initial state*, a set of *rules* that give rise to a transition relation, and a set of *goals* that describe a set of states, usually the violations of the security properties.[8] The security analysis shall then determine whether a goal state is reachable from the initial state by using the rules. Moreover, one may add Horn clauses to specify immediate consequences within a single state which we discuss in more detail below.

The rules have the form $PF.NF.C \Rightarrow RF$ where PF and RF are sets of facts, NF is a set of negative facts (denoted using the ASLan operator $\mathsf{not}(\cdot)$), and C is a set of inequalities on terms. The variables of RF must be a subset of the variables of PF. Such a rule is interpreted as follows: we can make a transition from state S to state S' if S contains a match for all "positive" facts of PF, does not contain any instance that can match a negative fact of RF, and the inequalities of C do hold under the given match. (More formally, the variables of PF are thus existentially quantified, and the ones that only occur in NF and C are universally quantified.) The successor state is obtained by removing the matched positive facts of PF and adding the RF under the matching substitution.

For example, the following rule expresses that, if an intruder resides at a node N and there is an edge from N to another node M and M is not contained in a particular zone z, then the intruder can move to M:

$$\mathsf{intruderAt}(\mathsf{N}).\mathsf{edge}(\mathsf{N},\mathsf{M}).\mathsf{not}(\mathsf{contains}(\mathsf{z},\mathsf{M})) \Rightarrow \mathsf{edge}(\mathsf{N},\mathsf{M}).\mathsf{intruderAt}(\mathsf{M})$$

Upon this transition, the fact $\mathsf{intruderAt}(\mathsf{N})$ is deleted (because it is not repeated on the right-hand side); the fact $\mathsf{edge}(\mathsf{N},\mathsf{M})$ remains in the graph because it *is* repeated on the right-hand side.

Goals are quite similar to rules in that have the form $PF.NF.C$ (like a rule without right-hand side) and by the same semantics as rules characterize a set of states, usually "bad" states for state-based safety properties. In *VALID*, goal specifications are also labeled with a graph type G.

Horn Clauses ASLan introduced the specification of Horn clauses to the transition system to allow for specifying immediate consequences within a state. One of the main application is the formalization of access control policies: access rights can be expressed as a direct consequence of other facts that express for instance that an employee is a member of particular group. Horn clauses and the state transition system can mutually interact. First, a transition can change the facts that currently hold (e.g., an employee changes to another group) which has immediate consequences for the access rights via the Horn clauses. Second, the fact representing the (current) access decision can be the condition of another transition rule (where an employee requests access to a resource). In our context, we can also use the Horn clauses to formalize properties of the current graph. E.g., to formalize that $\mathsf{connected}()$ is the symmetric transitive closure of the $\mathsf{edge}()$ predicate we can simply specify:

$$\begin{aligned}&\mathsf{connected}(\mathsf{A},\mathsf{B}) :- \mathsf{edge}(\mathsf{A},\mathsf{B})\\ &\mathsf{connected}(\mathsf{B},\mathsf{A}) :- \mathsf{edge}(\mathsf{A},\mathsf{B})\\ &\mathsf{connected}(\mathsf{A},\mathsf{C}) :- \mathsf{edge}(\mathsf{A},\mathsf{B}).\mathsf{connected}(\mathsf{B},\mathsf{C})\end{aligned}$$

Introducing or removing edges upon transitions would automatically change the $\mathsf{connected}()$ relation.

[8] Alternatively, ASLan allows for specifying goals also as LTL properties, a feature that we do not use in this work, however.

4.4 Problem Classes

During our analysis, we found that the analysis goals for virtualized infrastructures can be structured into orthogonal problem classes, and that different problem classes exhibit consistent complexity tendencies for the solver-backends. We consider problem classes with respect to three (syntactical) criteria on attack states and intruder rules: locality, positivity, and dynamics.

4.4.1 Local vs. Global

Definition 20 (Locality) *We call an attack state* local *if it only exhibits state facts that will be part of the initial state, e.g.,* edge() *and* contains(). *We call an attack state* global *if it exhibits state facts that must be derived by an evaluation over the topology (e.g.,* connected()*). We use these terms for the corresponding problem instances, as well.*

Secure migration—in the sense that the intruder cannot reach a state in which he controls the physical host to which a VM was migrated—is a local problem, because the attack state will be formulated on the edge() statement between these components.[9] Zone isolation mentioned in the introduction is an example of a global problem, because it needs to consider the connections throughout the topology.

All other factors equal, we conjecture that local problems can be consistently checked more efficiently than global problems. We also conjecture a positive performance correlation between Horn clause based models and problem solvers with local problems, and between transition based models and problem solvers with global problems.

4.4.2 Positive vs. Negative Attack States

Attack states formulated in *VALID* can contain positive as well as negative facts.

Definition 21 (Positivity) *We call an attack state* positive *if it exclusively contains positive state facts. We call an attack state* negative *if it contains at least one negative state fact. We use these terms for the corresponding problem instances, as well.*

The secure migration and zone isolation examples are positive problems. A negative attack state is, for instance, the guardian mediation introduced in [BG11], which is fulfilled if there exists any connection between a machine and a network that is *not* mediated by a guardian (firewall).

All other factors equal, we conjecture that positive problems can be checked more efficiently than negative problems.

4.4.3 Static vs. Dynamic

We consider problems that are statically checking whether the actual state fulfills a desired state. By introducing additional transition rules we can allow the intruder to transform the virtualized infrastructure to reach an attack state, and therefore introduce dynamics.

Definition 22 (Dynamics) *We call a problem instance* static *if its transition rules and Horn clauses only include topology traversal over the initial state. We call a problem instance* dynamic *if it contains transition rules or Horn clauses that model intruder capabilities to change the initial state.*

[9] Another example for a local problem is machine placement specified in [BG11], the question whether each VM in the actual state has an edge to the physical host specified in the desired state.

Many example problems presented in [BG11] (machine placement, zone isolation, guardian mediation) are static in first instantiation. As soon as we extend the intruder rules/clauses with rights to, e.g., start, stop or migrate machines or to reconnect networks/storage, we obtain dynamic problems. Secure migration introduced above is a dynamic problem.

All other factors equal, we conjecture that static problems can be checked more efficiently than dynamic problems. In the static case, it is more efficient to check for several attack states than in the dynamic case. We conjecture that first-order logic models and tools will have an advantage at static problems, whereas transition based models and tools will have an advantage at dynamic problems.

4.5 Compiling Problem Instances

This section discusses how we compile problem instances for the solver back-ends, thus, explains the compiler which is a key component of the architecture in Section 4.1.2. The compiler receives the following inputs: the realization model or derivatives as a graph representation of the actual state and a *VALID* policy as representation of the desired state.

The success and efficiency of the solver back-ends are largely determined by the initial size of the problem instance, by solution strategies that limit the search tree complexity, and by problem formulations that match the solvers' capabilities. Therefore, the compiler must strive for a significant complexity reduction while maintaining generality. Because we target sizable real-world infrastructures, the initial problem size may easily be in the order of tens of thousands of nodes and the compiler's pruning prove crucial.

The compiler works in multiple phases:

- *Graph Transformation:* Reducing the complexity of the graph and representing it as term algebra facts.
- *Strategy Amendment:* Introducing sensible analysis strategies into the problem instance that match the solver's strengths'.

4.5.1 Graph Transformation

A (colored) realization model input consists of high-level nodes, such as machine, and low-level nodes, such has ipInterface, as well as edges that model the connections between these components. In general, we aim at representing the edges of this graph as edge() facts in term algebra and give the problem solvers means to derive graph facts, notably connected().

Real-world virtualized infrastructures consist of tens of thousands low-level components and similarly many edges, an initial complexity that could easily overwhelm the solver back-ends. Therefore, we support the solver back-ends in traversing these graphs efficiently by either abstracting from low-level nodes not impacting the analysis or introducing "fast lanes" into the graph:

Definition 23 (Optimization: Graph Refinement) *For all adjacent high-level components, such as* machine *or* host*, connected though a sub-graph with low-level components with degree smaller than three, we replace the subgraph by edges maintaining the same connectivity. Similarly, "fast lanes" added to the graph allow the solver back-ends to reach other segments of the graph with fewer steps.*

The graph refinement maintains analysis generality if the pruned node types do neither occur in attack states nor in intruder or topology transformation rules/clauses.

4.5.2 Strategy Amendment

Graph Traversal A major part of the solver's strategy will depend on how the graph traversal is modeled, which we express by connected() facts derived from the edge() facts:

$$\mathsf{edge}(\mathsf{A},\mathsf{B}) \Rightarrow \mathsf{edge}(\mathsf{A},\mathsf{B}).\mathsf{connected}(\mathsf{A},\mathsf{B}).\mathsf{connected}(\mathsf{B},\mathsf{A})$$

$$\mathsf{edge}(\mathsf{A},\mathsf{B}).\mathsf{connected}(\mathsf{B},\mathsf{C}) \Rightarrow \mathsf{edge}(\mathsf{A},\mathsf{B}).\mathsf{connected}(\mathsf{B},\mathsf{C}).\mathsf{connected}(\mathsf{A},\mathsf{C})$$

Observe that this formulation to compute the connected() relation does not change the graph, i.e., edge() facts are neither introduced or removed by these rules. While this is a necessity for all evaluations of the graph in the dynamic case, in the static case, we can formalize evaluation procedures that *do* change the graph, for instance rules that remove edges from the graph as soon as they were visited by the evaluation. Our benchmarks show that such changes can improve the performance of our zone isolation example, however only slightly. Moreover, such "graph-consuming" strategies are helpful for formalizing some advanced properties below.

We propose an additional translation, which reduces the state complexity significantly. In this case, we imagine an intruder tries to traverse the topology from some start-point and "obtain" nodes he has access to. This avoids the binary fact connected and instead uses a unary fact intruderHas to represent all members of the largest connected subgraph that contains the intruder start point.

The transition rules are as follows:

$$\mathsf{intruderHas}(\mathsf{A}).\mathsf{edge}(\mathsf{A},\mathsf{B}).\mathsf{not}(\mathsf{intruderHas}(\mathsf{B})) \Rightarrow \mathsf{intruderHas}(\mathsf{A}).\mathsf{intruderHas}(\mathsf{B}).\mathsf{edge}(\mathsf{A},\mathsf{B})$$

$$\mathsf{intruderHas}(\mathsf{A}).\mathsf{edge}(\mathsf{B},\mathsf{A}).\mathsf{not}(\mathsf{intruderHas}(\mathsf{B})) \Rightarrow \mathsf{intruderHas}(\mathsf{A}).\mathsf{intruderHas}(\mathsf{B}).\mathsf{edge}(\mathsf{B},\mathsf{A})$$

For large graphs, the restriction of analyzing such chunks rather than the full connected-relation means substantial savings: roughly speaking, the number of derivable facts is in the worst case linear for the intruderHas strategy, while the number of connected facts is quadratic. This optimization requires, however, that we have to select one start point for the intruderHas() computation and thus get the verification of isolation from other zones only for that selected start point. In case a connected(A, B) fact is used in a security goal, we can translate it to intruderHas(A).intruderHas(B) for certain goals (e.g. zone isolation).

Depending on the used solver or back-end, the evaluation which nodes the intruder can obtain can either be expressed by a means of transition rules (as above) or as first-order Horn clauses (omitting the not(intruderHas(B)) condition on the left-hand sides).

Dynamic Problems In addition to the graph analysis model, we need to introduce intruder rules for the dynamic analysis to model his capabilities to modify the infrastructure. They are highly dependent on the scenario, but can easily be modeled by introducing new facts as well as transition rules or Horn clauses. We exemplify this by modeling the secure migration problem in Section 4.6.

Encoding Static Problems into FOL In case of static problems, such as zone isolation, we do not need to consider transition systems but can rather encode the problem into "static" formalisms like first-order logic (FOL) and alternation-free least fixed point logic (ALFP) for which mature tools exists. We now show that we can effectively use such tools as an alternative to the model-checking approach in the static case. We study the use of the SuccintSolver

for (ALFP) [NNS02], the FOL theorem prover SPASS [WDF+09] and the protocol verifier ProVerif [Bla01].

The example of zone isolation can be expressed as an initial set of facts representing the graph structure, a set of Horn clauses expressing the graph traversal as shown previously, and a predicate that an intruder can reach a machine in another different security zone.

The SuccintSolver [NNS02] is an effective tool for computing the least fixedpoint (i.e., all facts that are derivable by the ALFP clauses from the given facts) of an ALFP specification. (This fixedpoint is in our case always finite.)

The next tool we use is the generic first-order theorem prover SPASS [WDF+09] which is based on resolution. The problem is here that we want a Herbrand model of the symbols (e.g., different constants always represent different elements) which cannot be enforced directly. We thus formulate as a proof goal that an isolation breach can be reached; such a proof exists if and only if this is true in the Herbrand model. For the inequality of zones, we need to specify this as axioms for zones to properly handle the negation w.r.t. the "Herbrand-trick".

We finally consider the ProVerif tool [Bla01] which is also based on resolution but dedicated to security problems formulated by Horn clauses, and therefore often faster than SPASS. Like in SPASS, we have to axiomatically introduce here the inequality of zones, albeit using an uninterpreted predicate symbol (because negation is not possible in ProVerif).

Static Problems beyond FOL We now discuss static problems that are beyond the expressiveness of FOL. Consider the goal of the absence of single point of failures for network links, i.e., that a network contains sufficient redundancies, so that failure of a single node does not disrupt communication.[10] More formally, let us consider a network and the dependability constraint $depend(n_1, n_2)$ between two nodes n_1 and n_2. Then we require that there is are at least two disjoint paths (using disjoint nodes) in the network from n_1 to n_2. Even in the static case (when the network topology cannot change) this problem is beyond the expressiveness of first-order logic (as a consequence of the Löwenheim-Skolem theorem, see e.g. [HR04]).

As a consequence, we cannot use the solvers SPASS, ProVerif, and Succinct Solver. Also, the standard approach to specify the security property as a set of *VALID* or ASLan goals (even using Horn clauses to evaluate the graph) is not applicable, because that would also be FOL expressible relations. However, we can specify a transition system in ASLan to express a game that has a solution (expressed as a set of goal states) if and only if there exists no single point of failure.

We demonstrate this game for the absence of single point of failure in Section 4.6.

4.6 Model-Checking a Virtualized Infrastructure

In this section, we study three example problems, namely zone isolation, secure migration, and absence of single point of failure, and demonstrate how these problems can be analyzed using model-checking. We apply model-checking on small infrastructure examples to demonstrate the approach, and we will analyze a large-scale infrastructure with regard to zone isolation in Section 4.7.

We structure this section analogously to the architecture Section 4.1.2 where for each example problem, we first specify the desired state in *VALID* or ASLan goals along with the required

[10] [BG11] considers another goal for single point of failure that can be expressed as a goal state: when a node depends on a particular resource, then it is connected to more than one node to provide that resource.

language primitives. Second, we introduce the actual state, that is the infrastructure examples we analyze. Third, we discuss specialties of compiling the corresponding problem instance, the problem solvers employed and their output for the analysis.

4.6.1 Zone Isolation

We consider the following scenario to illustrate the zone isolation security goal: an enterprise network consists of three security zones, namely a *high* security zone containing confidential information, a *base* security zone for regular IT infrastructure, and a *test* security zone. Any machine in one zone should not be able to communicate with a machine from a different zone, and network isolation is realized using VLANs.

Desired State To have the solvers check violations of zone isolation, we define an attack state isolation_breach, which asks the question whether any two machines of any two different security zones are connected.

Definition 24 (Goal: Zone Isolation) *The isolation breach attack state matches if any two disjoint zones* ZA *and* ZB *contain machines* MA *and* MB *respectively, and in which there exists an information flow path between these two machines. It is determined as information flow goal by the graph type* info.

```
goal isolation_breach(info;ZA,ZB,MA,MB):=
  contains(ZA,MA).contains(ZB,MB).
    connected(MA,MB) & not(equal(ZA,ZB))
```

Furthermore, the *VALID* policy requires a specification of the membership of machines to specific zones. For example, contains(high, vm1) denotes that *vm1* is part of the *high* security zone.

Actual State As discussed in Section 4.2, SAVE discovers the given infrastructure and captures all low-level configuration details and resource associations. SAVE performs an information flow analysis with the different security zones as information sources and produces an information flow graph for the infrastructure.

Model-Checking Based on the actual state provided by SAVE, our compiler will generate a representation of the (potentially refined) information flow graph in edge() facts and node constants. Since we are dealing with a static problem, we use the efficient intruderHas modeling for graph traversal, and transform the goal accordingly. The output is ASLan for OFMC and a variety of first-order logic languages used by the static problem solvers.

Suppose the VLAN identifier of a machine *vm2* in the *test* zone was misconfigured and is identical to the VLAN ID of a machine *vm1* from the *high* security zone. OFMC will provide us with such an attack state (reduced for brevity) indicating a zone isolation breach:

```
SUMMARY
  UNSAFE
PROTOCOL
  zone_isolation.if
GOAL
  isolation_breach

% contains(zone(high),node(machine(vm1)))
% contains(zone(test),node(machine(vm2)))
```

```
% intruderHas(node(machine(vm1)),i)
% intruderHas(node(machine(vm2)),i)
```

4.6.2 Secure Migration

Secure migration is a problem often encountered in practice which was also highlighted by Oberheide et al. [OCJ08]. Secure Migration is an interesting problem as its very nature requires a dynamic modeling. However, we do not claim to solve it completely with this work, as is a complex endeavor in which many factors (network and storage connections, VLAN associations, correct configuration of VMs, machine contracts, etc.) need to be considered. Still, we want to demonstrate the principles of dynamic analysis with a simplified example of this problem class. We leave a full-scale analysis of secure migration of a production system for future work.

We consider the topology depicted in Figure 4.2 for our scenario: five hosts, where one is controlled by a malicious administrator, are connected to two networks. The malicious administrator can migrate virtual machines between hosts as indicated by the *migrate* edges. There is one VM running on host *HostA*.

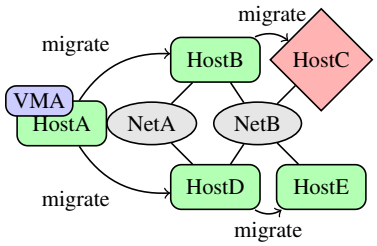


Figure 4.2: Migration Scenario Topology

Desired State We study two exemplary instantiations of the problem of secure migration. The attack state vm_breach asks whether the intruder can migrate a virtual machine from a secure environment to a physical host to which he has root access (in order to perform attacks demonstrated by Rocha et al. [RC11]). The attack state insecure_migration asks whether an intruder can migrate a VM through an insecure network in order to manipulate the VM (cf. attacks demonstrated by Oberheide et al. [OCJ08]).

We define these goals in *VALID*, for which we introduce the unary facts intruderAccess() and root(), and the binary fact migrate(). These model the intruder's access capability set of root access (typically to a given host) and machine migration between two hosts. These facts have the following signature:

$$\begin{aligned} &\mathsf{intruderAccess} : \mathsf{fact} \rightarrow \mathsf{fact} \\ &\mathsf{migrate} : \mathsf{host} * \mathsf{host} \rightarrow \mathsf{fact} \\ &\mathsf{root} : \mathsf{node} \rightarrow \mathsf{fact} \end{aligned}$$

The fact intruderAccess() models the set of all access rights the intruder has, that is, it has the semantic that any term enclosed by the fact belongs to the intruder's access capabilities. The fact root() models administrator rights on the enclosed node.

We model virtual machine migration in the following way.

Definition 25 (Migration) *The capability of migrating a VM* MA *from host* HA *to* HB *is expressed as Horn clause* canMig *that incorporates the intruder access to migrate between these two hosts, that both hosts are connected to the same network* NA*, and one host is running the VM.*

Migration is a transition rule that removes the association of a VM MA *to a host* HA*, and adds an association to a new host* HB *in case fact* canMig *matches.*

$$\mathsf{canMig}(\mathsf{MA},\mathsf{HA},\mathsf{HB},\mathsf{NA}) :- \mathsf{edge}(\mathsf{HA},\mathsf{MA}).\mathsf{edge}(\mathsf{HA},\mathsf{NA}).\mathsf{edge}(\mathsf{HB},\mathsf{NA}).\mathsf{intruderAccess}(\mathsf{migrate}(\mathsf{HA},\mathsf{HB}))$$

$$\mathsf{edge}(\mathsf{MA},\mathsf{HA}).\mathsf{canMig}(\mathsf{MA},\mathsf{HA},\mathsf{HB}) \Rightarrow \mathsf{edge}(\mathsf{MA},\mathsf{HB})$$

The goals are defined in *VALID* in the following way:

Definition 26 (Goal: VM Security) *The VM breach attack state matches if there is a* root() *fact on a host* HA *in the intruder's access capability set and a VM* MA *being connected to the host.*

```
goal vm_breach(real;HA,MA):=
  intruderAccess(root(HA)).
  edge(MA, HA)
```

Definition 27 (Goal: Secure Migration) *The attack state for insecure migration is the following. The intruder can migrate a VM* MA *from host* HA *to* HB*, and he has* root *access to a host* HC *that is connected to the same network.*

```
goal insecure_migration(net;HA,HB,HC,MA,NA):=
  canMig(MA, HA, HB).
  intruderAccess(root(HC)).
  edge(HA, NA).edge(HB, NA).edge(HC, NA)
```

Actual State We model the access capabilities of the intruder for our scenario in the following way.

- intruderAccess(root(hostC))
- intruderAccess(migrate(hostA, hostB))
- intruderAccess(migrate(hostA, hostD))
- intruderAccess(migrate(hostB, hostC))
- intruderAccess(migrate(hostD, hostE))

The network information flow graph for the scenario is generated by SAVE.

Model-Checking Unlike in the previous static example, we had to explicitly model the dynamic behavior of the intruder, i.e., machine migration, and its effects on the infrastructure. We modeled that as transition rules with restrictions based on access privileges of the intruder. Since we are dealing with a dynamic problem, we have to use a tool from the AVANTSSAR tool chain, for instance OFMC.

OFMC found the following attack states (reduced for brevity) for our scenario.

```
INPUT
   migration.if
SUMMARY
   ATTACK_FOUND
GOAL: vm_breach

% Reached State:
%
% intruderAccess(root(node(host(hostC))),i)
% edge(node(machine(vma)).node(host(hostC)),i)
```

OFMC finds this attack state for *vm_breach* due to the migration of *VMA* to *HostB*, and then to *HostC*.

```
INPUT
   migration.if
SUMMARY
   ATTACK_FOUND
GOAL: insecure_migration

% Reached State:
%
% canMig(node(machine(vma)).node(host(hostD)).node(host(hostE)),i)
% intruderAccess(root(node(host(mc))),i)
% edge(node(host(hostD)).node(network(netB)),i)
% edge(node(host(hostE)).node(network(netB)),i)
% edge(node(host(hostC)).node(network(netB)),i)
```

This attack state for *insecure_migration* is reached by the migration of *VMA* to *HostD*, then to *HostE* and intercepted by *HostC* due to the connection to the same network *NetB*.

4.6.3 Absence of Single Point of Failure

We consider the topology illustrated in Figure 4.3 for our scenario to demonstrate the absence of single points of failure for network links. We have two hosts that are depended on each other and connected through a combination of three networks.

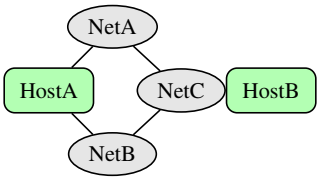


Figure 4.3: Single Point of Failure Scenario Topology

Desired State The goal of the absence of single point of failure for network links is not expressible in FOL, or *VALID* or ASLan goals. Therefore, we construct a game using transitions in ASLan that has a solution if and only if there exists no single point of failure.

This game works as follows for a single dependency constraint $depend(n_1, n_2)$ (if there are several such constraints, one must start each as a separate game). We have two phases in which sets S_1 and S_2 of nodes are collected. In the first phase we start with $S_1 = \{n_1\}$ and non-deterministically follow edges from a member of S_1 to a non-member that we then add, until we have reached n_2 and start the second phase. We begin similarly with $S_2 = \{n_1\}$ and non-deterministically follow an edge from a member of S_2 to a node that is not part of *either* S_1 and S_2 that we add to S_2 until we have reached n_2. Then S_1 and S_2 represent nodes for two

disjoint (except for start and end nodes) paths from n_1 to n_2. Since the transition system allows to non-deterministically choose the edge to follow, the goal state $n_2 \in S_2$ is reachable if and only if such disjoint paths exist.

In the following are the transition rules modeling this game. The first one starts the first phase, the second one traverses nodes in the first phase, and the third one terminates the first phase and starts the second phase. The fourth rule traverses nodes in the second phase.

$$\text{not}(\text{round1}).\text{not}(\text{round2}).\text{depend}(A, B) \Rightarrow \text{round1}.\text{depend}(A, B).\text{inS1}(A)$$

$$\text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{edge}(X, Y).\text{not}(\text{inS1}(Y)).\text{not}(\text{equal}(Y, B)) \Rightarrow \text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{inS1}(Y)$$

$$\text{round1}.\text{depend}(A, B).\text{inS1}(X).\text{edge}(X, B) \Rightarrow \text{round2}.\text{depend}(A, B).\text{inS1}(X).\text{inS1}(B).\text{inS2}(A)$$

$$\text{round2}.\text{depend}(A, B).\text{inS2}(X).\text{edge}(X, Y).\text{not}(\text{inS1}(Y)).\text{not}(\text{inS2}(Y)).\text{not}(\text{equal}(Y, B)) \Rightarrow \text{round2}.\text{depend}(A, B).\text{inS2}(X).\text{inS2}(Y)$$

Here we use special facts round1 and round2 to separate the different phases and inS1 and inS2 to denote the members of S_1 and S_2.

The following goal is reached when the second phase terminates, and thereby identified a second disjoint path between A and B.

```
section goals:
    attack_state spof_absence (A,B,X) :=
      round2.depend(A,B).inS2(X).edge(X,B)
```

Edge symmetry is not handled by the previously shown transitions and the goal, and has to be modeled explicitly with another set of transitions and a goal.

For our scenario, we also have to specify the dependency between *HostA* and *HostB* using the depend term.

Actual State The network information flow graph for the scenario is generated by SAVE.

Model-Checking Since we are dealing with a static problem that cannot be encoded in first-order logic, we modeled this goal in such a way that an attack state is actually a satisfaction of the goal, namely there are no single point of failures. This is contrary to the previous two examples, where an attack state always denoted a breach of a security goal.

For our scenario, the model-checker OFMC will not reach an "attack state", therefore the infrastructure contains a single point of failure. Now we consider connecting *HostB* also to *NetB*, therefore we get a second disjoint path from *HostA* to *HostB*. OFMC produces the following output showing the two disjoint paths (reduced to inS1 and inS2 facts, and re-ordered):

```
INPUT
  spof.if
SUMMARY
  ATTACK_FOUND
GOAL: spof_absence

% Reached State:
```

```
%
% inS1(node(host(hostA)),i)
% inS1(node(network(netB)),i)
% inS1(node(host(hostB)),i)
% inS2(node(host(hostA)),i)
% inS2(node(network(netA)),i)
% inS2(node(network(netC)),i)
```

4.7 Case Study for Zone Isolation

In this section, we analyze a real and large-scale production environment of a global financial institution. The infrastructure consists of approximately 1,300 VMs and its realization model modeling all networking and storage resources consists of approximately 25,000 nodes and 30,000 edges. The infrastructure is divided into several security zones, each containing multiple clusters, and models networking up to Layer 2 separation on VLANs and storage providers up to separation on file level. We have already analyzed this virtualized infrastructure extensively with specialized tools and know which attack states to expect. Given the large initial size of the actual state, this case study provides a suitable test environment for the subsequent performance analysis.

Whereas our compiler translates the problem instances to the different static and dynamic problem solvers introduced in Section 4.1.2, we focus the performance evaluation on three tools SPASS and ProVerif for the static case and OFMC for both the static and dynamic case. We have also performed initial experiments with SAT-MC, CL-AtSe, and SuccintSolver, but could not apply them to the large case study. We analyze various optimization and modeling techniques introduced in Section 4.5 to establish their effects in practice. We are focusing in this evaluation on two specific clusters (we call them *Cluster1* and *Cluster2*) and their corresponding information flow graphs, for which we know that *Cluster1* has an isolation problem and *Cluster2* is safe.

Graph Refinement We first measure the simplification of the information flow graphs for the different clusters in terms of the number of edges and nodes. The information flow graph of *Cluster1* consists of 14386 nodes and 17817 edges. We achieve a reduction of the graph by 13428 nodes and 16860 edges, resulting in a graph with only 958 nodes and 957 edges. The algorithm performs this simplification in 0.18 seconds. *Cluster2* has a smaller information flow graph with 6218 nodes and 7543 edges. The graph reduction completes within 0.06 seconds and results in a graph with 359 nodes and 358 edges.

Zone Isolation We are now evaluating the analysis of the zone isolation goal for the large-scale infrastructure. For our evaluation, we consider all analysis cases for the following parameters: attack/safe, simplified/non-simplified graph, and different graph traversal models. *Attack* denotes an isolation breach and *Safe* denotes secure isolation.

For the graph traversal modeling using connected() in form of Horn clauses or transition rules, all tools we are evaluating either run out of memory (OFMC) or do not terminate within our time limit of 4 hours. We therefore focus our detailed performance analysis on our intruderHas graph traversal model with the following analysis cases.

- *Simplified Graph:* Attack **1**, Safe **2**
- *Non-Simplified:* Attack **3**, Safe **4**

The time measurements of the analysis cases for the different tools are depicted in Figure 4.7.

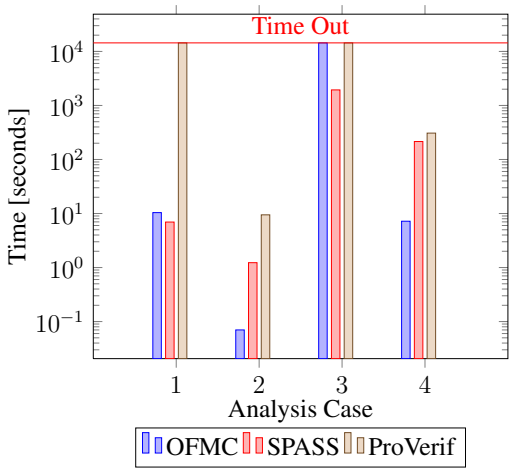


Figure 4.4: Time measurements (on logarithmic scale) for analysis cases of zone isolation.

The measurements show that ProVerif is only able to analyze the *Safe* configuration, because in the other case it does not terminate within our time frame of 4 hours. Since ProVerif is based similarly on resolution as SPASS (which terminates within the time limit for all problems), we suspect that the pre-processing of rules in ProVerif may be the cause.

OFMC yields good performance results and is very fast for analyzing such a large-scale infrastructure. We noticed a problem in analyzing the vulnerable cluster with the non-simplified graph, that is OFMC runs out of memory. SPASS terminates for all analysis cases and is faster for case 1 as OFMC.

Discussion The analysis of a large-scale infrastructure with regard to the zone isolation goal gave us insights into the efficiency of our modeling and the employed problem solvers. We learned that our initial modeling of connected() facts using Horn clauses or transitions were only applicable for small infrastructures and not for such real-world scenarios. Therefore, we developed the more efficient modeling of using intruderHas() facts for graph traversal, which made the analysis in a reasonable time frame possible. The complexity of this graph traversal is only linear to the number of edges, whereas the graph traversal using connected() yields a quadratic complexity.

Furthermore, we learned that problem solvers were overwhelmed by the detailed modeling of the infrastructure in form of our realization model. In case of security goals concerned with graph connectivity, we developed a graph refinement algorithm that simplifies the realization graph, but preserves its connectivity properties. The combination of efficient graph traversal modeling and graph simplification yielded results in the order of seconds for the analysis of our scenario infrastructure.

In terms of employed problem solvers, SPASS and OFMC performed best for our scenario.

4.8 Related Work

Virtual systems introduce several new security challenges [GR05]. Two important drivers that inspired our work is the increase of scale as well as the transient nature of configurations that render continuous validation with a variety of security goals more important.

Narain et al. [NCPT06] analyze network infrastructures with regard to *single point of failure* using a formal modeling language. In contrast, our approach focuses on a variety of high-level security goals, among them the absence of single point of failure, that can be evaluated using general-purpose model-checkers. Previous work has also analyzed network reachability in an automated way using specialized tools, for example, [XZM+04] for IP networks, [KSS+09] for VLANs, and [BSP+10a] for Amazon cloud configurations. Narain [Nar05] proposes modeling a network configuration using a formal language and do automated reasoning on this formal model. We are extending this concept by considering the entire virtualization infrastructure, not just networking resources. Ritchey et al. [RA00] employ model-checkers to check for vulnerabilities in networks.

The main differentiation of our work to previous ones is two-fold. First, we have a generic way to specify and verify security goals for virtualized infrastructures rather than specialized analysis. Second, our framework includes the modeling of a dynamic infrastructure, in particular one where the intruder can influence the topology (for instance by migrating machines) to mount an attack. This work is the first to formally verify security properties of virtualized infrastructures with this dynamic behavior.

4.9 Conclusion and Future Work

In this work we demonstrated our novel approach for the automated verification of virtualized infrastructures. We are able to specify a variety of security goals in a formal language and validate heterogeneous infrastructure against them. We are the first to employ general-purpose model-checker and theorem provers for this matter.

We studied three examples of static and dynamic problems, namely zone isolation, secure migration, and single point of failure. For each problem, we showed how to specify goals in the formal languages and proposed efficient modeling strategies. We successfully demonstrated the automated verification of these examples against small infrastructures. Finally, we also validated a large-scale infrastructure against the zone isolation secure goal and showed the practical feasibility of our approach.

Future work includes the further study of dynamic problems in virtualized infrastructure and their efficient analysis on large-scale infrastructures.

Chapter 5

Design and Architecture of Distributed Security Management

Chapter Authors:
Imad M. Abbadi, Andrew Martin, Anbang Ruan (OXFD),
Alexander Bürger, Michael Gröne, Norbert Schirmer (SRX),
Johannes Behl, Rüdiger Kapitza, Klaus Stengel (TUBS)

Cloud computing today is shaped by the involvement of a large number of entities with diverse trust relationships and large-scale and complex systems. Security management, in particular also in a distributed way, is essential for achieving a trusted cloud service that fulfills the requirements of high availability, fault tolerance, and scalability.

Section 5.1 approaches the security and management concerns of Cloud providers as well as users that stem from the dynamic nature of clouds. For example how can Cloud providers assure users that: (a.) dependent applications running on different VMs (Virtual Machines) are hosted within physical proximity (performance reasons); (b.) mutually exclusive VMs are not hosted at the same physical server (e.g. availability and security reasons); and (c.) when migrating VMs the new allocated physical servers satisfy users application requirements and security and privacy criteria. We propose a framework, which at this foundation stage focuses on providing secure environment for the management of Clouds' virtual layer. It also helps in establishing trust in Cloud's operational management.

Section 5.2 discusses the logical separation of security and cloud management, while providing the necessary and required integration. We propose an architecture that integrates Public-Key-Infrastructure (PKI) management with OpenStack.

Section 5.3 presents *DQMP*, a decentralized, fault-tolerant, and scalable quota-enforcement protocol. It allows customers to buy a fixed amount of resources (e. g., CPU cycles) that can be used flexibly within the cloud. DQMP utilizes the concept of diffusion to equally balance unused resource quotas over all processes running applications of the same customer. This enables the enforcement of upper bounds while being highly adaptive to all kinds of resource-demand changes.

5.1 Secure Virtual Layer Management in Clouds

5.1.1 Introduction

NIST defines 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'[MG09]. Cloud support three main deployment types Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS), and Infrastructure-as-a-Service (IaaS) [JNL10, MG09]. IaaS provides the most flexible type for Cloud users who prefer to have the greatest control over their resources, while SaaS provides the most restrictive type for Cloud users where Cloud providers have full control over the virtual resources.

Organizations when outsourcing their applications (or part of their applications) at IaaS Cloud would typically do the following (as discussed in [Abb11a, Abb11c]): The organization must first decide on the application that will be outsourced on the Cloud. The application nature, organization policy, and legislation factors would play an important role in this decision. For example, organization policy makers might reject to host applications, which process financial data, e.g. for legislative reasons. After the organization decides on the applications to be outsourced, it defines application requirements, which we refer to as *User Properties*. In potential Cloud *User Properties* would include the following:

Technical Requirements — For potential Cloud, in IaaS the organization enterprise architects team would provide an architecture for the outsourced infrastructure based on application requirements. This includes VMs, storage and network specifications. Enterprise architects could also provide the properties of outsourced applications, e.g. DBMS instances that require high availability with no single point of failure, middle-tier web servers that can tolerate failures, and the application nature is highly computational. Realizing these would enable Cloud providers to identify the best resources that can meet user requirements [Abb11a].

Service Level Agreement (SLA) — SLA specifies quality control measures and other legal and operational requirements. For example, these define system availability, reliability, scalability (in upper/lower bound limits), and performance metrics.

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 providers properly isolate VMs that run in the same physical platform from each other (i.e. problems of multi-tenant architecture [RTSS09b]); and (iii.) users might need to enforce geographical location restrictions on the processing and storage of their data.

Finally, for potential Cloud the organization would provide the above user properties via a set of APIs. The APIs would be supplied by the Cloud provider, which would then create virtual resources considering the provided user properties. Cloud provider manages the organizational outsourced resources based on the agreed user properties. In turn the organization pay Cloud provider on a pay-per-use model.

Current Cloud providers have full control over all hosted services in their infrastructure; e.g. 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) [CGJ+09, JNL10]. Cloud users have very limited control over the deployment of their services, have no control over the exact location of the provided services, and have no option but to trust Cloud provider to uphold the guarantees provided in the SLA. Potential Cloud which is expected to be used by critical applications must provide strong assurance to Cloud users that their requirements are continually enforced. Cloud users should be capable of attesting to the Cloud provider assurance level at any time. This is one of the top challenges in Cloud environment which is difficult to deal with considering Cloud's infrastructure complexity and dynamic nature [Abb11c, AL11, AFG+09, BSP+10b, CGJ+09, JNL10, KM10a, Mic09, RTSS09b].

5.1.1.1 Objectives and Limitations

In this work we focus on an important angle in the above direction. Specifically, our objective is to propose a framework that helps Cloud providers to securely manage the allocation of physical resources to users' virtual resources based on user properties and infrastructure properties. Simultaneously, we also propose a possible approach to assure users that their virtual resources are managed based on their defined requirements. It is important to stress that we do not claim that this work addresses all identified problems for establishing the framework — we mainly propose a framework architecture, propose the mechanisms for securely managing the framework, and identify challenges. For example, we do not discuss issues related to providing users the capabilities to control their own resources and management keys, which are planned future research.

5.1.2 Cloud Management

In this section we start by briefly outlining a conceptual model of the Cloud focusing on infrastructure management and relationships amongst components (detailed discussion of the model associated with real life deployment scenarios can be found at [Abb11a]).

5.1.2.1 Cloud Structure

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 resources that share common characteristics. Layering concept helps in understanding the relations and interactions amongst Cloud resources. We use the nature of the resource (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 resource (i.e. server, network, or storage) as the key characteristic for vertical slicing. Based on these key characteristics a Cloud is composed of three horizontal layers (Physical Layer, Virtual Layer, and Application Layer) and three vertical layers (Server Layer, Network Layer, and Storage Layer).

In the context of this work we mainly focus on *Horizontal Layer*. We identify a *Horizontal Layer* to be the parent of physical, virtual or 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. *Domains* at physical layer are related to Cloud infrastructure and, therefore, are associated with infrastructure properties and policies. *Domains* at virtual and application layers are Cloud user specific and therefore are associated with Cloud user properties. *Domains* that need to interact amongst themselves within a horizontal layer join a *Collaborating Domain*, which controls the interaction among members using a defined policy.

Domains and *Collaborating Domains* help in managing Cloud infrastructure, resource distribution and coordination in normal operations as well as during incidents such as hardware failures. The management of resources, including their relationships and interactions, is governed by polices. Such policies are established based on several factors including: infrastructure and user properties. Infrastructure properties are associated with Physical Layer Domains and Collaborating Domains, while user properties are associated with Virtual and Application Layers' Domains and Collaborating Domains.

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

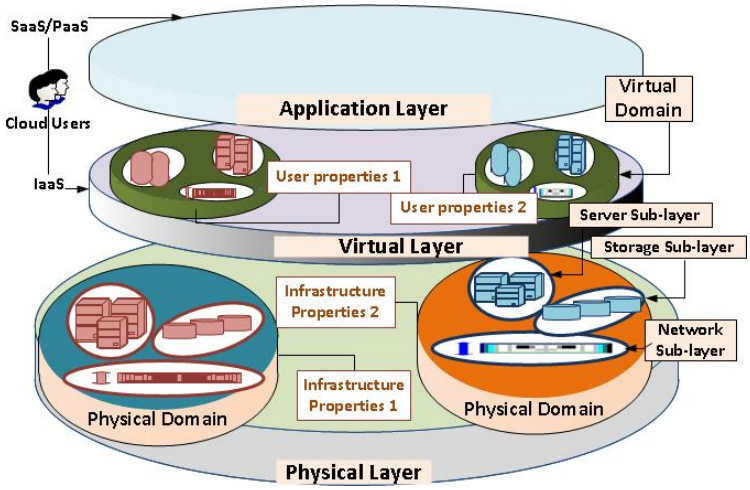


Figure 5.1: Cloud Taxonomy (source [Abb11a])

composed of groups of virtual resources. A group would typically run and manage a specific component within an *Application Domain*. Also, each group is associated with user properties which are related to the application component that is served by the group. Cloud providers use such properties to manage the group. For example, such user properties help Clouds to decide on i) Vertical Scalability of VMs within the group (i.e. minimum and maximum resources allocated to a VM based on current load), ii) Horizontal Scalability of the group (i.e. minimum and maximum number of VMs that can be allocated and deallocated within a group based on load/incidents), iii) deciding on the right *Physical Domain* that can serve the needs of the application (e.g. highly transactional application, DBMS that should provide no single point of failure, etc), and iv) helps management services, on incidents, to react based on user requirements (e.g. if *Physical Domain* fails then all groups served by the *Physical Domain* should transparently failover to another *Physical Domain* which is associated with properties that can serve the group requirements).

The above simplified taxonomy of the Cloud helps in realizing the challenges involved in managing and providing self-managed services (partially identified in [Abb11c]). This taxonomy does not contradict or even replace previously proposed ones (see, for example, [YBS08]) 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.

5.1.2.2 Illustrative Example

Figure 5.2 illustrates the above terms using a simplified multi-tier example deployed at a Cloud (see [Abb11a] for detailed discussion). The *Application Layer* has an *Application Domain* which is composed of two parts of a multi-tier application: middle-tier application and backend DBMS. The user provides his requirements on each application component (e.g. backend DBMS should provide no single point of failure, minimum/maximum resources' scalability values for each component, physical location restrictions on data storage/processing, etc). The *Physical Layer*, as illustrated in Figure 5.2, is composed of two *Physical Domains*: Physical Domain-1 which is physically configured and optimized by Cloud provider to host DBMS applications that can provide no single point of failure (e.g. configured to support Real Application Cluster [Ora11]); while Physical Domain-2 is physically configured and optimized

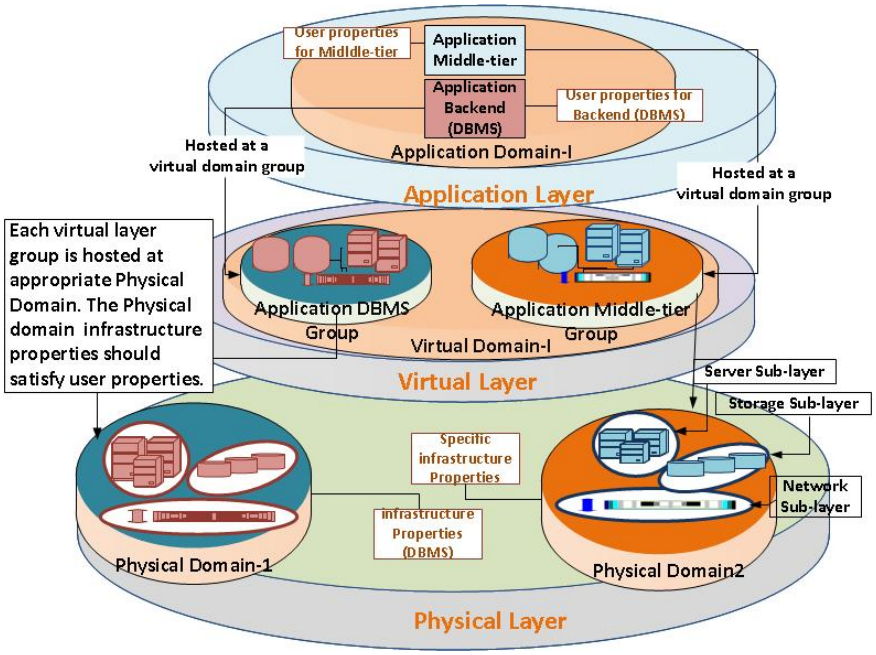


Figure 5.2: Multi-tier example using the provided Cloud structure

to host lightweight type of applications (e.g. middle-tier applications which mainly runs web-application servers).

The *Virtual Layer* creates a *Virtual Domain*, which has two *Groups* of virtual resources: application DBMS group runs application backend component and is hosted using Physical Domain-1, while the second group (i.e. application middle-tier group) runs application middle-tier component and is hosted using Physical Domain-2. As we discussed above, the hosting decision of a *Virtual Domain's Group* at a *Physical Domain* would be based on the *Physical Domain* infrastructure properties that best fit with user properties.

The concept of *Collaborating Virtual Domains* represents different applications which depend on each other. For example, an organization might have multiple applications sharing the same database. This means, for performance reasons, such applications need to run within physical proximity to the database. We propose for managing application dependencies to join all related *Virtual Domains* within a single *Collaborating Virtual Domain*. This collaborating domain is associated with policies defined by user governing the way Cloud should manage its member domains.

The concept of *Collaborating Physical Domains* can be realized by having redundant *Physical Domains* which have similar infrastructural properties. In this case the members of the *Collaborating Physical Domains* (based on defined policy) can automatically act as a backup of each other when any member *Physical Domain* has an emergency; e.g. has a failure, get overloaded or in maintenance window.

5.1.2.3 Virtual Control Centre

In this section we outline part of Cloud's virtual resource management, detailed discussion of which can be found in previous work [Abb11a, Abb11c]. Currently there are many tools for managing Cloud's virtual resources, e.g. vCenter [VMw10] and OpenStack [Ope10b]. For convenience we call such tools using a common name Virtual Control Centre (VCC), which is a Cloud specific device that manages virtual resources and their interactions with physical

resources using a set of software agents. Currently available VCC software agents have many security vulnerabilities and only provide limited automated management services (what we refer to as self-managed services). For example, the management of *Collaborating Virtual Domain* and *Collaborating Physical Domain* is controlled manually by Cloud employees using VCC. VCC manages the infrastructure by establishes communication channels with physical servers to manage Cloud's Virtual Machines (VMs). VCC establishes such channels by communicating with Virtual Machine Manager (VMM) running on each server. Such management helps in maintaining the agreed SLA and Quality of Service (QoS) with customers. VCC will play a major role in providing Cloud's automated self-managed services, which are mostly provided manually at the time of writing. For potential Clouds the factors which affect decisions made by VCC self-managed services are summarized as follows.

Infrastructure Properties — Clouds' physical infrastructure are very well organized and managed. Enterprise architects, for example, build the infrastructure to provide certain services, and they are aware about physical infrastructure properties for each infrastructural component and groups of components. Examples of such properties include: components reliability and connectivity, components distribution across Cloud infrastructure, redundancy types, servers clustering and grouping, and network speed.

User Properties — Exactly as discussed in section 5.1.1.

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 and redundancy type), infrastructure policy, and other changes (increase/decrease system load, component failure and network failure).

Self-managed services for potential Cloud should automatically and continually manage Cloud environment with minimal human intervention. It should always enforce Cloud user properties; e.g. ensure that user resources are always hosted using physical resources which have properties enabling such physical resources to provide the services as defined in user properties.

5.1.3 Management Framework Requirements

Our proposed framework forms the foundation which helps in establishing trust in the operation of the Cloud. In previous sections we discussed Cloud structure and self-managed services. In this work we are mainly concerned about two aspects of such management services: i) support self-managed services with secure environments which enable them to securely manage virtual layer resources at physical layer resources (i.e. VM creation and migration) based on both user properties and infrastructure properties; and ii) propose mechanisms to enable users to assess the trustworthiness of such management services. Point (i) requires establishing trustworthy and secure infrastructure across Clouds' distributed entities, while point (ii) involves establishing a chain of trust between Cloud users and the Cloud infrastructural resources (see the blue-dash arrow in Figure 5.3).

One of the key features of Cloud is providing transparent infrastructure management. Therefore, the chain of trust needs to be established at two stages: 1) a chain of trust between users and VCC management agents, and 2) a chain of trust between VCC management agents and Cloud's infrastructural resources. We now 'informally' discuss the requirements to achieve these objective. Users need to establish trust in VCC, for example, by attesting to the trustworthiness of its management agents. Also, VCC management agents need to establish trust in the infrastructural resources. This requires the following: (1.) VCC and VMMs should attest to each other execution environment; (2.) VCC and VMMs need to have management agents

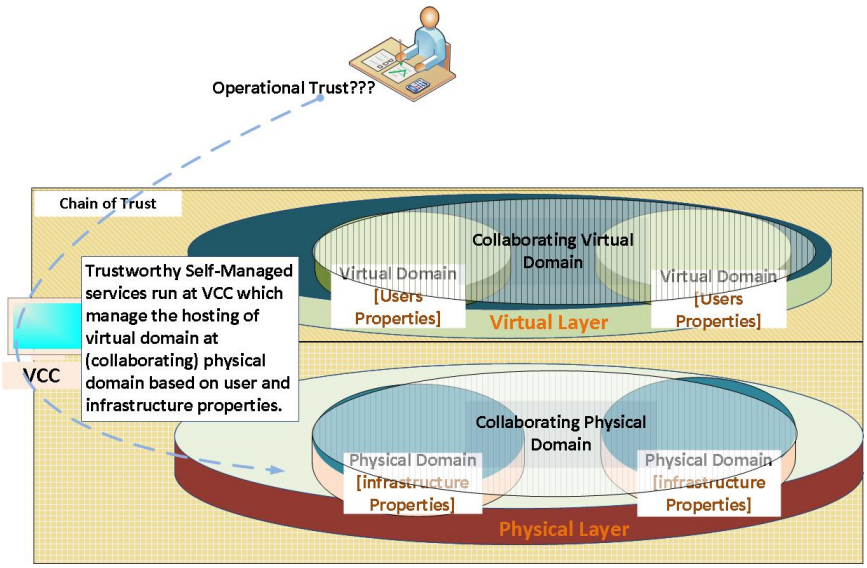


Figure 5.3: The proposed framework — focusing on chain of trust

that are trusted to behave as expected. Such agents should have mechanisms to ascertain their trustworthiness to remote parties; (3.) VCC and VMMs should provide protected storage functions; (4.) VCC and VMMs 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. In this work we do not discuss the last two points and leave them for planned future research.

The provision of secure and trustworthy infrastructure for the operation of self-managed services would 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 in next paragraph; (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. In this work we do not discuss points 5–6.

The Cloud infrastructure must be capable of protecting the integrity, confidentiality and availability of Cloud's management data from 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. Our proposed mechanism in this work partially mitigate the risks of Clouds' insiders — we do not consider insiders attacks at hypervisor and hardware levels.

5.1.4 Device Properties

We require that devices are commercial off-the-shelf hardware enhanced with trusted computing technology that incorporates a Trusted Platform Module (TPM), as defined by the Trusted Computing Group (TCG) specifications [Tru07b]. Trusted computing systems are platforms whose state can be remotely tested, and which can be trusted to store security-sensitive data in ways testable by a remote party. Trusted computing technology can enforce access control policies associated with a resource in such a way that a user cannot bypass these policies, whilst maintaining access to the resource. TCG compliant trusted platforms (TP) are not expensive, and are currently available from a range of PC manufacturers, including Dell, Fujitsu, HP, Intel and Toshiba. We now provide a very brief overview of the main entities in TCG compliant platforms, which are required in the proposed scheme. TCG is a wide subject and has been discussed by many researchers; we will not address the details of TCG specifications in this work for space limitations.

The TCG specifications require each TP to include an additional inexpensive hardware component to establish trust in that platform. This component is referred to as the Trusted Platform Module (TPM), which has protected storage and protected capabilities. Once a TPM has been assigned an owner, it generates a new Storage Root Key pair (SRK), which is used to protect all TPM keys. The private part of the SRK is stored permanently inside the TPM. Other TPM objects (key objects or data objects) are protected using keys that are ultimately protected by the SRK in a tree hierarchy structure. The entries of a TPM platform configuration registers (PCRs), where integrity measurements are stored, are used in the protected storage mechanism. This is achieved by comparing the current PCR values with the intended PCR values stored with the data object. If the two values are consistent, access is then granted and data is unsealed[1]. Storage and retrieval are carried out by the TPM. Therefore, if a software process relies on the use of secrets, it cannot operate unless it and its software environment are correct.

A TPM can generate two types of keys, known as migratable and non-migratable keys. Migratable keys can be transmitted to other TPs if authorised by both a selected trusted authority and the TPM owner. A non-migratable key is bound to the TP that created it. The TP associates the current platform software state, which is stored in PCRs, with the non-migratable key, and then protects them using the SRK. Stored secrets are only released after the platform's PCRs have been compared with the values associated with the stored key. Data encrypted using a non-migratable key can leave the TP if and only if the software agent (whose execution status matches the one associated with the non-migratable key, i.e. is authorised to read data encrypted using the non-migratable) authorises the release of the data to other platforms.

Establishing trust in a TP is based on the mechanism that is used for measuring, reporting and verifying platform integrity metrics. TP measurements are performed using the RTM (Root of Trust for Measurement), which measures software components running on a TP. The RTS (Root of Trust for Storage) stores these measurements inside TPM shielded locations (i.e. the PCR). Next, the RTR (Root of Trust for Reporting) mechanism allows TP measurements to be reliably communicated to an external entity in the form of an integrity report. The integrity report is signed using an AIK[2] (Attestation Identity Key) private key, and is sent with the appropriate identity credential. This enables a Verifier to be sure that an integrity report is bound to a genuine TPM.

[1] Seal/unseal are TCG terms used for encrypting/decrypting a data object. Seal binds a data object with an integrity measurement that must match the platform PCR value when unsealing the object. Also, a data object must be unsealed on the same TPM that sealed the object.

[2] AIKs are signature key pairs function as aliases for the TP; they are generated by the TPM, and the public part is included in a certificate known as an Identity Credential, signed by a trusted third party called a privacy certification authority (privacy CA). The identity credential asserts that the (public part of the) AIK belongs to a TP with specified properties, without revealing which TP the key belongs to.

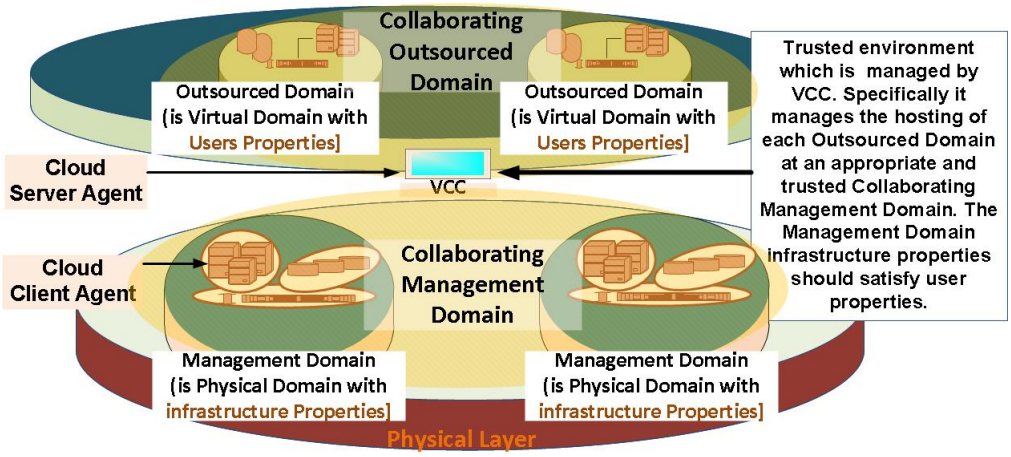


Figure 5.4: Framework Domain Architecture

5.1.5 Framework Architecture

In this section we propose a framework which forms the foundation to help in addressing the identified requirements in section 5.1.3. The framework uses the dynamic domain concept proposed in [AA08b]. We start by defining the dynamic domain concept, and then discuss the adaptation of such concept to architect the framework.

5.1.5.1 Dynamic Domain Concept

Definition 5.1.1 ***A Dynamic Domain*** *represents a group of devices that need to share a pool of content. Each dynamic domain has a unique identifier i_D, a shared unique symmetric key k_D and a specific PKL_d composed of all devices in the dynamic domain. k_D is shared by all authorized devices in a dynamic domain and is used to protect the dynamic domain content whilst in transit. This key is only available to devices that are member of the domain. Thus only such devices can access the pool of content bound to the domain. Each device is required to securely generate for each dynamic domain a symmetric key k_C, which is used to protect the dynamic domain content when stored in the device.*

A device can join multiple dynamic domains to access content bound to these domains. Devices not member of the domain do not possess a copy of k_D, and hence cannot decrypt domain-specific content.

5.1.5.2 Proposed Architecture

Our proposed framework architecture is composed of the following (see Figure 5.4): *Cloud provider Management Domain* (MD), *Cloud provider Collaborating Management Domain* (CMD), *User Outsourced Domain* (OD), and *User Collaborating Outsourced Domain* (COD).

We now map the above domains using the Cloud infrastructure taxonomy concept, which is summarized in section 5.1.2. A Cloud provider MD and CMD represents a Physical Domain and Collaborating Physical Domain at Cloud Infrastructure Physical Layer. User OD and COD represents a Virtual Domain and Collaborating Virtual Domain at the Virtual Layer. We do not consider Application Layer Domains, which is outside the scope of this work to discuss and are planned future work.

Definition 5.1.2 ***Management Domain:*** *MD represents a group of devices at Physical Layer. The capabilities of devices member of MD and their interconnection reflect the overall properties of MD. Such properties enable MD to serve the part of user requirements related to physical layer (e.g. geographical location restrictions, restrictions on dependent VMs to run within physical proximity or not to run in the same physical platform, etc).*

An MD has a specific policy defined by Cloud architect to manage the behaviour of MD members in normal operations and on incidents when providing services to OD and/or COD. Such policy, which is controlled by VCC, helps in providing Cloud properties (e.g. availability and resilience) as reflected at virtual resources hosted by MD (i.e. OD/COD). Each MD is also associated with infrastructure properties enable the MD to serve certain category of user requirements, as we discussed earlier. Each MD has a specific credentials consisting of a unique identifier i_{md}, a unique symmetric key k_{md}, and a public key list (PKL_{md}). These are defined as follows.

Definition 5.1.3 ***MD identifier*** *i_{md} is a unique number that we use to identify an MD. It is securely generated and protected by the TPM of VCC.*

Definition 5.1.4 *MD key k_{md} is used to protect management data that controls the behaviours of MD. k_{md} is a symmetric key that is securely generated and protected by the TPM of VCC. k_{md} is not available in the clear even to Cloud administrators, it is shared between all devices member of MD, and it can only be transferred from VCC to a device when the device joins the MD.*

Definition 5.1.5 ***MD's public key list (PKL_{md})*** *is an MD-specific list that is composed of the public keys of all devices of MD. Enterprise architects assign devices to each MD by providing each device public key to VCC in a form of PKL. The PKL_{md} is securely protected and managed by VCC.*

Definition 5.1.6 ***Collaborating Management Domain:*** *CMD represents groups of MDs that have similar infrastructural properties. Such grouping enables all MDs member of CMD to serve as a backup of each other, such as on MD failure, maintenance window, or overloaded resources. This operation is controlled by a defined policy at VCC. Such policy considers user properties and infrastructure properties. For example, before OD can migrate from MD_1 to MD_2 both MDs must be within the same CMD and user requirements must be validated against MD_2 properties, e.g. users might have restrictions when migrating across different legal jurisdictions, or even when migrating a resource across long distant data centres user might require migrating all COD members for performance reasons.*

Definition 5.1.7 ***Outsourced Domain:*** *OD consists of the virtual machines which host user outsourced applications at the Cloud. As outlined in section 5.1.2.3 the VCC establishes and manages **OD membership** (i.e. it creates/migrates/controls the hosting if virtual resources at MD and CMD) based on User Properties and Infrastructure Properties. As in the case of MD, OD has a unique identifier i_{od}, a shared unique key k_{od} and a specific PKL_{od} composed of all devices in the OD. k_{od} is used to protect OD content that is needed to be shared between OD devices. This key is only available to devices that are members of OD. OD credentials have similar definitions provided in def 5.1.3, 5.1.4, and 5.1.5, but managed by the user and not VCC. It is outside the scope of this work to discuss user application deployment issues and key management at OD and COD, as our focus is on providing secure virtual layer management at physical*

layer. Future work will focus on mechanism for providing users with full control over the security of their deployed data inside OD and COD without getting involved into infrastructural complexities.

Definition 5.1.8 ***Collaborating Outsourced Domain:*** *COD Consists of groups of related ODs that share common policy. For example, the policy could state that i) COD members should be hosted within physical proximity for performance reason, and ii) COD members should not be hosted in the same MD as in the case of a primary and standby DBMS. Such a policy is defined based on user properties but controlled by VCC.*

In section 5.1.9 we discuss the benefits of the proposed domain structure.

5.1.6 Required Software Agents

The proposed framework architecture has two types of software agents (as illustrated in Figure 5.4): Cloud server agent and Cloud client agent. These software agents run on trusted devices that must have all TP properties, as outlined in section 5.1.4. Cloud server agent runs in VCC, while Cloud client agent runs at physical devices member of MD. Other software agents might be required to run inside VMs member of OD and possibly controlled by Cloud users, which is outside the scope of this work to discuss and is planned future work. The way the functions of these agents are implemented is described in section 5.1.7. For convenience, herein we use the word content to mean infrastructure management data.

Assumption 5.1.9 *We assume the identified software agents are designed in such a way that they do not reveal domain credentials in the clear, do not transfer domain protection keys to others, and do not transfer sensitive domain content unprotected to others. Although, this is a strong assumption; however, recent research shows promises in the direction of satisfying such an assumption [MLQ+10]. TCG compliant hardware using the sealing mechanism alone is not enough to address such an assumption. Trustvisor ([MLQ+10]) moves one step forward and focuses on protecting content encryption key utilizing recent development in processors technology (e.g. Intel TXT); however, this does not protect clear text data once decrypted and more work is required in this direction. Formal security analysis of such work and others in this direction is very important to the success of our framework, which is a planned future work.*

5.1.6.1 Cloud Server Agent Functions

The Cloud server agent has the following functions:

(a.) Create and manage MDs. This includes the following: (i.) Securely generating and storing MDs protection keys; (ii.) Attesting to the execution environment status of devices whilst being added to the MD and ensuring they are trusted to execute as expected; hence trusted to securely store the MD key and to protect MD content; (iii.) Adding and removing devices to MD by releasing the domain-specific key to devices joining the MD; and (iv.) Backing up and recovering MD-specific credentials.

(b.) Manage policies, and ensure protected content is only accessible to authorized devices. These covers the collaboration between related MDs forming a CMD.

(c.) Installs and manages Cloud client agent at devices member of MD.

(d.) As outlined in section 5.1.2.3, Cloud server agent manages the policy of hosting each OD at appropriate MD within a specific CMD, and also manages the policy of hosting COD members. This is by matching user properties with Clouds infrastructural properties, which continually considers changes and incidents.

5.1.6.2 Cloud Client Agent Functions

Cloud client agents have the following functions: (i.) Used by physical devices when interacting with Cloud server agent for joining an MD, and (ii.) Manage and enforce MD/CMD policy at individual physical devices as defined by Cloud server agent. This, for example, includes managing the allocation of OD members on resilient architecture at an MD and across CMD members. In addition, Cloud client agents are responsible of ensuring that access to exchanged management data and policy of MD/CMD is granted only to authorized agents bound to specific devices. Client agents access critical management data by communicating with client agents running at devices member of MD/CMD.

5.1.6.3 Cloud Server Agent Initialization

This section describes the procedure of initializing the Cloud server agent discussed in section 5.1.6.1. The main objective of this procedure is to prepare the server agent to implement the framework of the proposed scheme. This includes the following: (1.) System administrators install the server agent on VCC — the installation of the server agent includes generating a non-migratable key pair (Pr,Pu) to protect domain credentials; and (2.) The server agent could also manage security administrator(s) credentials and securely stores them to be used whenever administrator(s) need to authenticate themselves to the server agent.

The first time security administrators run the server agent it performs the following initialization procedure (as described by algorithms 1). The objective of this algorithm is to initialize the server agent. The server agent executes and sends a request to the VCC-specific TPM to generate a non-migratable key pair, which is used to protect domain credentials. TPM then generates this key and seals it to be used by the server agent when the hosting device execution status is trusted.

The server agent then needs to ensure that only security administrators can use the server agent. For this the server agent instructs security administrators to provide their authentication credentials (e.g. password/PIN), as described by algorithm 2. The objective of this algorithm is to enroll system administrators into the server agent. The server agent then requests the TPM to store the authentication credentials of the security administrators associated with its trusted execution environment state (i.e. the integrity measurement as stored in the TPM's PCR) in the VCC protected storage. We mean by storing data in a protected storage is 'sealing data' in TCG terms, so that data can only be accessed by the trusted agent. The authentication credential is used to authenticate security administrators before using the server agent; see algorithm 3.

Given the definitions and the assumptions above, the protocol is described by algorithms 1, 2, and 3. The objective of the protocol is installing the server agent at VCC, which generates the non-migratable key to encrypt the domain credentials and securely store a copy of the security administrators' credentials. The protocol is used every time security administrators want to manage the domains. Following are the notations used in the provided algorithms:

M is the software agent; **TPM** is the TPM on VCC; S is the platform state at release as stored in the PCR inside the TPM; and **(Pu, Pr)** is a non-migratable key pair such that the private part of the key Pr is bound to TPM, and to the platform state S. The following protocol functions are defined in [Tru07b]: $\text{TPM}_{\text{C}_{createWrapKey}}$, $\text{TPM}_{\text{L}_{oadKey2}}$, $\text{TPM}_{\text{S}_{eal}}$, and $\text{TPM}_{\text{U}_{nseal}}$.

Algorithm 1 Server Agent Initialization

1. $M \rightarrow$ TPM: $\text{TPM}_{\text{C}_{createWrapKey}}$.
2. TPM: generates a non-migratable key pair (Pu, Pr).

   Pr is bound to TPM, and to the required platform state S at release, as stored in the PCR inside the TPM.
3. TPM$\rightarrow$ M: TPM_KEY12[Pu, Encrypted Pr, TPM_KEY_STORAGE, tpmProof=TPM (NON-MIGRATABLE), S, Auth_data]

Algorithm 2 Administrators Registration

1. $M \rightarrow$ Administrators: Request for system administrators authentication credentials.
2. $M \rightarrow$ TPM: $\text{TPM}_{\text{L}_{LoadKey2}}$(Pr).

   Loads the private key Pr in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr (i.e. S). If the PCR value does not match S, M returns an appropriate error message.
3. $M \rightarrow$ TPM: $\text{TPM}_{\text{S}_{Seal}}$(Authentication_Credential).

Algorithm 3 Authentication Verification

1. $M \rightarrow$ Administrators: Request for authentication credentials.
2. $M \rightarrow$ TPM: $\text{TPM}_{\text{L}_{LoadKey2}}$(Pr). TPM on M loads the private key Pr in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr (i.e. S). If the PCR value does not match S, M agent returns an appropriate error message.
3. $M \rightarrow$ TPM: $\text{TPM}_{\text{U}_{Unseal}}$(Authentication_Credential).
4. TPM: Decrypts the string Authentication_Credential and passes the result to M.
5. M: Authenticates the administrators using the recovered authentication credentials. If authentication fails, M returns an appropriate error message.

5.1.6.4 Cloud Client Agent Initialization

This section describes the procedure of initializing Cloud client agents. The goal of this procedure is to prepare devices to join an MD and to collaborate within CMD, which includes installing the client agents at physical devices. This covers generating a non-migratable key to protect domain credentials.

The protocol of initializing client agent is described by algorithm 4. The objective of this algorithm is to install at each device a copy of the client agent, which generates a non-migratable key to protect MD credentials. Following are the notations used in the provided algorithm: D is the client agent running on a client device; **TPM**, S and **(Pu, Pr)** have the same meanings provided earlier.

Algorithm 4 Client Agent Initialization

1. $D \rightarrow \text{TPM}_D$: $\text{TPM}_{\text{C}_{createWrapKey}}$.
2. TPM: generates a non-migratable key pair (Pu, Pr).
2. TPM$\rightarrow$ D: TPM_KEY12[Pu, Encrypted Pr, TPM_KEY_STORAGE, tpmProof=TPM (NON-MIGRATABLE), S, Auth_data]

5.1.7 Framework Workflow

This section discusses a possible workflow of the proposed system framework. At this early stage of our work we propose a set of protocols as a proof of concept without providing any

formal analysis. This is to clarify how the framework components could possibly be managed. Once we proceed in this work and address the identified challenges, we then need to provide a formal analysis in which the proposed protocols will likely be updated.

5.1.7.1 MD and CMD Establishment

In this section we discuss the procedure of establishing an MD and joining related MDs to collaborate within a CMD, which are managed by the Cloud server agent. In the provided protocol we use the same notations described earlier (note that M resembles the Cloud server software agent running on VCC).

In this subsection we require that the Cloud server agent has already been installed on VCC, exactly as described earlier in section 5.1.6.3. This includes installing the server agent, which interacts with the TPM to generate a non-migratable key pair that can be only used by the server agent. This key pair is used to protect MD credentials.

Domain establishment begins when the Cloud administrators wants to add a new domain to the Cloud infrastructure. Administrators instruct the server agent to create a new MD. The server agent authenticates administrators as described in algorithm 3. If authentication succeeds the server agent interacts with the TPM to securely generates MD specific secret key k_{md} and identifier i_{md}, as described by algorithm 5.

At the successful completion of this protocol MD credentials are initialized at VCC, which include the MD key, MD identifier and an empty PKL. These are protected by VCC, which manages MD membership.

Cloud employees assign devices to MD based on the required overall MD properties. They also join related MDs in an CMD using policies which consider COD policy. Such policies control the migration of OD resources within a single MD resources and across CMD members.

Algorithm 5 Management Domain Establishment

1. $M \rightarrow$ TPM: $\text{TPM}_{GetRandom}$.

   TPM generates a random number to be used as a MD domain key k_{md}.

2. TPM$\rightarrow$ M: k_{md}

3. $M \rightarrow$ TPM: $\text{TPM}_{GetRandom}$.

   M generates a unique number to be used as MD domain identifier i_{md}.

4. TPM$\rightarrow$ M: i_{md}

5. The MD domain credentials k_{md}, i_{md}, and an empty PKL_{md} are stored in VCC protected storage, and sealed to the server agent so that only the server agent can access these credentials when its execution status is trusted. This is achieved as follows.

   $M \rightarrow$ TPM: $\text{TPM}_{LoadKey2}$(Pr);

   Loads the private key Pr in the TPM trusted environment to be used in the Sealing function, after verifying the current PCR value matches the one associated with Pr (i.e. S). If the PCR value does not match S, M returns an appropriate error message.

   $M \rightarrow$ TPM: $\text{TPM}_{Seal}(k_{md}||i_{md}||\text{PKL}_{md})$.

   TPM securely stores the string $k_{md}||i_{md}||\text{PKL}_{md}$ using the platform protected storage, such that they can only be decrypted on the current platform by M, and only if the platform runs as expected (when the platform PCR values matches the ones associated with Pr, i.e. S).

5.1.7.2 Adding Devices to MD and CMD

This section describes the process for adding a device to a MD, and joining MDs to a CMD (the process could also be applied to adding devices to OD, which is outside the scope of this work to discuss as it involves substantial further work on supporting users with mechanisms to have ultimate control over key management of their OD's content protection keys). Following notations are used in the provided protocol.

D is the client agent running on a device; M is the server agent running on VCC; $\mathbf{TPM}_D$ is the TPM on a client device; $\mathbf{TPM}_M$ is the TPM on VCC; S_D is the platform state at release as stored in the PCR inside the TPM_D; S_M is the platform state at release as stored in the PCR inside the TPM_M; $(\mathbf{Pu}_D, \mathbf{Pr}_D)$ is non-migratable key pairs such that the private part of the key Pr_D is bound to TPM_D and to the platform state S_D; $(\mathbf{Pu}_M, \mathbf{Pr}_M)$ is non-migratable key pairs such that the private part of the key Pr_M is bound to TPM_M and to the to the platform state S_M; i is the domain specific identifier, which is equal to i_{md} for MD; **PKL** is the domain public key list, which is equal to PKL_{md} for MD; k is the domain specific content protection key, which is equal to k_{md} for MD; $\mathbf{Cert}_M$ is the VCC certificate; $\mathbf{Cert}_D$ is the joining device certificate; A_M is an identifier for the server agent device included in Cert_M; A_D is an identifier for a client agent device included in Cert_D; $\mathbf{Pr}_{MAIK}$ is the corresponding private key of the public key included in Cert_M; $\mathbf{Pr}_{DAIK}$ is the corresponding private key of the public key included in Cert_D; N_1 is a randomly generated nonce; N_2 is a randomly generated nonce; $e_{Pu_D}(Y)$ denotes the asymmetric encryption of data Y using key Pu_D, and where we assume that the encryption primitive in use provides non-malleability, as described in [Int06]; and **SHA1** is a one way hash function.

The client agent on the device sends a join domain request to the server agent to install the domain specific key k. This request includes the domain specific identifier i this is achieved as follow.

$D \rightarrow M$: Join_Domain

Two algorithms are then initiated to add the device to the domain. The first algorithm involves the server agent and the client agent to mutually authenticate each other conforming to the three-pass mutual authentication protocol [Int98]. The server agent sends an attestation request to the client agent to prove its trustworthiness then the client agent sends the attestation outcome to the server agent. These steps are achieved using algorithm 6.

Adding a device into a domain uses the second algorithm 7, which starts upon successful completion of algorithm 6. The objective of algorithm 7 is to securely transfer the key k to client agent D. k is sealed on D, so that it is only released to client agent when its execution environment is as expected. If D execution status is trusted, the server agent checks if the device's public key is included in the public key list of the domain. If so, it securely releases the domain specific key k to D using algorithm 7. The key are sealed on D, so that they are only released to client agent when its execution environment is as expected.

Upon the successful completion of the above algorithms the client agent and the server agent establish a trusted secure communication channel that is used to transfer the domain key to the client agent. This channel is also used to transfer MD policies which cover the following: i) MD-specific resource management policy, ii) related MDs' identifier which form a CMD, and iii) CMD management policy. The established secure channel provides the assurance to the server agent about the client agent state, and, also, forces the future use of the transferred key to the agent on specific trusted state. The device hosting D is now part of the domain, as it possesses a copy of the key k, and its public key matches the one stored in the server agent. Member devices of the domain can access the domain associated content, and hence such content are now shared by all devices member of the domain.

Algorithm 6 Client and Server Agents Mutual Authentication

1. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{GetRandom}$.
2. $\text{TPM}_M \rightarrow M$: Generates a random number to be used as a nonce N_1.
3. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{LoadKey2}(\text{Pr}_{MAIK})$;

   Loads the server agent hosting device AIK in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr_{MAIK}.
4. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{Sign}(\text{N}_1)$.
5. $\text{TPM}_M \rightarrow M \rightarrow D$: $N_1||\text{Cert}_M||\text{Sign}_M(N_1)$.
6. D: verifies Cert_M, extracts the signature verification key of M from Cert_M, and checks that it has not been revoked, e.g. by querying an OCSP service [MAM+99]. D then verifies message signature. If the verifications fail, D returns an appropriate error message.
7. $D \rightarrow \text{TPM}_D$: $\text{TPM}_{GetRandom}$.
8. $\text{TPM}_D \rightarrow D$: Generates a random number N_2 that is used as a nonce.
9. $D \rightarrow \text{TPM}_D$: $\text{TPM}_{LoadKey2}(\text{Pr}_{DAIK})$;

   Loads the private key Pr_{DAIK} in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr_{DAIK}.
10. $D \rightarrow \text{TPM}_D$: $\text{TPM}_{CertifyKey}(\text{SHA1}(N_2||N_1||A_M||i),\text{Pu}_D)$. TPM_D attests to its execution status by generating a certificate for the key Pu_D.
11. $\text{TPM}_D \rightarrow D$: $N_2||N_1||A_M||Pu_D||S_D||i||\ \text{Sign}_D(N_2||N_1||A_M||i||Pu_D||S_D)$.
12. $D \rightarrow M$: $N_2||N_1||A_M||Pu_D||S_D||i||\text{Cert}_D||\ \text{Sign}_D(N_2||N_1||A_M||i||Pu_D||S_D)$.
13. M verifies Cert_D, extracts the signature verification key of D from the certificate, and checks that it has not been revoked, e.g. by querying an OCSP service. M then verifies message signature, message freshness by verifying the value of N_1, and then verifies it is the intended recipient by checking the value of A_M. M determines if D is executing as expected by comparing the platform state given in S_D with the predicted platform integrity metric. If these validations fail, then M returns back an appropriate error message.

Algorithm 7 Sealing Domain Key to Client Agent

1. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{LoadKey2}(\text{Pr}_M)$.

   TPM on M loads the private key Pr_M in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr_M (i.e. S_M). If the PCR value does not match S_M, the server agent returns an appropriate error message.
2. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{Unseal}(k||i||\text{PKL})$.
3. $\text{TPM}_M \rightarrow M$: decrypts the string $k||i||\text{PKL}$ and passes the result to M.
4. M verifies i matches the recovered domain identifier and Pu_D is included in the PKL. If so M encrypts k using the key Pu_D as follows $e_{Pu_D}(k)$.
5. $M \rightarrow \text{TPM}_M$: $\text{TPM}_{CertifyKey}(\text{SHA1}(N_2||A_D||e_{Pu_D}(k)),\text{Pu}_M)$.
6. $\text{TPM}_M \rightarrow M$: attests to its execution status by generating a certificate for the key Pu_M, and sends the result to M.
7. $M \rightarrow D$: $N_2||A_D||Pu_M||S_M||\ e_{Pu_D}(k)||\text{Sign}_M(N_2||A_D||e_{Pu_D}(k)||\ Pu_M||S_M)$.
8. The device D verifies message signature, it is the intended recipient by checking the value of A_D, and verifies message freshness by checking the value of N_1. If verifications succeed, D stores the string $e_{Pu_D}(k))$ in its storage.

5.1.7.3 OD and COD Establishment

In section 5.1.1 we discuss the general steps that organizations (as Cloud users) would follow, when deciding to outsource part of (or the whole of) their infrastructure to the Cloud. In this section we assume that the Cloud user has chosen the applications to be outsourced on the Cloud, defined their requirements (i.e. user properties, as identified in section 5.1.1), and negotiated the SLA with the Cloud. We now mainly focus on establishing an OD and joining related ODs within an COD based on user technical requirements, which is part of user properties.

The user first would need to communicate with the Cloud provider supplied APIs to create virtual resources considering the defined user properties. For simplicity we assume that the server agent running on VCC is the entity that also serves such APIs (this assumption can

be easily changed, but we do not want to get into discussing secure communications between different software components inside the Cloud, which is outside the scope of this work).

The work on dynamic domain [AA08a] has provided a protocol for establishing trusted secure channels between collaborating organizations' endpoints. The trusted secure channel provides 'offline' assurance to such endpoints about each other execution environment is trusted to behave as expected. Establishing such a channel requires the Cloud to provide an authentic copy of its certificate to the Cloud user. A Cloud provider could provide his certificate in different means, e.g. sign the certificate using a reputable PKI authority and publish it on the Cloud website.

The trusted secure channel establishes a secure and trustworthy communication between the Cloud user device and the Cloud server agent running on VCC. As discussed in section 5.1.7.2, Cloud server agent and client agent have already established trusted secure channels. Such secure channels assure users that the channel from user device to VCC, and then from VCC to physical devices are trusted. The following steps are then followed:

(1.) The user sends, via supplied APIs, a request to the Cloud server agent to establish IaaS virtual resources. The request includes the user technical requirements. Such requirements should be specified using a standard language or submitted via defined online form, which is outside our scope to discuss.

(2.) The Cloud server agent validates user properties. If validation succeeds, the Cloud server agent identifies CMD that can serve user properties. Such identification would mainly be based on the infrastructure properties of MDs' member of the CMD.

(3.) Based on user properties and the identified CMD, the Cloud server agent establishes user-specific processes and policies. These define how CMD would manage OD, for example, it defines for each OD the primary MD and backup MDs which should be member in the same CMD. Such processes and policies enable the primary MD to instantiate and control the user OD virtual resources. It also contains management decisions related to other ODs member in the same COD.

(4.) The Cloud server agent coordinates with the client agents at the primary MD to create user VMs.

(5.) Cloud client agents coordinate amongst themselves and create VMs, as defined in the user requirements.

(6.) Cloud server agent sends the details of the newly created VMs (PKL, IP addresses, and default authentication details) to the user.

(7.) The user can now interacts with his new OD. Future work will focus on providing mechanisms enabling Cloud users to have full control over OD credentials.

Section 5.1.9 discusses the advantages of the above steps on establishing a chain of trust between Cloud users and providers, and on establishing secure management infrastructure.

5.1.8 Planned Implementation Layout

In this section we briefly outline part of our implementation architecture, which we are working on as part of TClouds project[3]. We use OpenStack Compute as management framework to

[3] http://www.TClouds-project.eu

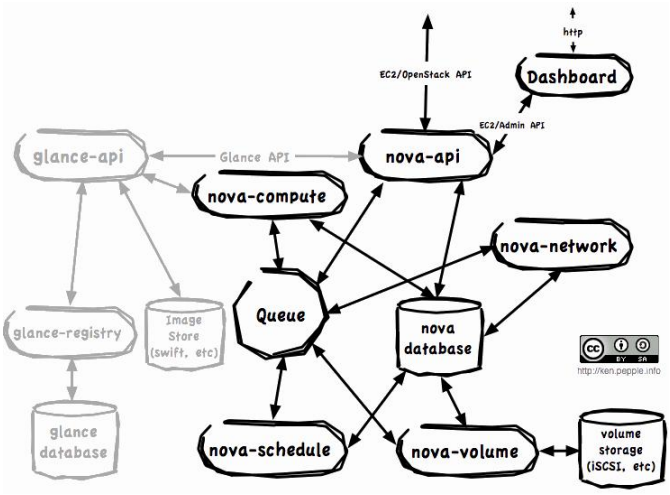


Figure 5.5: OpenStack Components (Source [Ope11])

represent VCC. OpenStack Compute is composed of many components as illustrated in Figure 5.5. For space limitations we only discuss the ones related to our framework (see [Ope11] for detailed discussion about OpenStack components): *nova-api* intermediates the communications between OpenStack and Cloud users, *nova-database* is the central repository for OpenStack management data, *nova-schedule* manages the hosting of VMs at Clouds physical layer, and *nova-compute* creates and terminates VMs.

In our prototype we plan to use *nova-database* to securely store user properties and the structure of the Cloud (following the taxonomy discussed in section 5.1.2). We build a relational database to hold, for example, the following: i) physical layer components, their infrastructural properties, their membership in MD, and the policy governing them; ii) virtual layer components, their membership within ODs, associated user properties; and iii) CMD and COD policies.

We are planning to provide two interfaces to interact with *nova-database* via *nova-api*: the first is related to managing users' properties and the second is for managing Clouds infrastructural properties and policies. At this stage the infrastructure properties would be provided by administrators. These data should only be accessed and managed using Cloud server agent. Future implementation will consider the utilization of Cloud client agents to collect such properties and securely push them to *nova-database*.

Nova-schedule is the central component of our scheme which controls the hosting of VMs at physical resources. Current implementations of *nova-schedule* do not consider the entire cloud Infrastructure neither they consider the overall user and infrastructure properties.

Once we finish from the above, we are planning to extend the proposed scheme framework from providing secure and trustworthy distributed environment to provide policy management (using the server agent) by proposing mechanisms which matches user properties with infrastructure properties). In this case the policy enforcement will be enforced by client agents.

5.1.9 Discussion and Analysis

In this section we discuss the advantages of the proposed framework architecture and how it achieves our objectives identified in section 5.1.3.

5.1.9.1 Benefits of Using Trusted Computing

We use the 'remote attestation' concept in trusted computing, which provides the ability to remotely attest to the execution environment of running software agents (i.e. server and client agents). It also binds the release of domain credentials to the attested trusted environment. This provides an 'offline' assurance that agents are behaving as expected. Such assurance establishes *a chain of trust* between Cloud users and Cloud providers as follows: 1) Cloud user trusts Cloud server agent to enforce user properties; and 2) Cloud server agent trusts Cloud client agent to enforce both user properties and infrastructure properties. These assure users that Cloud client agents can enforce user properties at the infrastructural level, without getting involved into infrastructural complexities. As we indicated in section 5.1.3 and our assumption (provided in 5.1.9), at this stage we do not consider attacks at physical or hypervisor levels and are a planned future research.

5.1.9.2 Benefits of the Framework Architecture

The proposed framework architecture adds the following benefits: (a.) Cloud server agent delegates MD/CMD policy enforcement (which manages the hosting of OD/COD) to Cloud client agents. If all resources must be fully managed all the time by a centralized management unit (i.e. VCC) this could be subject to single point of failure and would also raise performance concerns; (b.) The proposed framework supports domains with special properties (e.g. expandability, changeability of member devices, and collaboration with other domains). As the Cloud environment is dynamic and complex, this feature will be extremely helpful in satisfying Cloud properties and management requirements defined earlier; and (c.) Secure domains enable controlled and secure sharing of management data between members of MD, CMD, OD and COD. This helps in mitigating insider threats as described in next subsection.

5.1.9.3 Mitigating Insider Threats

Protecting Cloud management data from Cloud insiders is achieved as follows (as we discussed earlier we do not focus on hypervisor security threats and physical security threats in this work – also see scheme limitations at the end). If an authorized or unauthorized insider sends content from a device member of MD/CMD to unauthorized user the content stay protected and the unauthorized user will not be able to access content on any device not member of MD/CMD. This is because client agents are trusted to not reveal protected content to others. Thus, if an insider transfers protected content to another device not member of MD, the receiver will not be able to access the protected content as the receiver does not possess a copy of the content decryption key.

If an insider attempts to send a copy of the MD key to unauthorized users they will fail to do so. This is because the server and client agents are the only entities authorized to access the key after verifying the execution environment state of the device matches the one associated with the keys. In other words these keys are sealed to be only used by a trusted application, which is implemented to not reveal the keys in the clear even to system administrators.

If an insider attempts to add unauthorized device (e.g. less secure) to MD they will fail to do so. This is because system administrators explicitly identify member devices of a domain by adding their public keys in a domain-specific public key list. This means only predefined devices can join a domain. Therefore, unauthorized devises will fail to join a domain as their public keys are not listed in the PKL and, so, they will not be able to get a copy of the domain key.

From the above we can see that the proposed scheme enables controlled transfer of management data between devices. Simultaneously, such data can only be accessed by devices, which are authorized by security administrators. The same discussion applies to all other types of domains, but to address different security threats. For example, an insider might add an insecure MD to CMD, migrates OD resources to the insecure MD, and then leaks content. Our scheme addresses such threat by controlling the members of CMD using user-specific policy as discussed earlier.

5.1.9.4 Limitations

Our framework relies on trusting the administrators to implement and manage the proposed scheme; e.g. manage public key list. For this point we have the following comments: (a.) In our scheme we decrease the need to trust all employees, to the need to trust a very small and specific group of employees, which could be selected senior administrators. (b) As long as the administrators are the only party who are allowed to manage the scheme, and secure auditing mechanisms are in place, these will act as deterrent measures. This is because if any breach happens administrators will be the first to be asked, and their detection will be easier; and (c.) We have to acknowledge that in any scheme there should be a starting point of trust, which eventually should go to trusting humans. In our scheme we could add restrictions on the administrators which lessen their privileges (e.g require the presence of N out of M of administrators to perform any administrative activity).

5.1.10 Related Work

Many researches on Cloud focus on independent structural components. Research on Cloud's structural components and their security existed long-time ago before the term 'Cloud Computing' and are well established research areas; however, such areas still have unresolved problems which are inherited to the Cloud [Abb11c]. The work on [BSP+10b], for example, focuses on auditing firewall rules for multi-tier architectural components to assure users that their environment is protected; however such work still does not provide Cloud users the ability to control their own resources at the Cloud neither it assures them about the trustworthiness of Cloud environment.

The issue of establishing trust in the Cloud has been discussed by many authors (e.g. [Aba09, HRM10, HNB11, KM10b, SMV+10]). Much of the discussion has been cantered around reasons to "trust the Cloud" or not to. Khan and Malluhi [KM10b] discusses factors that affect consumer's trust in the Cloud and some of the emerging technologies that could be used to establish trust in the Cloud including enabling more jurisdiction over the consumers' data through provision of remote access control, transparency in the security capabilities of the providers, independent certification of Cloud services for security properties and capabilities and the use of private enclaves. The issue with jurisdiction is echoed by Hay et al [HNB11], who further suggest some technical mechanisms including encrypted communication channels and computation on encrypted data as ways of addressing some of the trust challenges. Schiffman et al [SMV+10] propose the use of hardware-based attestation mechanisms to improve transparency into the enforcement of critical security properties. The work in [Aba09, HRM10] focus on identifying the properties for establishing trust in the Cloud; however, these only considers partial operational services properties and they do not consider how trust values could be established.

In addition to the above, OpenStack discusses the importance of providing management

services that can automatically manage the allocation of VMs at physical layer considering the properties of the whole Cloud infrastructure. This work do not discuss the issue of trust establishment, neither they discuss the provision of assurance of the enforcement of the management services rules at physical resource. Moreover, this work still not developed yet by OpenStack community.

Our work differs from previous research in that we define a Cloud management framework and identify the requirements which form the foundation for providing Cloud users with the capabilities to control their resources and establishing trust in the operation of the Cloud without getting user to be involved into infrastructure management.

5.1.11 Conclusion

This work discusses an important topic in Clouds computing which has not yet received considerable attention. It proposes a framework for establishing trust in Cloud's operational management. The objectives of the framework is to establish the foundation for future work in providing trustworthy self-managed services that can automatically and with minimal human intervention manage Clouds users resources at physical resources based on user and infrastructure properties. In this work we specifically focus on providing protocols for the secure and trustworthy management of Clouds components. In addition we discussed how Cloud users can establish an 'offline' chain of trust in the operation of the Cloud infrastructure.

5.2 PKI Management for OpenStack via TrustedObjectsManager

OpenStack already comprehends interfaces allowing SSL encrypted communication for

1. the transfer of commands from an OpenStack Controller to other OpenStack Nodes and
2. the envisaged live time migration of virtual machine instances from one node to another

although any communication is unencrypted by default. In the latter case, the transfer of the virtual machine image itself is currently not encrypted. Furthermore, the compute nodes directly exchange plain libvirt commands in order to steer the virtual machines. The main PKI management component for OpenStack is envisaged to be the TrustedObjectsManager (TOM). It provides the central Public-Key-Infrastructure (PKI), serves as the main Certification Authority and manages all OpenStack nodes in terms of authenticity and accessibility. This chapter describes the envisaged functionality of the PKI applied to the TOM and the impact to an existing OpenStack infrastructure.

5.2.1 PKI management for an OpenStack installation by the TOM

In principle, all security related information (keys, certificates and probably configuration settings) are exchanged via the TrustedChannel, which itself is a secured connection between the TOM and an OpenStack instance (Figure 5.6). Initially, this is the only connection a newly setup OpenStack node is allowed to establish. This newly installed OpenStack-instance has to be pre-registered, based upon a platform-identifier (P-ID), at the TOM's management interface. In case the platform owns a Trusted Platform module (TPM) this identifier is created by and

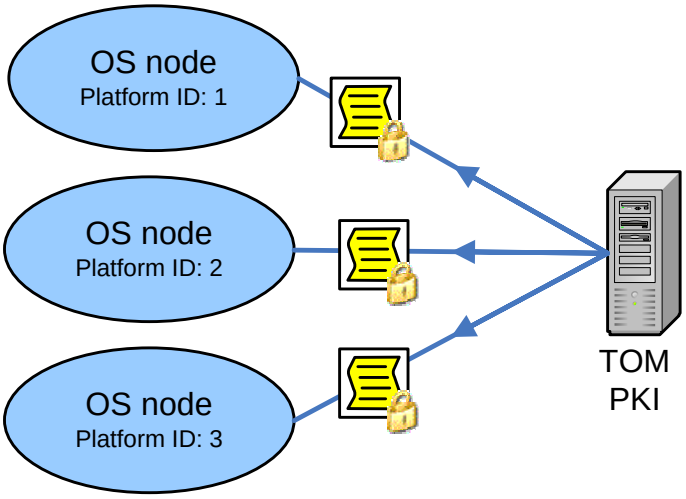


Figure 5.6: TrustedObjectsManager acting as PKI management component for OpenStack nodes

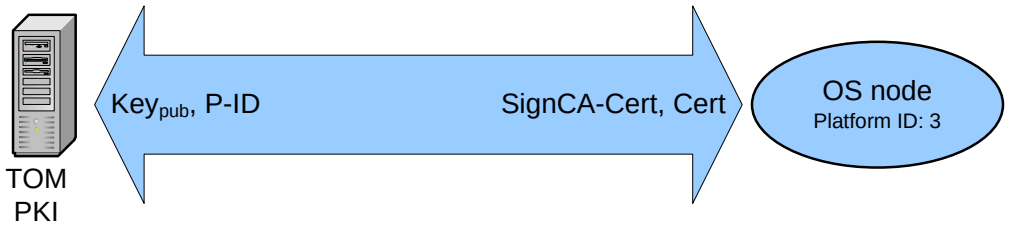


Figure 5.7: Public Key and Platform ID sent to the TOM, Signing-CA-Certificate and Node-Certificate sent back via the TrustedChannel

stored within the TPM. Otherwise, a software based platform identifier is calculated, using the nodes MAC- and IP-address, hostname or similar as input.

Once the node is pre-registered at the TOM, and the platform-ID matches the one sent by the node, the TrustedChannel connection will be established. After that, a public-private keypair will be generated on the node. The key's public part is sent to the TOM, where it will be signed by the Certification Authority. The signed certificate along with the Signing-CA-Certificate of the TOM is sent back to the node via the TrustedChannel. Both certificates will persist on the nodes' harddisk, allowing a secure connection between this node and the TOM. See figure 5.7

Thus, distributing the TOM-signed certificates to the nodes within the OpenStack landscape allows a secure communication between them without interacting with the TOM again.

This mechanism can be used to ensure a private communication between any OpenStack Controller and other OpenStack nodes via the RabbitMQ (or other AMPQ-solution). Furthermore, the secure live time migration of virtual machine instances as stated in 1. could be realized.

Since all OpenStack components can potentially be distributed logically as well as physically, the mentioned platform identifier (keys and certificates) is not enough to isolate the communication between virtual machines, belonging to different OpenStack projects . Therefore the concept of TrustedVirtualDomains (TVD) is introduced. These allow the deployment of isolated virtual infrastructures upon shared computing and networking resources. (See Chapter D.2.3.1 Chapter 6.3)

By default, different TVDs are isolated from each other. Remote communication between

components of the same TVD over an untrusted network is encrypted. The TOM provides a logical object "Company", which allows to group and sort users, TVDs, virtual machines (compartments), appliances and their relationship in detail. Thus the TOM's company-object reflects almost all parts of an OpenStack project. A company and all of it's ingredients (i.e TVDs) have to be defined beforehand, in order to be recognized and finally used by the associated nodes. When OpenStack facilitates a new project, and an enclosed new virtual machine, these informations can be mapped to a newly created company (=project) and a TVD, the virtual machine belongs to.

Now, there are mainly two possibilities to synchronize the mapping.

- The project and virtual machine configuration has to be sent to the TOM after creation in OpenStack. Therefore the involved OpenStack components could be hooked in that way, that an https-request can be sent to the TOM, which applies the company (=project) and the TVD. To avoid the https-request, the TrustedChannel can be extended with administrative capabilities, so that companies and involved TVDs can be created/deleted directly on the TOM during the creation of projects and/or virtual machines within OpenStack.

- When any virtual machine is started via OpenStack, an additional OpenStack plugin on the executing node queries the company/project and TVD informations from the TOM and applies these on the virtual machine. The unique identification of the virtual machines can be realized by the existing OpenStack-IDs.

5.2.2 Conclusion

Although the various underlying components of OpenStack support the usage of certificates and encryption keys for ensuring authentication, confidentiality and integrity, the cryptography is not enabled in the default installations of OpenStack. The main reason is that the management and distribution of certificates and keys are missing. This leaves OpenStack vulnerable to insider attacks where, for instance, rogue management operations can be injected into the cloud infrastructure and virtual machines can be infected by malware while being migrated.

In this work we outlined a proposal for solving this shortcoming in OpenStack by integrating an existing PKI solution, i.e., the TrustedObjectManager, in OpenStack. This forms the foundation for implementing and enabling: i) confidentiality and integrity during virtual machine migration; ii) authentication among compute nodes when issuing management operations in a peer-to-peer fashion (e.g. for VM migration); iii) secure communication within the OpenStack management infrastructure.

5.2.3 Future work

It is envisaged to extend the described one-to-one mapping from an OpenStack project to a company, containing one virtual machine, which belongs to exactly one TVD. Since there are imaginable scenarios, where the communication between virtual machines within the same project/company should be segregated, an expansion to a multiple TVD/project concept will be examined. Furthermore it is planned to integrate the web-interfaces provided by the TOM to OpenStack or vice versa. Provision is made for establishing a virtual private network (VPN) between virtual machines, belonging to the same TVD.

5.3 Distributed Quota Enforcement

5.3.1 Introduction

Cloud computing is considered a fundamental paradigm shift in the delivery architecture of information services, as it allows to move services, computation, and/or data off site to large utility providers. This offers customers substantial cost reduction, as hard- and software infrastructure needs not to be owned and dimensioned for peak service demand. With Platform-as-a-Service (PaaS) clouds like Windows Azure [Win] and Google App Engine [Goo] providing a scalable computing platform, customers are able to directly deploy their service applications in the cloud. In the ideal case, cloud customers only pay for the resources their applications actually use; that is, "... pricing is based on direct storage use and/or the number of CPU cycles expended. It frees service owners from coarser-grained pricing models based on the commitment of whole servers or storage units." [Cre09]

While it is very inviting to have virtually unlimited scalability and pay for it like electricity and water, this freedom poses a serious risk to cloud customers: the use of vast amounts of resources, caused, for example, by program errors, attacks, or careless use, may lead to high financial losses. Imagine an unforeseen input leads to a livelock that consumes massive amounts of CPU cycles. The costs for the resources used unintentionally could be tremendous and may even exceed the estimated profits of running the service.

To address this problem, we propose to employ a quota-enforcement service that allows cloud customers to specify global quotas for the resources (e. g., CPU, memory, network) used by their applications. Such a service can be integrated with the cloud infrastructure in order to ensure that the combined usage of all processes assigned to the same customer does not exceed the upper bound defined for a particular resource.

In domains like grid computing, where application demands are predictable, enforcing global quotas can be done statically during the deployment of an application [Sch04]. However, for user-accessed services in a dedicated utility computing infrastructure [RCAM06] like a PaaS cloud, this problem needs to be solved at run time once previously unknown services get dynamically deployed. Further, the quota-enforcement service must not impose any specific usage restrictions: processes must be able to freely allocate resources on demand as long as free quota is available. In this respect, the enforced global quota can be compared to a credit-card limit, which protects the owner from overstepping his financial resources while not making any assumptions on when and how the money is spent. All in all, dealing with a dynamically varying number of processes with unknown resource usage patterns makes quota enforcement a challenging task within clouds.

The straight-forward approach would be to set up a centralized service that manages all quotas of a customer and grants resources to applications on demand. However, as shown in our evaluation, such a service implementation does not scale for applications comprising a large number of processes, which is a common scenario in the context of cloud computing. Moreover, additional mechanisms like, for instance, state-machine replication had to be applied in order to provide a fault-tolerant and highly available solution. Otherwise, the quota-enforcement service would represent a single point of failure.

To avoid the shortcomings of a centralized approach, we devised a decentralized quota-enforcement service including a novel protocol named *Diffusive Quota Management Protocol*, short *DQMP*. DQMP is fault-tolerant and highly scalable by design, two properties that are indispensable for cloud environments. Its basic idea is to use the concept of diffusion to balance information about free quotas across all machines hosting a certain application of a customer.

By distributing quota information, the permissions to allocate resources can be granted via local calls. Our service offers a simple and lightweight interface that can be easily integrated to extend existing infrastructures with quota-enforcement support. An evaluation of our prototype with up to 1,000 processes residing on 40 machines shows that DQMP scales well and outperforms a centralized solution.

The remainder of this chapter is structured as follows: Section 5.3.2 discusses related approaches, Section 5.3.3 presents the architectural components of our quota-enforcement service, Section 5.3.4 outlines the concept of diffusive quota enforcement and presents the DQMP protocol, Section 5.3.5 presents results gained from an experimental evaluation of our prototype, and Section 5.3.6 concludes.

5.3.2 Related Approaches

Whereas earlier work on diffusion algorithms and distributed averaging addressed various areas such as dynamic load balancing [Cyb89, Boi90, CLZ99], distributing replicas in unstructured peer-to-peer networks [UOI06], routing in multihop networks [TE90] and distributed sensor fusion [XBL05], none of them handles quota enforcement. Karmon et al. [KLS08] proposed a quota-enforcement protocol for grid environments that relies on a decentralized mechanism to collect information about free resource quotas as soon as an application issues a demand. In contrast, our protocol proactively balances such information over all machines serving a customer, which allows granting most demands for free quota instantly. Furthermore, this chapter goes beyond [KLS08] in extending fault tolerance and in discussing how to integrate with cloud computing. Raghavan et al. [RVR+07] proposed an approach targeting distributed rate limiting using a gossip inspired algorithm in cloud-computing environments. They specifically focus on network bandwidth and neglect fault tolerance. Pollack et al. [PLG+07] proposed a micro-cash–inspired approach for disk quotas that provides lower overhead and better scalability than centralized quota-tracking services. A quota server acts as a bank that issues resource vouchers to clients. Clients can spend fractions of vouchers to allocate resources on arbitrary nodes of a cluster system. For good resource utilization and to prevent overload of the quota server bank, this requires previous knowledge about the resource demand. Gardfjäll et al. [GEE+08] developed the SweGrid accounting system that manages resources via a virtual bank that handles a hierarchical project namespace using branches. Based on an extended name service, each branch can be hosted on a separate node. This approach requires explicit management to be scalable and misses support for fault tolerance. Furthermore, there are distributed lock systems [HKS+08, Bur06] that provide fault-tolerant leases based on variants of the Paxos algorithm. Contrary to the presented approach, they are dedicated to manage low volume resources like specific files. As shown by the evaluation, our decentralized protocol scales above such replicated service solutions.

5.3.3 Architecture

In this section, we present the key components of our quota-enforcement service which is realized on basis of DQMP and explain how these components interact with existing cloud infrastructures.

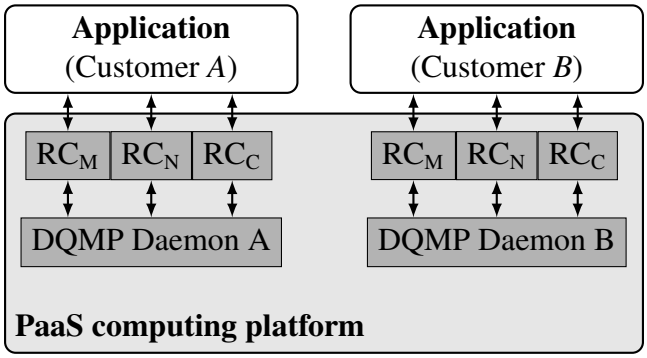


Figure 5.8: Basic architecture of a PaaS cloud host running DQMP to enforce resource quotas: quota requests issued by applications of different customers are handled by different DQMP daemons relying on a set of resource controllers (e. g., for memory usage (RC_M), network transfer volume (RC_N), and CPU cycles (RC_C)).

5.3.3.1 Host Architecture

DQMP uses a decentralized approach to manage the resource quota of customers. It distributes information about free quota units across the machines running applications of the same customer, providing each machine with a *local quota*. Quota enforcement in DQMP spans two levels: (1) At the host level, a *resource controller* guarantees that the local resource usage of an application process does not exceed the local quota. (2) At the global level, a network of *DQMP daemons* enforces a *global quota* by guaranteeing that the sum of all local quotas does not exceed the total quota for a particular resource, as specified by the customer.

Figure 5.8 shows the basic architecture of a PaaS cloud host that relies on our protocol to enforce quota for two customers *A* and *B*. For each of them, a separate DQMP daemon is running on the host. Each DQMP daemon is assigned a set of resource controllers (RC_*) which are responsible for enforcing quotas for different resource types (e. g., memory, network, and CPU).

Resource Controller In general, PaaS computing platforms provide means to monitor the resource consumption of an application process [WB09]. For DQMP, we extend these mechanisms with a set of resource controllers, one for each resource type. Each time an application seeks to consume additional resources, the corresponding resource controller issues a resource request to its local DQMP daemon and blocks until the daemon grants the demand.

DQMP Daemon A cloud host executes a separate DQMP daemon for every customer executing at least one application process on the host; that is, a DQMP daemon serving a certain customer is only executed on a host when there actually runs a process that may demand resource quota. The main task of a DQMP daemon is to fulfil the resource demands of its associated resource controllers. To do so, the daemon is connected to a set of other DQMP daemons (assigned to the same customer) that run on different cloud hosts, forming a peer-to-peer network. For the remainder of this chapter, we will refer to daemons connected in a DQMP network as *nodes*. Moreover, the first node that joins the network is called *quota manager*. It serves as a stable access point for the infrastructure, since the composition of a DQMP network is dynamic as nodes join and leave depending on whether their local machines currently host processes for the customer.

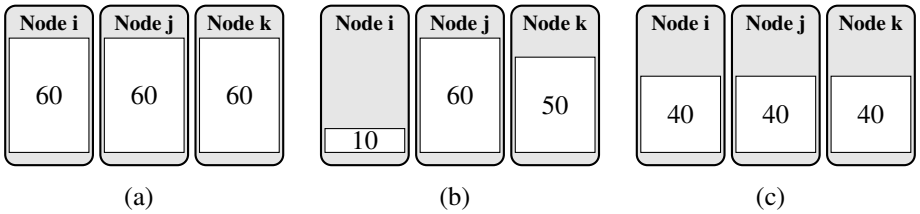


Figure 5.9: Example scenario for diffusion-based quota balancing: (a) The local free quotas are balanced across nodes. (b) Processes on nodes i and k demand resources $\rightarrow$ the diffusion of quota starts. (c) The free quotas have been rebalanced.

5.3.3.2 Node Registry

In addition to the DQMP components running on the same hosts as the customer applications, we provide a *node-registry* service that manages information about all nodes (i. e., DQMP daemons) assigned to the same customer. We assume the node registry to be implemented as a fault-tolerant service; for example, by using multiple registry instances. When a new node joins the DQMP network, the registry sets up an entry for it. As each node periodically sends a heartbeat message the registry is able to garbage collect entries of crashed nodes. When a node leaves the DQMP network (e. g., due to the last local application process having been shut down), the node instructs the registry to remove its entry.

5.3.4 The DQMP Protocol

This section presents the algorithms used by our decentralized quota-enforcement protocol DQMP to enforce global resource quotas of customers. In addition to a description of the basic protocol, we also discuss extensions for fault tolerance.

5.3.4.1 Diffusion-based Quota Balancing

We give a basic example scenario to outline how the general concept of diffusion is applied to balance free global quota information. In this example, three machines have been selected to host the application of a customer. For simplicity, we examine the diffusive balancing process of a single resource quota.

Each node (i. e., DQMP daemon) in the DQMP network is connected to a set of *neighbour nodes* (or just "*neighbours*"). Quota balancing is done by pairwise balancing the free local quota of neighbours. As neighbour sets of different nodes overlap, a complete coverage is achieved. In our example (see Figure 5.9), nodes i and j form a pair of neighbours, and nodes j and k form another pair of neighbours. At start-up, the global quota of the customer (180 units in our example) is balanced over all participating nodes (see Figure 5.9(a)).

When the application starts executing, the resource controller at node i demands 50 resource units and the resource controller at node k demands 10 units. Figure 5.9(b) shows that nodes i and k react by reducing the amount of locally available free quota q. Thus, both nodes can grant their local resource demands immediately. Changing the amounts of free quota starts the diffusive quota-balancing process and causes nodes i and k to exchange quota information with other nodes; in this case node j. As the free quota of node j exceeds the free quota of node i (i. e., $q_j > q_i$), $\lceil \frac{q_j - q_i}{2} \rceil$ quota units are migrated to i. The same applies to nodes j and k which, again, leads to different amounts of free quota on nodes i and j. As a result, further balancing processes are triggered and balancing continues until equilibrium is reached. The equilibrium

```
def initial_connect (nodes):
  for node in nodes:
    if node.connect( self ):
      neighbors.append(node)
      if level == None
        or level > node.↩
            level:
        level = node.level + ↩
            1
        uplink = node

def connect(node):
  if node not in neighbors:
    neighbors.append(node)
    return true
  return false
```

Figure 5.10: Connecting nodes

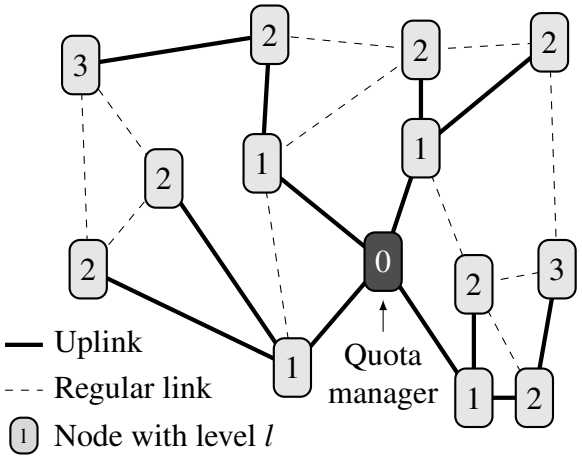


Figure 5.11: Example tree in a DQMP network

(see Figure 5.9(c)) enables node i to be well prepared for future resource demands, as its amount of free quota has risen to the global average of 40.

In case a resource controller issues a resource demand that exceeds q, a node obtains the requested quota by successively reducing q after each balancing process. As soon as the node has collected the full amount, it grants the resource demand to the resource controller.

Using discrete quota, there might be an imbalance of one unit between two neighbouring nodes if $mod(\sum q, n) \neq 0$ (n is the total number of nodes), causing balancing to never stop. To avoid this, we restrict balancing to differences above one unit. As a result, this introduces a potential system-wide gradient, which we cope with using probabilistic migration [DH04]. This strategy migrates small amounts of quota with a certain probability, even if the imbalance is not reduced.

5.3.4.2 Basic Protocol

This section describes the basic DQMP protocol. We assume a fail-stop behaviour of nodes and the reliable detection of node and connection failures.

Connection Process When a new application is deployed, our quota-enforcement service starts local DQMP daemons on the corresponding hosts and selects one of these nodes to be

Field	Description
level	Level in the quota tree
neighbours	List of neighbours, where each entry is a triple of **connection**, **counter**, and **level**
quota	Available local resource quota
consumed	Consumed resource quota (see Section 5.3.4.3)

Table 5.1: Data structures managed by a DQMP daemon

```
def do_balancing():
  for n in neighbors:
    free = quota
    # ask other node how to change my quota
    quota += n.balance(free)
```

```
def balance(remote_free):
  free = quota
  avg = (free + remote_free) / 2
  quota += avg - free
  return -(avg - free)
```

Figure 5.12: Simplified quota balancing process

the quota manager (see Section 5.3.3.1) Then, our service supplies all nodes of this first set with the addresses of all other nodes. Next, each node establishes a connection to some of the other nodes, adding them to its neighbour set (see `initial_connect()` in Figure 5.10 and Table 5.1).

During this procedure, every node determines its *level* in a tree (see Figure 5.11) that is formed as a by-product of the connection process. At first, only the quota manager (representing the tree root) is part of the tree and is therefore assigned level zero. Next, all other nodes join the tree using the following algorithm: (1) A node collects the level information of all of its neighbours. (2) It selects the neighbour n that has the lowest level l_n (i. e., the node with the smallest distance to the tree root) to be its parent node in the tree. From now on, we refer to the connection to n as the *uplink*; in Section 5.3.4.3, we investigate how the uplink is used to provide fault tolerance. (3) The node sets its own level to $l_n + 1$.

When a node has connected a predefined number of neighbours, it sends an announcement including its contact details and level information to the node registry managing a list of nodes assigned to the customer (see Section 5.3.3.2). In case the application of a customer scales up capacity by starting processes on additional hosts, newcomers query the node registry for addresses of nodes in the DQMP network. This information is then used as input for `initial_connect()`.

Quota Balancing When the set of initial nodes is connected, nodes can be provided with quota by simply initializing the quota manager's local free quota with the amount of globally granted quota. In consequence, the diffusion process starts and every node balances its free quota with all connected neighbours.

Figure 5.12 outlines the basic balancing process, organized in rounds, each comprising a single call to `do_balancing()`. During a round, for each neighbour, a node d determines the amount of free quota and sends it to the neighbour via `balance()`. This method adjusts the free quota at the neighbour and returns the amount by which to change the local free quota of d. The round ends when d has balanced quota with each of its neighbours. Note that quota balancing with a neighbour only takes a single message round-trip time.

If the local free quota has changed during a round of balancing, a node immediately starts another round. Otherwise, the next round is triggered when the node receives a demand from a local resource controller or when the quota exchange with another node modifies the local free quota.

5.3.4.3 Extension for Fault Tolerance

In this section, we describe how to extend the basic protocol presented in Section 5.3.4.2 in order to tolerate node failures.

General Approach To handle faults, every node maintains a counter for each neighbour link. This link counter represents the net amount of free quota transferred to the neighbour and is

```
def fix_crashedNode(neighbor):
    quota += neighbor.counter
    neighbors.remove(neighbor)
    # check if uplink is concerned
    replace_crashedNode()
```

Figure 5.13: Recovery after neighbour crash

```
def do_balancing():
 for n in neighbors:
   free = quota
   if n.level < level:
     # pass the consumed quota
     # up to the root
     n.counter += consumed
      result = n.balance(id, free,
                         consumed)[0]
     consumed = 0
   else:
     # receive consumed quota
     # from lower nodes
     (remote_consumed, result) =
                 n.balance(id, free)
     n.counter -= remote_consumed
     consumed += remote_consumed

   n.counter -= result
   quota += result
```

Figure 5.14: Issuing a balancing request

```
def balance(id, remote_free,
            remote_consumed = 0):

 neighbor = neighbors[id]
 free = quota
 avg = (free + remote_free) / 2

 # handle the consumed quota
 if neighbor.level < level:
   remote_consumed = consumed
   neighbor.counter += remote_consumed
   consumed = 0
 else:
   neighbor.counter -= remote_consumed
   consumed += remote_consumed

 # balance the remaining quota
 if remote_free < 0 and free < 0:
   # nothing left on both sides
   return (remote_consumed, 0)
 elif remote_free < 0 or free < 0:
   # take care of negative quotas
   # [...]
 else: # free quota on both sides
   quota += avg - free
   neighbor.counter -= avg - free
   return (remote_consumed,
           -(avg - free))
```

Figure 5.15: Responding to a balancing request

updated on each quota exchange via the corresponding link: if a node passes free quota to a neighbour, it increments the local link counter by the amount transferred; the neighbour decrements its counter by the same amount. A negative counter value indicates that a node has received more free quota over that link than the node has passed to the neighbour.

When a node crashes, all connected neighbours detect the crash: each neighbour removes the crashed node from its neighbour set and adds the counter value of the failed link to its local amount of free quota (see Figure 5.13). This way, the free quota originally held by the crashed node is reconstructed by all neighbours, requiring no further coordination. Note that such a recovery may temporarily leave single nodes with negative local free quota. However, the DQMP network compensates this by quickly balancing quota among remaining nodes.

Consumed Quota So far, this approach is only suitable for refundable quota like disk space, since link counters are unaware that non-refundable quota, like CPU cycles, transferred to a node may have been consumed by a local application process. Thus, neighbours would reassign more free quota than the crashed node actually had. To address this, nodes gather and distribute information about consumed quota, and adjust their link counters to prevent its reassignment.

For each resource, a node maintains a `consumed` counter (see Table 5.1) that is updated

whenever a local application process consumes quota. Each node periodically reports the value of its `consumed` counter to its uplink, which in turn passes it to its own uplink, and so on, all up to the quota manager. Having reported the consumed quota, a node increments its uplink link counter by the amount announced; the uplink in turn decrements its link counter by the same value, similar to the modifications triggered during quota balancing. As a result, link counters are adjusted to reflect the reduced global free quota. Figures 5.14 and 5.15 show updated listings of the balancing process presented in Figure 5.12.

Handling Cluster Node Failures Link counters are an easy and lightweight mean to compensate link crashes and single node failures. They also allow tolerating multiple crashes of directly connected nodes, because adjacent nodes can be seen as one large node with many neighbours. In case a node set is separated from the rest of the network, the node set that is not part of the quota-manager partition eventually runs out of quota, since free quota is always restored in the direction of its origin (i. e., the quota manager). However, after reconnection, the balancing process re-distributes the free quota, enabling the application processes on all nodes to make progress again. To avoid permanent partitions within the network, the protocol makes use of the level information. When a node except the quota manager and its direct neighbours loses the connection to its uplink, it has to select a node with a lower level than its own as new uplink. Preferably, the node uses one of its current neighbours for that purpose; however, it can also query the node registry (see Section 5.3.3.2) for possible candidates. If a suitable uplink cannot be found, the node is shut down properly.

Handling Crashes of the Quota Manager If the quota manager crashes, its neighbours do not consolidate their link counters. If they did, all global quota of a customer would vanish as it has been originally injected via the quota manager. Instead, all links to the quota manager are marked *initial links* and are therefore ignored during failure handling, allowing the network to proceed execution.

However, we assume a timely recovery of the quota manager as an application cannot be provided with additional quota while this node is down. We therefore assume that its state can be restored (e. g., using a snapshot). Note that the state of the quota manager to be saved is small: it only includes the set of neighbour addresses as well as the `quota`, `consumed`, and `counter` values (see Table 5.1) for every managed resource, making frequent snapshots and a fast recovery feasible.

At restart, the quota manager reconnects all level-one nodes. In case of one or more of them having crashed in the meantime, it starts the regular failure handling. At this point, we cannot tolerate network partitions between the quota manager and its neighbours, as this would lead to a duplication of free quota.

5.3.5 Evaluation

We evaluate DQMP on basis of a prototype implemented in Java. The tests are performed on 40 hosts, all equipped with 2.4 GHz quad-core CPU, 8 GB RAM, and connected over switched Gigabit Ethernet. Each host executes up to three Java virtual machines (JVMs) to support the simulation of larger networks. In this set-up, raw ping times range from 0.2 to 0.5 ms and simple Java RMI method calls take between 0.7 and 1.0 ms. On top of the physical network, two DQMP networks, consisting of 100 and 1,000 nodes, are simulated, with the maximum number of neighbours set to 6. Comparison measurements show, that simulating up to nine nodes within a single JVM has no significant impact on the results.

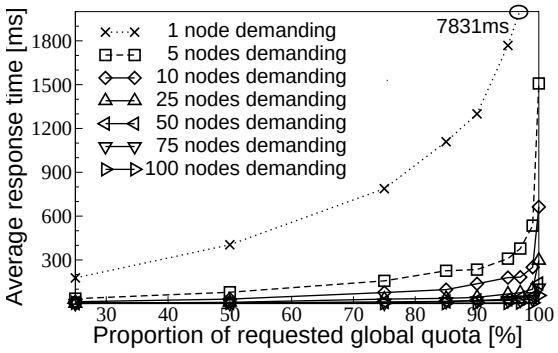


Figure 5.16: Response times for single demands of varying amounts from varying parts out of 100 nodes

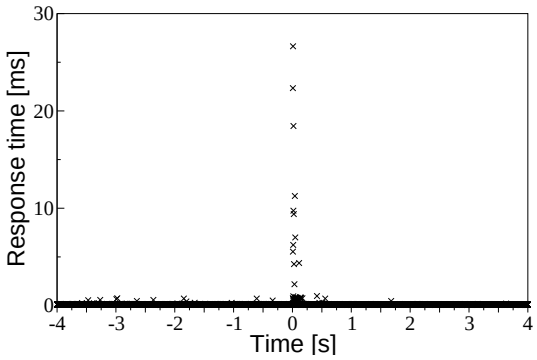


Figure 5.17: Response times of a single test run with 100 constantly requesting nodes and a crash of 25 nodes at $t = 0$

Test runs are performed as follows: After the DQMP network is built up, a quota amount of 50,000 units per node is injected. When the initial equilibrium is established, all nodes are instructed to begin with the execution of the actual test. After a test has finished, the local results of the nodes are collected. Except time charts, all presented results are the average of at least three test runs.

5.3.5.1 Response Time Behavior of DQMP

Single Demands In the first test, we examine the response times of DQMP for single demands within the small network containing 100 nodes. In this scenario, a subset of nodes orders a predefined amount of quota at the same time. The proportion of demanding nodes is raised stepwise from 1% to 100% and the overall amount of quota requested by this proportion is varied between 25% and 100%. This means that in one case, for instance, a single node requests the entire quota available and in another case, each of 100 nodes requests 1% of it.

From the results, as depicted in Figure 5.16, it can be inferred that the decisive factor for the performance of our protocol is the ratio between the free local quota held by each node and the size of the local demand: the smaller the demand compared to the local quota, the faster it can be satisfied. Since DQMP aims to an even distribution of free quota over all nodes, the demand size can be put into relation to the globally free quota: if demands of single nodes exceed the average size of free quota held by each node to a great extent, it is likely that quota has to be transferred not only from nearer nodes but also from farther ones to satisfy the demand. For instance, if a single node asks for the entire available quota, every quota unit in the network has to reach the same destination. With our settings, this takes about 7.8 seconds and 770 balancing rounds per node. However, this case is not realistic as only such nodes participate in DQMP networks that are actually used by processes demanding quota. If 50 nodes request 95% of the overall quota, the provisioning time already drops below 30 ms. Here, it takes about 45 balancing rounds per node until the request is fulfilled and until the network comes to a rest, that is, until no messages are transmitted anymore. Moreover, if only a small amount of the overall quota is needed or a large demand is split between many nodes, DQMP can provide extremely low response times. When a demand of a node can be fulfilled by its local quota, the DQMP daemon is even able to instantly grant the demanded amount, turning the assignment of global quota within a distributed system into a local operation.

Crashes of Nodes After this first evaluation, we now examine how our protocol behaves in the presence of node crashes, since fault tolerance was a primary objective for the design of DQMP. As basis for this evaluation, we choose a scenario in which nodes demand and release quota constantly. In detail, each node performs the following in a loop: It adds a randomly chosen delta d, with $-10,000 \leq d \leq +10,000$, to its previous quota demand. It ensures that the new demand does not exceed the upper bound b of $50,000$ units, which limits the demand of all nodes combined to 100% of the overall quota injected into the system. According to the calculated value, the node issues a request either demanding new or releasing already granted quota. Subsequently, it waits until the request is fulfilled. Then it sleeps for a randomly chosen time between 25 and 75 ms to simulate fluctuating resource requirements.

Figure 5.17 shows the course of response times from a single test run with 100 requesting nodes, issuing a total of approximately 14,000 requests within 8 seconds, and an induced crash of 25 nodes at $t = 0$. The first outcome of this test is, that under the given scenario, which simulates the distribution of a large demand over all available nodes, almost all quota requests can be fulfilled locally, leading to a standard response time below 0.2 ms. For the same reason, the processing of most requests is hardly affected by crashes of neighbours. Quota releases are inherently not affected at all anyway. Consequently, despite the crash of 25% of the nodes, there are only 4 requests for which it took between 10 and 30 ms to process them and 8 requests that lie in the range between 1 and 10 ms. Thus, the balancing process of DQMP is able to compensate node crashes very quickly by redistributing the quota over all remaining nodes.

5.3.5.2 Comparison of Different Architectures

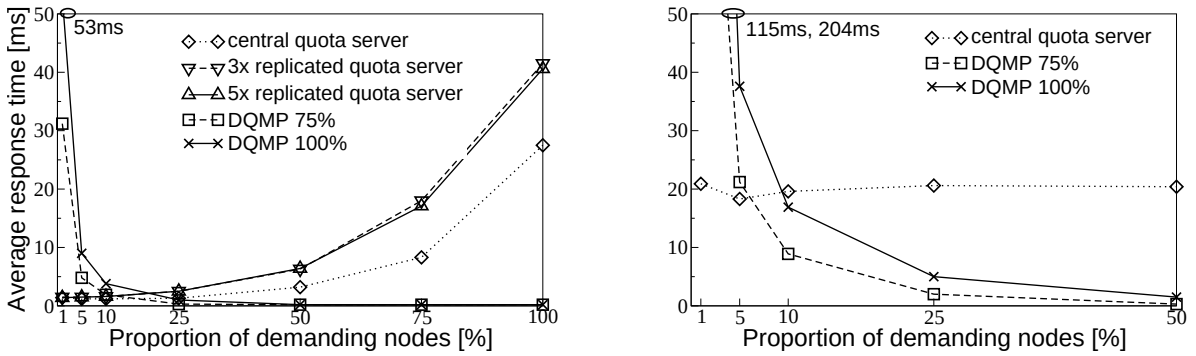
Next, we compare DQMP to other architectures addressing quota enforcement in distributed systems. For this purpose, we implemented a RMI-based quota server and a passively replicated variant of it by means of the group communication framework JGroups[4]. During test runs, the quota server as well as each replica is executed by a dedicated machine. In the following, the term "node" is not confined to DQMP daemons; it also denotes clients in the other architectures.[5]

As scenario for the comparison serves an extended variant of the scenario used for examining the behaviour of DQMP in the presence of nodes crashes (see Section 5.3.5.1). Different to the previous scenario, here, a network of 1,000 nodes is used and the proportion p of requesting nodes is varied between 1% and 100%. Further, the combined demand of all requesting nodes is limited to 75% of the overall injected quota in one case and to 100% in another. This is achieved by setting the maximum demand of a single node b to $b_{75\%} = \frac{37.500}{p}$ and $b_{100\%} = \frac{50.000}{p}$, respectively. The delta d for every simulated demand change is randomly chosen between $-0.2b$ and $+0.2b$ quota units.

Single-cluster Network For a first comparison, all network connections have similar latencies, just as in the previous tests and just as found within a local area network, for instance within a single data centre of a cloud provider. The results of this scenario are depicted in Figure 5.18(a). Since response times of the central quota server and its replicated variant are

[4] http://www.jgroups.org/

[5] We also implemented a quota-enforcement service based on the coordination service Apache ZooKeeper (http://zookeeper.apache.org/). However, the optimistic lock approach of ZooKeeper is not suitable for the high number of concurrent writes needed in such systems, resulting in some orders of magnitude higher response times.



(a) Single-cluster network with 1,000 nodes

(b) 10 clusters of 100 nodes

Figure 5.18: DQMP compared to other architectures regarding response times

only dependent on the number of quota requests that have to be processed, and particularly are independent of quota amounts, only a single set of results is reported for these architectures.

This test reveals the deficiencies of not completely decentralized systems in terms of scalability: Due to limited resources such as CPU power, memory and bandwidth and due to the contention arising from the shared usage of such resources, these systems have a limited rate they can process requests at. In our settings, for instance, all quota-server–based systems are able to process the requests of a smaller number of requesting nodes within less than 2 ms on average. However, in the presence of 1,000 requesting nodes, a single quota server already requires about 28 ms. Using a more reliable replicated server system makes this even worse. The increased communication overhead leads to an average response time of over 40 ms.

In contrary, using DQMP response times decrease when demands are split up between more nodes. DQMP is able to fulfil requests within an average of 1 ms, and is thus faster than the server systems when the proportion of requesting nodes exceeds 25%. Beyond 50% the response time drops constantly below 0.2 ms. Since the total demand was fixed to either 75% or 100% of the globally injected quota, single demands get smaller with an increasing number of requesting nodes, leading to a higher chance that requests can be fulfilled through the local quotas of the nodes. That is the reason why, as shown by our results, DQMP is even able to outperform a non-saturated central quota server in terms of average response times when demands are distributed over multiple nodes.

Clustered Network Normally, cloud providers do not maintain only a single data centre but multiple ones, spread all over the world. These data centres form a clustered network, a network in which groups of well-connected nodes can only communicate among each other over relatively slow connections. To simulate such an environment, respectively wide area networks in general, we assign each out of 1,000 nodes to one of 10 clusters and artificially delay message exchange between nodes from different clusters by 20 ms.

The results, as presented in Figure 5.18(b), suggest the conclusion that a central quota server is not well suited for the scenario described here. The server is located in one of the 10 clusters, which entails that 90% of all nodes experience prolonged delays while communicating with it. Thus, in 90% of all quota requests, demands or releases, the delay of 20 ms is fully added as an offset to the processing time. In case of DQMP, nodes can exchange quota with all of their neighbours in parallel, mitigating the effects of slower connections. Furthermore, all requests that can be fulfilled locally, including all releases, are not affected at all by communication

delays. These are the reasons, why DQMP is able to provide better response times than a quota server in this scenario already when only 10% of the nodes demand and release quota.

Protocol Overhead Concerning the protocol overhead of DQMP regarding network transfers, it can be observed that DQMP has completely different characteristics than a traditional quota server. If a quota server is used, each quota request leads to the exchange of two messages, a request message and its reply. In our implementation, the two messages require about 100 bytes. With DQMP instead, requests have only an indirect influence on the balancing process and hence, on the number of messages transferred. For the unrealistic case (see above) that relatively large demands are infrequently issued by a single node, causing, in the worst case, continuous balancing processes all over the network, the ratio between number of requests and messages transferred is unfavorable. With an increasing number of requests, however, the ratio gets more appropriate. In the scenario of 1,000 constantly requesting nodes our protocol requires about 3 kilobytes per request in average. Although this is still more than needed by the quota-server system, it has to be noted, that DQMP provides fault-tolerant operation while a central quota server does not and that network traffic between hosts of the same data centre is usually not billed by cloud providers, hence, using DQMP would not generate additional transfer costs for cloud customers.

5.3.6 Conclusion

In this chapter, we presented DQMP, a decentralized quota-enforcement protocol that provides the fault tolerance and scalability required by cloud-computing environments. DQMP can help customers of platform services, to prevent themselves from financial losses due to errors, attacks, or careless use causing involuntary resource usage. The utilized diffusion-based balancing of free quota enables customers to enforce global limits on resource usage while retaining flexibility and adaptability regarding the actual local demands within their deployments. Nonetheless, DQMP is not confined to this application. Cloud providers can employ it, for example, to restrict customers of their platform or infrastructure services on a global level by enforcing quota for virtual machines. As the evaluation of our prototype implementation shows, DQMP is able to provide better response times than a centralized service in a setting with 1,000 nodes. Moreover, our protocol is well suited for clustered networks as formed by interconnected data centres. Both is important since traditional, not fully decentralized solutions might soon reach their limit as distributed systems get larger and larger.

Chapter 6

Trust Anchors in Management Tasks

Chapter Authors:
Imad M. Abbadi, Andrew Martin, Anbang Ruan (OXFD)

Managing the allocation of Clouds virtual machines at physical resources is a key requirement for the success of Clouds. Current implementations of Cloud schedulers do not consider the entire Cloud infrastructure neither they consider the overall user and infrastructure properties. This results in a major security, privacy and resilience concerns. In this chapter we propose a novel Cloud scheduler which considers both users' requirements and infrastructure properties. We focus on assuring users that their virtual resources are hosted using physical resources that match their properties without getting users involved into understating the details of the Cloud infrastructure. As a proof-of-concept we present our prototype which is built on OpenStack. The provided prototype implements not only the proposed Cloud scheduler; however, it does also provide an implementation of our previous related works on Clouds' trust management which provide the scheduler with trustworthy input about the state of trust in the distributed Cloud infrastructure

6.1 Introduction

Cloud infrastructure is complex and heterogeneous in nature, with numerous components provided by different vendors [Abb11c]. Applications deployed at the Cloud might need to interact amongst themselves and, in some cases, depend on other deployed applications. The complexity of the infrastructure and application dependencies create an environment which requires careful management and raises security and privacy concerns [AFG+09, RTSS09b]. The central component that manages the allocation of virtual resources at physical resources at the Cloud infrastructure is known as Cloud scheduler [Ope10b]. Currently available schedulers do not consider users security and privacy requirements, neither they consider the properties of the entire Cloud infrastructure. For examples a Cloud schedulers should consider application performance requirements (e.g. the physical hosting of interdependent application components need to be within physical proximity), a scheduler should consider user privacy requirements, and a scheduler should manage the problem of the multi-tenant architecture [RTSS09b] and the trust status of the hosting components.

This work presents a "trustworthy scheduling algorithm" that can automatically manage the Cloud infrastructure by considering both user requirements and infrastructure properties and policies. The work also develops the required trustworthy software agents which automatically manage the collection of the properties of physical resources. Having a trustworthy and timely copy of the infrastructure properties and user requirements is critical for the correct operation of

the scheduler. This is a difficult problem to deal with, i.e. provide the scheduler with trustworthy input enabling it to take the right action. In this work we specifically focuss on providing the scheduler with trustworthy input about the trust status of the Cloud infrastructure. This work establishes the foundations of a planned future work to cover other properties.

OpenStack refers to the Cloud scheduler component using the name *nova-scheduler* [Ope11]. Openstack identifies nova-scheduler as the most complex component to develop, and lots of efforts are still remaining to have an appropriate nova-scheduler. A key requirement to develop the trustworthy scheduler component, in our opinion, is to have a critical understanding of how Clouds are managed and how they work in practice, which we have covered in our previous work [Abb11a, Abb11b]. In the previous work we concluded that establishing trust in Clouds requires two mutually dependent elements: a) supporting Cloud infrastructure with trustworthy mechanisms and tools helping Cloud providers to automate the process of managing, maintaining, and securing the infrastructure; and b) developing methods helping Cloud users and providers to establish trust in the operation of the infrastructure. Point (a) includes (but not limited to) supporting the Cloud infrastructure with trustworthy self-managed services which automatically manage Cloud infrastructure [Abb11a, Abb11c]. Automated self-managed services should 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 consider users' security and privacy requirements by design [Abb11a]. The proposed Cloud scheduler belongs to point (a) which is our first step in providing trustworthy self-managed services. Our previous work in [Abb11a, Abb12] partially covers point (b), which we also prototype and integrate to the proposed scheduler to present a coherent solution.

6.1.1 Organization

The chapter is organized as follows. Section 6.2 provides essential foundations which cover Cloud structure and management services. Section 6.3 discusses the Clouds compositional chains of trust. Section 6.4 presents our framework architecture and Section 6.5 presents our prototype. Section 6.6 discusses related work. We conclude in Section 6.7.

6.2 Background

In this section we briefly summarize previous work which this work builds on. Specifically, we highlight Cloud structure and management services.

6.2.1 Cloud Structure Overview

In this section we briefly highlight part of Cloud taxonomy (see [Abb11a] for further details). Cloud environment is composed of enormous *resources*, which are categorized based on their types and deployment across Cloud infrastructure. A *resource* is a conceptual entity that provides services to other entities. Cloud environment conceptually consists of multiple intersecting layers as follows: i) *Physical Layer* — This layer represents the physical resources and their interactions, which constitute Cloud physical infrastructure. Examples of these resources include server, storage, and network resources. The physical layer resources are consolidated to serve the *Virtual Layer*. ii) *Virtual Layer* — This layer represents the virtual resources, which are hosted by the *Physical Layer*. Examples of these resources include virtual machine (VM),

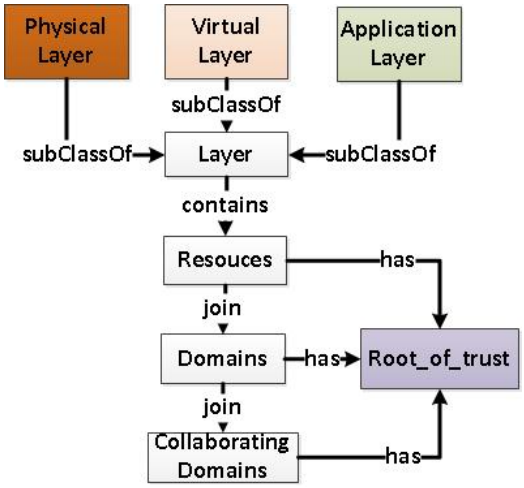


Figure 6.1: Cloud Computing — Layering Conceptual Model

virtual network, and virtual storage. Cloud customers in IaaS Cloud type interact directly with the resources of the Virtual Layer as it hosts the application of Clouds' customers. iii) *Application Layer* — This layer runs the applications of Cloud's customer. These are hosted using *Virtual Layer* resources. Cloud customers using PaaS type deploy their applications at virtual layer resources, while the customers of Cloud SaaS type access a deployed application via the Internet.

Figure 6.1 provides a conceptual model in which we identify an entity *Layer* as the parent of the three Cloud layers (i.e. the physical, virtual, and application layers). From an abstract level the *Layer* contains *Resources* which join *Domains* (i.e. we have physical domain, virtual domain, and application domain). A *Domain* resembles a container which consists of related resources. *Domain*'s resources are managed following *Domain* defined policy. *Domains* that need to interact amongst themselves within a layer join a *Collaborating Domain* (i.e. we have physical collaborating domain, virtual collaborating domain, and application collaborating domain). A *Collaborating Domain* controls the interaction between *Domain* members of the *Collaborating Domain* using a defined policy.

The nature of *Resources*, *Domains*, *Collaborating Domains*, and their policies are layer specific. *Domain* and *Collaborating Domains* concepts help in managing Cloud infrastructure, and managing resources distribution and coordination in normal operations and in incidents. *Collaborating Domains* communicate across Cloud layers to serve a collaborative customer application needs. *Domains* communicates horizontally within a layer-specific *Collaborative Domain*, and/or vertically across multiple layers' *Collaborative Domains*. Each of the identified Cloud entities has a root of trust which helps in establishing trust in Clouds. Subsequent sections clarify the roots of trust in more details.

6.2.2 Virtual Control Center

Currently there are many tools for managing Cloud's virtual resources, e.g. vCenter [VMw10] and OpenStack [Ope10b]. For convenience we call such tools using a common name Virtual Control Centre (VCC), which is a Cloud device[1] that manages virtual resources and their interactions with physical resources using a set of software agents. VCC will play a major role

[1]VCC (as the case of OpenStack) could be deployed at a set of dedicated and collaborating devices that share a common database to support resilience, scalability and performance.

in providing Cloud's automated self-managed services, which are mostly provided manually at the time of writing. Self-managed services should automatically and continually manage Cloud environment with minimal human intervention. It should always enforce Cloud user properties; e.g. ensure that user resources are always hosted using physical resources which have properties enabling such physical resources to provide the services as defined in user properties. The proposed Cloud scheduler covers a key component of such management services.

In previous work ([AAM11]) Abbadi et. al. proposed a framework for establishing trust in the operational management of Clouds. The framework identifies the challenges, requirements, and addresses the ones related to establishing secure and trustworthy environment at the infrastructure level. This includes the establishment of offline chains of trust amongst Clouds' entities. This work uses the framework for providing secure environment for collecting the infrastructure properties and enforcing the scheduler policies at client devices. The functions of the framework are provided using two types of software agents: a server software agent that runs at VCC and a client agent that runs at Cloud's physical resources. We refer to the server software agent of VCC as (Domain Controller Server Side, DC-S), and we refer to the client software agent as (Domain Controller Client Side, DC-C). DC-C is in charge of enforcing the DC-S policies at physical resources. DC-S establishes chains-of-trust with each DC-C as follows: the DC-S verifies each DC-C trustworthiness to continually enforce the domain policies and to only access the domain credentials when the resource execution status is as expected. In turn, DC-C provides assurance to DC-S about the trustworthiness of its hosting resource's execution environment when managing the domain and enforces the domain policies. This provides the assurance that only resources with a trustworthy DC-C can be member of a domain. See ([AAM11]) for detailed discussion of how offline chains of trust are established and assured.

6.3 Clouds Compositional Chains of Trust

One of the key properties of a Cloud infrastructure is its trustworthiness to manage users' virtual resources at physical resources as agreed in service level agreement (SLA). This is not only beneficial to Cloud users, but also help Cloud providers to understand how their infrastructure is operated and managed. However, assessing the trustworthiness of the Cloud infrastructure is a difficult problem to deal with considering its dynamic nature and enormous resources [Abb11c, AN11, AFG+09, BSP+10b, CGJ+09, JNL10, Mic09, RTSS09b]. As we discussed in section 6.6 some of the proposed schemes in this direction focus on measuring the trustworthiness of the overall Cloud infrastructure and others attempt to establish a chain of trust with a specific resource at a specific point in time. Abbadi ([Abb12]) analyzed this problem, and argues that it is impractical to measure the trustworthiness of the overall Cloud infrastructure (considering its complexity), neither we should measure the trustworthiness of a single component (as Cloud dynamism breaks any established chain of trust). Abbadi's method is based on segmenting the infrastructure and measuring the trustworthiness of each segment independently. The boundaries of each segment is based on how the infrastructure is managed in practice, as we discuss it earlier (i.e. a segment covers a domain entities). The domain and collaborating domain concepts enables the control of where an entity in layer (n) could possibly be hosted at layer (n-1). For example, a virtual machine can only be hosted within a collaborating physical domain boundaries.

Abbadi ([Abb12]) proposed the concept of *compositional chains of trust*, which provides a single chain of trust representing a group of entities, as many entities exist as a composition of multiple entities (e.g. a cluster of physical servers, clusters of load balanced application

and database servers). Members of such grouping may have identical or different chains of trust. However, to an entity depending on this grouping, they should see a single chain of trust representing the trust they have in the grouping. In other words, relying entities will see a single entity, even though that entity will be a grouping representing multiple entities. Moreover, the functions proposed to calculate the compositional chains of trust provide different levels of transparency based on the Cloud user type (i.e. IaaS, PaaS, or SaaS). Abbadi's paper do not provide a prototype. Our proposed scheduler uses his methods to measure the trustworthiness of the Cloud infrastructure, and as a result we provide a prototype of the related part of Abbadi's compositional chains of trust. This section covers the part which is only related to physical layer (other parts are not covered as our scheduler is not related to the application layer).

6.3.1 Types of Chains of Trust

A Chain of Trust (CoT) is composed of a set of elements primarily used to establish the trust status of an object. The first element of the CoT (also called the root of trust) should be established from a trusted entity or an entity that is assumed to be trusted, e.g. trusted third party, a tamper-evident hardware chip (as in the case of Trusted Platform Module (TPM) [Tru07a, Tru07b, Tru07c]). The trust status of the second element in the CoT is measured by the root of trust (i.e. the first element in the CoT). If the verifier trusts the root of trust, then the verifier must also trust the root of trust measurement of the second element. The second element then measures the trust status of the third element in the CoT. If the second element is trusted, and the second element measures the third element trust status, then the verifier trusts the measurements of the third element. This process is a simplified example of how a CoT could possibly be built.

Clouds have two types of CoT: a single resource CoT, and a compositional CoT representing multiple entities (i.e. domains and collaborating domains). A verifier is mainly interested in evaluating compositional CoTs without the need to get involved in understanding the details of Cloud infrastructure. The compositional CoT would be built on resources CoT. As a result this section defines both types of CoTs, which includes defining the nature of their roots of trust.

6.3.2 A Resource Chain of Trust

As stated earlier, a resource is a conceptual entity that provides services to other entities. Therefore, we begin the discussion by defining the CoT for a single resource (RCoT) as a triple comprising an initial trust function (itf), a set of trust functions (stf) and a sequence of elements in the chain ($sq. < x_0, x_1, .., x_n >$) where x is an element representing any component (software, hardware, etc.) that contributes to the chain of trust. RCoT requires the following:

1. The initial function evaluates to trusted or assumed to be trusted when applied to the first element of the sequence, and
2. Every function in the set of trust functions evaluates to true when applied to any two consecutive elements of the sequence.

This is formally defined as follows:

$$
\begin{aligned}
&RCoT = (\\
&\quad itf, stf, sq. < x_0, x_1, .., x_n > | \\
&\quad itf(x_0) \in \{trusted, assumed\_trusted\}, \\
&\quad \forall i : [1..n] \bullet \forall f : stf \bullet f(x_{i-1}, x_i) == true)
\end{aligned}
$$

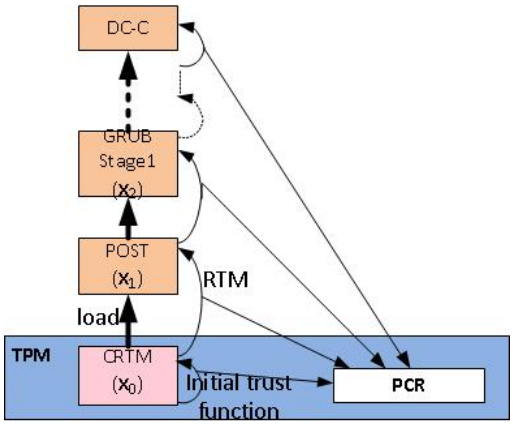


Figure 6.2: Example of a part of physical resource's RCoT

The nature of the *root_of_trust* (i.e. the first element in the sequence, x_0) is based on the type of the entity and its location within Cloud layers. We now discuss a single resource *root_of_trust* and subsequent part of this section covers compositional entities *root_of_trust*. We now clarifies RCoT in context of TCG specifications, as we require each resource within the physical layer to be TCG compliant fitted with a TPM which is physically bound to that resource. A TPM must be tamper-evident; i.e. provides a limited degree of protection against physical attack. TPM helps in providing three roots of trust: Root of Trust for Measurement (RTM), Root of Trust for Storage (RTS), and Root of Trust for Reporting (RTR). The RTM is a computing engine capable of making reliable measurements of Trusted Platform (TP) running components, which is known as an integrity measurement. Integrity Measurement is a cryptographic digest or hash of a TP component; i.e. a piece of software executing on a TP [PEC05]. The RTS is a collection of capabilities, which must be trusted if the storage of data in a TP is to be trusted [Pea02]. The RTS uses TPM components to achieve its functions. The RTR is a collection of capabilities that must be trusted if reports of integrity metrics are to be trusted (platform attestation) [Pea02]. The RTR works in conjunction with the RTM and the RTS for the implementation of platform attestation. The RTR enables a TPM to reliably report information about its identity and the current state of the TPM host platform. This is achieved using set of keys and certificates, which are signed by a variety of third parties that must be trusted if the state of the platform is to be trusted. In TCG the RCoT starts from the Core Root of Trust of Measurement (CRTM) which should be stored in protected location such as the TPM (currently it is protected by the BIOS). Once CRTM measures the initial environment state it stores the result in a protected registers inside the TPM (referred to as PCR). The CRTM represents the *root_of_trust*, x_0, and the Set.(trust_functions) contains RTM, RTS, RTR, and other functions. The *initial_trust_function* is the one that measures the CRTM itself and stores the result inside the TPM's PCR. Figure 6.2 illustrates these relations.

Unlike the RCoT at the physical layer, an RCoT at the virtual and application layers have different treatments when discussing a specific resource roots of trust. This is becuase physical resources are the foundation of virtual resources roots of trust, which in turn forms the foundation of application resources roots of trust. In other words, the virtual and application layers RCoT in Clouds context, considering its dynamisms, should build on a compositional CoTs and not a specific RCoT. It is out side the scope of this work to discuss these any further; detailed discussion of which can be found in [Abb12].

We now defined two operations over RCoTs: i) $extend(RCoT_1, < elements >)$: concatenates $< elements >$ to $sq.RCoT_1$; and ii) $combine(RCoT_1, RCoT_2)$: returns a set of elements, each represent the input CoT. In such an operation we should properly check for cycles in the trust relation — but our context cannot allow this, and so we do not concern ourselves with a detailed discussion of that aspect of the model. The *combine* operation is of course idempotent, commutative, and associative, so we will allow ourselves to use terms such as $combine(X, Y, Z)$ since these are unambiguous.

6.3.3 Physical Layer DCoT and CDCoT

We now define how trust is composed from the members in a particular grouping in Clouds. Understanding compositional chains of trust is a vital requirement for establishing trust in Clouds. This is because Cloud resources at upper layers are served by collaborating set of resources rather than a specific resource. We identify two types of domain configurations: *homogeneous* and *heterogeneous*. In a homogeneous setting all resources are configured uniformly resulting in identical CoTs. Example of this is the resources within a physical domain or a virtual domain. Each resources member of a physical domain are identical and carefully selected, interconnected and positioned to achieve the domain properties. Similarly, resources of a virtual domain are identical as they represent (as a result of horizontal scalability) a replication of VMs hosting an instance of an application resource. Application domains, on the other hand, are heterogeneous as they are composed of resources having different CoTs. Collaborating domains follow the same concept as domains. For example, collaborating domain of the physical layer are homogeneous as they should serve as backup of each other. Virtual and application layer collaborating domains, on the other hand, are heterogeneous as they serve to identify the interdependencies between domains rather than as backup of each other.

We identify two types of compositional CoT, namely: the `domains chain of trust (DCoT)` and `collaborating domains chain of trust (CDCoT)`. These CoTs are composed of two entities: a *root_of_trust* and the combination of all CoT of the entities member of the domain/collaborating domains. Unlike a RCoT, the *root_of_trust* of DCoT/CDCoT attests to the trustworthiness of the way the domain or collaborating domain is managed and operated. We need a *root_of_trust* that satisfies two main properties: i) its trustworthiness can be measured and assessed at all times, and ii) can provide strong assurance about the trustworthiness of the way the domain or collaborating domain are managed and operated.

The DCoT at the physical layer, $DCoT(D_{Physical})$, is composed of two entities: i) the first is the combination of all RCoTs member of the physical domain. The physical domain is homogeneous, and as a result all RCoTs member of the domain are identical. Combining identical chains of trust is equal to either chain of trust (as discussed in section 6.3.2). ii) The second element is the *root_of_trust* of the domain. As explained in section 6.2.2, each physical resource member of a domain would run a trustworthy copy of DC-C, and VCC would run a trustworthy copy of DC-S. DC-S measures the trustworthiness of the DC-C running at each resource, and also provides the assurance that DC-C can only operate in a domain's device if the device is running in a specific state. A verifier can independently acquire VCC's CoT and assess its trustworthiness as outlines in 6.2.2 and explained in [AAM11]. These satisfy our two stated properties of the *root_of_trust* of DCoT and CDCoT. Therefore, we propose the VCC's chain of trust ($RCoT(VCC)$) to act as the *root_of_trust* of physical DCoT.

Assume a homogeneous physical domain, $D_{Physical}$, consists of resources R_0, R_1, .., R_n such that $\forall\, i,j : [0..n] \bullet RCoT(R_i) == RCoT(R_j)$. The $DCoT(D_{Physical})$ is then defined as follows.

$$DCoT(D_{Physical}) = combine(RCoT(VCC), RCoT(R_0))$$

DC-S, which is part of RCoT(VCC), vouches and attests to the trustworthiness of the members of $D_{Physical}$, i.e. DC-S attests to the trustworthiness of each RCoT including DC-C. DC-S also provides the assurance that DC-C can only operate and be member of a domain when its serving host has a specific RCoT. Therefore, a verifier only needs to attest to the trustworthiness of RCoT(VCC) and DC-C, i.e. an extended CoT starts from RCoT(VCC) and extends to DC-C (DC-S measures and attests to DC-C — see [AAM11]). All physical domain resources have identical values of DC-C when it runs as expected. As a result we can redefine the DCoT($D_{Physical}$) as follows

$$DCoT(D_{Physical}) = extend(RCoT(VCC), DC-C)$$

After discussing DCoT we move to CDCoT at the physical layer. DC-S and DC-C manages both physical domains and physical collaborating domains. As a result, an appropriate *root_of_trust* of CDCoT is the same as the *root_of_trust* of DCoT. The *root_of_trust* of CDCoT is already included in DCoT, thus we can exclude it from the physical CDCoT. DCoTs member of a physical collaborating domains would typically have identical DCoT (this is a result of Cloud properties ,e.g. resilience and availability, which are achieved by having identical physical domains to join a collaborating domain(s). Suppose a collaborating domain $CD_{Physical}$ is composed of domains: $D_{P0}, D_{P1}, .., D_{Px}$ such that $\forall\, i,j : [0..x] \bullet DCoT(D_{Pi}) == DCoT(D_{Pj})$). The $CD_{Physical}$ is then defined as follows.

$$CDCoT(CD_{Physical}) = DCoT(D_{P0})$$

By substituting the value of $DCoT(D_{P0})$ from $DCoT(D_{Physical})$, we then have the following

$$CDCoT(CD_{Physical}) = extend(RCoT(VCC), DC-C)$$

The above shows that physical CDCoT is mainly based on VCC and DC-C. VCC trustworthiness can be measured by a verifier, and DC-C trustworthiness can be verified by VCC. This is the foundation of the *root_of_trust* of physical layer which acts as a foundation for the layer above it (i.e. virtual layer), as discussed in [Abb12].

6.4 High Level Architecture

Having defined the Cloud taxonomy, relationship between Cloud components, and the compositional chains of trust which help in assessing Cloud trustworthiness we can now cover the scheme framework; subsequent section provides a prototype of the discussed framework. We use OpenStack Compute ([Ope11]) as a management framework to represent the VCC. OpenStack is an open source tool for managing Cloud infrastructure which is still under continuous development.

Figure 6.3 presents a high level architecture which illustrates the main entities and the general layout of our scheme framework. We use OpenStack controller node (i.e. VCC) and OpenStack nova-compute (i.e. a computing node at the physical layer). The computing node runs a hypervisor which manages a set of VMs. VCC receives two main inputs: user requirements and infrastructure properties. VCC manages user virtual resources based on such input. In this section we focus on our introduced components and, in addition, we cover the changes we

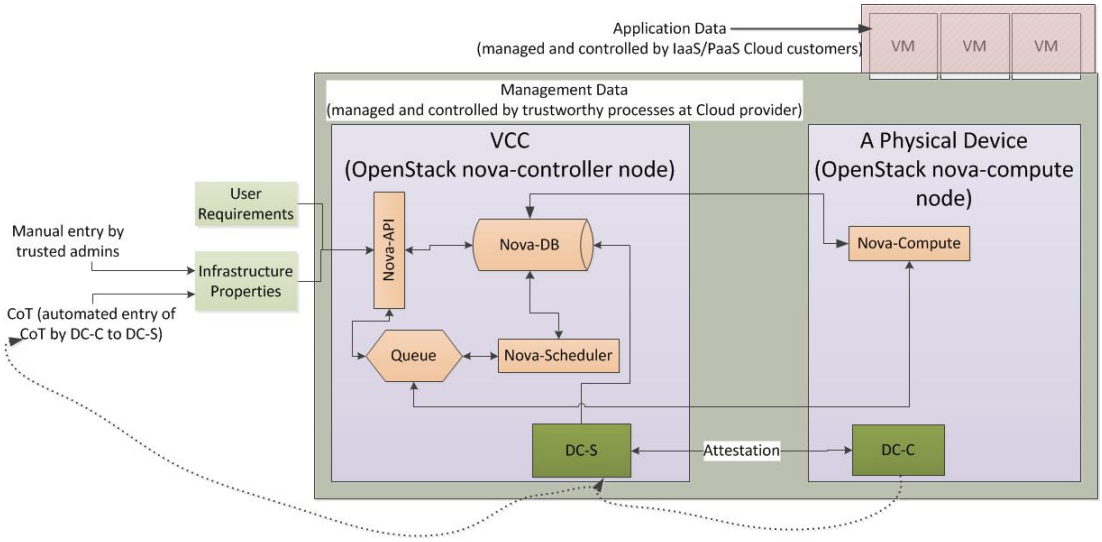


Figure 6.3: High level architecture

introduced at the following components of OpenStack (further details about other components can be found at [Ope11]): *nova-api*, *nova-database*, *nova-scheduler*, and *nova-compute*. We update some of the functions of those components and introduce new functions.

6.4.1 Nova-api

Nova-api is a set of command lines and graphical interfaces which are used by Cloud customers when managing their resources at the Cloud, and are also used by Cloud administrators when managing the Cloud virtual infrastructure. We updated *nova-api* library to consider the following: i) **Infrastructure Properties** — Clouds' physical infrastructure are very well organized and managed, and its organization and management associates its components with infrastructure properties. Examples of such properties include: RCoT, components reliability and connectivity, components distribution across Cloud infrastructure, redundancy types, servers clustering and grouping, and network speed. ii) **User Requirements** — These include technical requirements, service level agreement, and user-centric security and privacy requirements. And iii) **Changes** — These represent changes in: user properties (e.g. security/privacy settings), infrastructure properties (e.g. components reliability, components distribution across the infrastructure and redundancy type), and infrastructure policy.

The main changes which we introduced at *Nova-api* includes the following: i) add an option to enable users to manage their requirements which include but not limited to security and privacy aspects (adding, updating, listing and removing user requirements), ii) add an option which enable administrators to manage Clouds infrastructure properties and policies, e.g. associate computing nodes to their domains and collaborating domains, and iii) provide an interface which enable automated collection of the properties of the physical resources through trustworthy channels — at this stage we specifically focuss on collecting resources' chains of trust, RCoT. These data are stored on *Nova-database* and are used by our proposed scheduler.

6.4.2 Nova-database

Nova-database is composed of many tables holding the details of the Cloud components. It also holds users, projects and security details. We extended *nova-database* in different directions to

realize the taxonomy of Clouds, user security requirements, and infrastructure properties which include the compositional chains of trust. Figure 6.4 illustrates our proposed modification of *nova-database* in bold format which are as follows.

Compute_nodes is an existing nova-database table that holds records reflecting a computing resource at the physical layer. We updated this table by adding the following additional fields: i) physical resource chain of trust, RCoT(Physical), ii) security properties that holds a list of security details of the computing resource, and iii) a foreign key establishing the relation between the physical resource with its physical domain as exists in Physical_Layer_Domain table.

Physical_Layer_Domain. We added this table to hold the records of the Cloud physical domains, to define the relationship amongst resources, and to hold physical domains' metadata. The domain metadata includes the domain capabilities, DCoT, and a foreign key pointing to the table which identifies the relative location of the physical domain within Clouds infrastructure.

Location and Location_Distances. The aim of those two tables is to identify all possible *locations* at the Cloud infrastructure. It also defines relative distance between pairs of all the identified locations. How the two tables are bound is as follows: the compute_node table is bound to the physical_layer_domain table, and the physical_layer_domain table is bound to a specific location identifier in the *Location* table. The latter is bound with location_distances table which specifies all distances between a location identifier and all other location identifiers. In this we assume the resources of a physical domain are within close physical proximity which reflects current deployment scenarios in practical life.

Collaborating_PL_Domain. We added this table which establishes the concept of collaborating physical domains. Each record in the Collaborating_PL_Domain table identifies a specific collaborating domain (i.e. a backup domain) for each physical source domain with a priority value. A source domain can have many backup domains. The value of the priority identifies the order by which physical backup domains could possibly be allocated to serve a source domain needs. Backup domains are used in maintenance windows, emergencies, load balancing, etc. For example, a virtual resource of a failed source domain can be hosted at an available backup domain that is associated with highest priority flag. Backup domains of a specific source domain should have the same capabilities and DCoT as the source physical domain itself.

Instances is an existing OpenStack table representing the running instances at computing nodes. We updated the table by adding the following fields: i) virtual resource chain of trust RCoT(Virtual), ii) application resource chain of trust RCoT(Application) and iii) two foreign keys which establish a relationship with the instance's virtual and application domain tables, as defined in the *Virtual_Layer_Domain* and *Application_Layer_Domain* tables, respectively. It is outside the scope of this work to cover the details of those tables as we mainly focusing on the scheduler requirements.

Services table is an existing OpenStack table which binds the virtual layer resources to their hosting resources at physical layer. We did not modify this table.

Other tables. Openstack has many more tables and we also added more tables which is outside the scope of this work to discuss.

Most of the records in the *nova-database* are uploaded automatically using: i) the proposed software agents, ii) the modified *nova-api*, and/or iii) via OpenStack management tools. Ideally such records should be securely protected, collected and managed via trustworthy processes. At this stage our focus is on providing high level architecture design, providing a running Cloud scheduler, and providing software agents that can attests to the trustworthiness of Openstack components and then push the result to to nova-database. Full automation of Cloud management services is our planned long term objective.

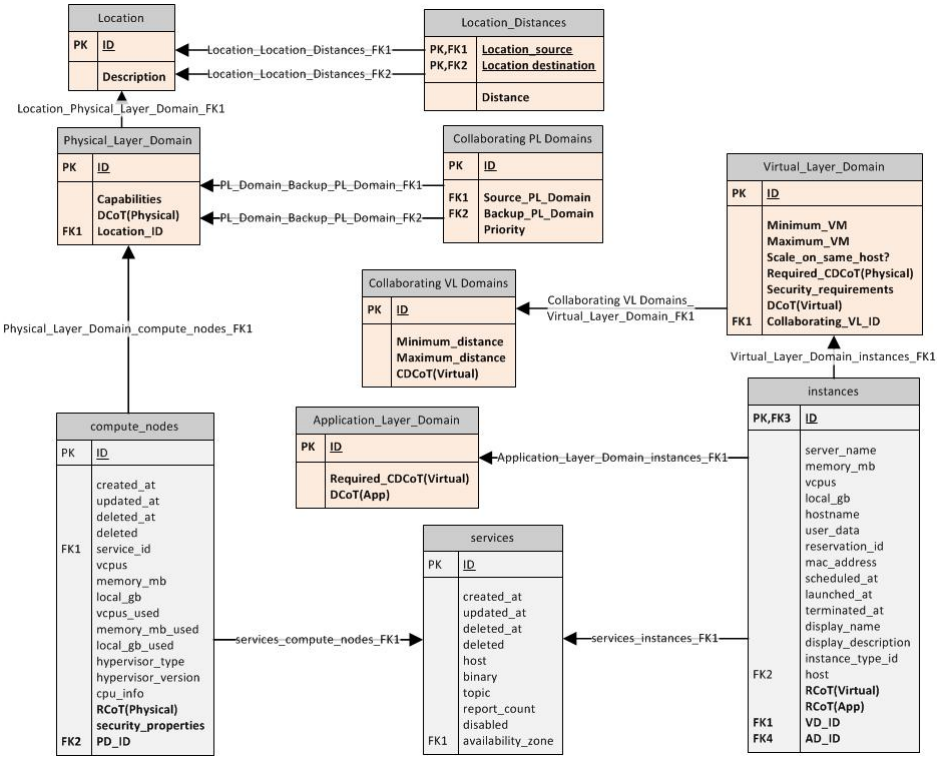


Figure 6.4: Updates on nova-database to include part of the identified Clouds relations

6.4.3 Nova-scheduler

In OpenStack, *Nova-scheduler* controls the hosting of VMs at physical resources considering user requirements and infrastructure properties. Current implementations of *nova-scheduler* do not consider the entire Cloud infrastructure neither they consider the overall user and infrastructure properties. According to OpenStack documentation, nova-scheduler is still immature and great efforts are still required to improve *nova-scheduler*. We implemented a new scheduler algorithm (referred to as ACaaS (Access Control as a Service)) that performs the following main functions: i) considers the discussed Cloud taxonomy when allocating a virtual resource to be hosted at a physical resource, for example, it enforces domains and collaborating domains policies when managing the life-cycle of virtual resources, ii) selects a physical domain's resource which has physical infrastructure properties that can best match user properties, and iii) ensures that the user requirements are continually maintained.

ACaaS collaborates with the following software agents (see Figure 6.3).

- *Cloud client agent, DC-C* runs at computing nodes and it does the following main functions: calculates the computing node chain of trust, RCoT, and passes the result over to DC-S, continually assess the status of the computing node, manage the domain and collaborating domains members based on policies distributed by DC-S (e.g. a VM can only operate with a known value of a chain of trust and when the hosting physical collaborating domains have a specific value of CDCoT(Physical) as defined by user properties).

- *Cloud server agent, DC-S* runs at OpenStack node controller (i.e. VCC) and it does the following main functions: maintains and manages OpenStack components (including the nova-scheduler) by ensuring they operate the Cloud only when they are trusted to behave as expected, manages the membership of the physical and virtual domains, and attests

to DC-C trustworthiness when its computing node joins a physical domain. DC-S also intermediates the communication between DC-C and nova-scheduler, attests to DC-C's computing-node trustworthiness, collects the computing node RCoT, and then calculates DCoT, CDCoT, and stores the the result in an appropriate field in nova-database.

Next section discusses our prototype and how these components collaborate to establish trust in Clouds.

6.5 Prototype

Having defined a high level architecture of our scheme, this section describes our prototype. As we discussed earlier, at this stage we focus on a trustworthy collection of resources' RCoT, calculate DCoT and CDCoT, and then using the ACaaS scheduler to match user properties with the infrastructure properties. Other infrastructure properties, at this stage, are either collected automatically (such as physical resources capabilities) or entered manually (such as physical location of joining computing nodes). However, such values could be altered by system administrators. Planned future work will focus on extending our framework to establish trustworthy collection/calculation of the other properties. The trust measurements performed by DC-C identifies the building up of a resource's RCoT and its integrity measurements. In this section we discuss the functionalities we implement at OpenStack controller node (i.e. the VCC) and computing node, which covers trust establishment building on both remote attestations and secure scheduling.

6.5.1 Trust Attestation via DC-C

Our implementation is based on open source trusted computing infrastructure. The open source is built on a Linux operating system and it helps in building an RCoT on a computing node. Such RCoT, as discussed earlier, starts from the computing node TPM and move up to the DC-C, as illustrated in Figure 6.5. In this section we cover the details of this. The workflow starts by the platform bootstrapping procedure in which the trusted BIOS initializes the TPM. Once this is done, the trusted BIOS measures and then loads the trusted bootloader [Trud]. The trusted bootloader is the Trusted Grub in our case, which measures and loads the Linux kernel.

The Linux kernel is updated ensuring that the IBM Integrity Measurement Architecture (IMA) [SZJvD04] is enabled by default. The IMA measures all critical components before loading them, which includes kernel modules, user applications, and associated configuration files. The values of such measurements are irreversibly stored inside the Platform Configuration Registers (PCRs) which are protected by the TPM. By default the IMA implementations use PCR #10 to store such measurements.

The TPM driver and the Trusted Core Service Daemon (TCSD) [Trub] expose the Trusted Computing Services (TCS) to applications. These components constitute the part of DC-C for collecting and reporting the trust measurement of a resource. An RCoT is, hence, constructed from the CRTM [Truc], which itself resides and protected by the Trusted BIOS.

Table 6.1 lists the IMA measurement log which illustrates part of the records of the bootstrapping process for our prototype. The IMA measurement log is the source for generating Integrity Reports (IR). IR is used, as we discuss latter, to determine the genuine properties of a target system during the remote attestation process. We now discuss the IMA measurement log in further details. The first column, in Table 6.1, shows the value of $PCR10$ after loading the components of the third column. The second column records the hash value of the

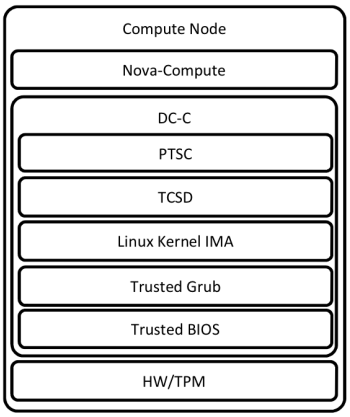


Figure 6.5: Compute Node Architecture

Table 6.1: Compute node bootstrapping measurement log

PCR10	HASH	Loaded Component
$pVal_0$	$hVal_0$	boot_aggregate
$pVal_1$	$hVal_1$	/init
$pVal_2$	$hVal_2$	ld-linux-x86-64.so.2
$pVal_3$	$hVal_3$	libc.so.6
...	...	...
$pVal_i$	$hVal_i$	nova-compute.conf
...	...	...
$pVal_j$	$hVal_j$	python
...	...	...
$pVal_k$	$hVal_k$	nova-compute
$pVal_{k+1}$	$hVal_{k+1}$	libssl.so.1.0.0
...	...	...
$pVal_l$	$hVal_l$	nova.conf
...	...	...

loaded component. The first record, in Table 6.1, holds the value of the *boot_aggregate*. The *boot_aggregate* is the combined hash value of $PCR0 - PCR7$; i.e. it possesses the measurement of the Trusted Computing Base (TCB) of a computing node, including the Trusted BIOS, Trusted Bootloader and the image of the Linux Kernel together with its initial ram-disk and kernel arguments. Whenever a software component is loaded, the IMA module of the kernel generates a hash value of the binary code of the loaded component. The hash value (e.g. $hVal_i$) is then *extended* into $PCR10$ by invoking the *TPM_Extend* command [Tru07c]. Such command updates $PCR10$ to reflect the loaded component as follows: $pVal_i = hash(pVal_{i-1}, hVal_i)$.

Subsequent rows in Table 6.1 present the measurement logs for the bootstrapping workflow at the adopted operating system, Ubuntu 11.04. Other OpenStack components, as illustrated in the table, are then measured which includes *nova-compute.conf* init script, *python* program, *nova-compute*, supporting libraries, and critical configuration files.

To reduce the complexity of the system and to focus on practical Cloud deployment cases, our prototype turns off all unnecessary services at the base system. As a result, the value of $PCR10$ do not get changed by default except if a new software module (e.g. user program,

kernel modules, or shared libraries) is forced to be loaded on the computing node (could be either a good one (e.g. security patching) or a malicious one (e.g. loading attacking codes)). In such a case, the new software module will be measured and added to the log records. This would change the value of *PCR*10, reflecting the changes in system state.

Our prototype intentionally filters out the IMA measurements of VMs, i.e. the QEMU program in our prototype. This is because, as explained in this work, VMs chain of trust should be built on compositional chains of trust and not a single resource chain of trust, i.e. they should not be considered part of the TCB of a computing node. Most importantly, measurements of VMs should be controlled by IaaS Cloud users and not Cloud providers as such measurements will likely to raise user's privacy concerns and, in addition, such measurements would significantly increase the complexity of trust management. If an exploited VM runs on a computing node, for example, to perform malicious behaviour to other components and application, e.g. by side-channel attacks or exploiting hypervisor vulnerabilities, the properties of the computing node would change once the exploited VM starts to affect the TCB components, as we discussed earlier. In such a case, DC-C will stop to operate, i.e. it evicts itself from the physical domain and VMs will be forced to migrate to heathy computing node.

Finally, *DC-C* collects the integrity measurement logs as recorded by the IMA and generates an IR following the specifications of the Platform Trust Service interface (PTS) [Tru06]. The DC-C, as we discuss in the next section, sends the IR report and the signed PCR values to DC-S on request. In our prototype, this component is implemented by integrating the *PTSC* module from the OpenPTS [Tru11].

6.5.2 Trust management by DC-S

This section starts by summarizing high level steps of the implemented part of the system workflow and then moves into the prototyping details related to DC-S, as follows (our previous work [Abb11a] provides detailed discussion of some of these steps supported by cryptographic protocols, our prototype discussed in subsequent subsections explains how we implemented these steps).

1. Cloud security administrators could either create a new physical domain or use an existing domain. The creation process involves deciding on domain capabilities, location, and defining its collaborating domains. As discussed in Section 6.4, we updated *nova-api* to enable administrators to manage this process.

2. Cloud security administrators then install DC-C and nova-compute at all new physical computing nodes that are planned to join the domain.

3. DC-C joins the Cloud physical domain by communicating with DC-S. DC-S attests to DC-C trustworthiness and establishes offline chain of trust with DC-C (using sealing and remote attestation concepts as proposed by TCG specifications). DC-C calculates RCoT(Physical), as described in our prototype. DC-C then passes the results to DC-S.

4. DC-S stores the RCoT(Physical) at the *compute_nodes* table. DC-S ensures that all devices in each domain have the same capabilities. The sealing mechanism, which is established in previous steps, assures DC-S that DC-C can only operate with the same value of the reported RCoT(Physical). If this value changes (e.g. hacked) DC-C will not operate. This prevents VMs from starting at hacked device and rather get migrated to other computing nodes member of the same physical domain.

5. Users, using *nova-api* command, deploy their VMs and associate them with certain properties. Such properties include, for example, the required CDCoT(Physical), and the multi-tenancy restrictions which control the sharing of computing nodes with other users.

6. The ACaaS scheduler allocates an appropriate physical domain to host the user VM. The properties of the physical domain and its member devices should satisfy users requirements.

The remaining part of this section covers the implementation of the remote attestation process and the secure scheduling.

6.5.2.1 Remote attestation

Our prototype implements remote attestations using the OpenPTS [Tru11]. OpenPTS is controlled by DC-S which runs on VCC. OpenPTS sends an attestation request to each computing node to retrieve their IR and the PCR values. When a computing node sends their results, OpenPTS examines the consistency of the IR and the PCR values [Truc], and then examines the security properties by matching the reported IR with the expected measurement from a white-list database [Truc]. White-list represent sets of measurements which are stored in a database, each set of measurements is calculated based on a carefully selected "good" platform configuration state. The calculation is performed in a form of hash values for each pre-loaded software components on the "good" platform. For the purpose of our prototype, we used two newly installed Ubuntu 11.04 servers that each has a default configurations. Such configurations are composed of the minimal required environment for a computing node to perform its planned functions. The hash values of the software stack of any computing node at the Cloud infrastructure should exist within the white-list database. If not, we consider the computing node to be untrusted. The "good" configurations (or any set of white-lists) can be extended or changed by adding or updating the entries of the corresponding database.

We now discuss the attestation protocol in more details. Every computing node (C_i) is identified by its AIK (Attestation Identity Key), and the Cloud controller VCC (M) serves as the Privacy-CA [Truc, Trua] for certifying and managing all these AIKs. When a new computing node is added to the Cloud infrastructure, it must first register at the VCC and certify a specific AIK. Only registered computing nodes can connect to the VCC as their certified AIKs cannot be forged and, importantly, AIKs can only be used inside the genuine TPM which generates them. The protocol 8 outlines the registration steps of C_i at M.

Whenever a computing node sends a request to connect to the VCC, e.g. after the registration or a after reboot, a trust establishment protocol is established, which is listed in Protocol 9.

The configurations of computing nodes could possibly be altered after an attestation session, e.g. loading a new application. In such cases, the computing node attestation properties (as maintained by the VCC) would be violated. Addressing this require establishing a trusted channel [GSS+07] to *seal* [Truc] the communication key with the verified PCR values. The sealing process provides the assurance that the communication key is protected by the TPM and can only be loaded when the values of the platform PCRs have the expected attested values. Whenever a computing node's configurations change, the DC-S will not be able to load the key and, hence, any future communication between the computing node and the VCC will be discarded. This in turn would trigger a new attestation from the VCC to the computing node.

When the sealed keys are loaded into memory, the implementation of the trusted channel requires a very small TCB enforcing strict access to the memory area storing the keys. Having large TCB, could results in the tampered system to be capable of retrieving the key from the

Protocol 8 Computing node registration protocol.

C_i sends a registration request to M as follows. First, C_i sends a request to its TPM to create an AIK key pair using the command TPM_CreateAIK. The TPM then generates an AIK key pair. The generated private part of the key pair never leaves the TPM, and the corresponding public part of the key pair is signed by the TPM Endorsement Key (EK)[Truc]. The EK is protected by the TPM, and never leave it. C_i then sends a registration request to M. The request is associated with EK certificates and the AIK public key, and other parameters.

$$C_i \rightarrow M : Cert(K_{EK_i}), \{K_{AIK_i}\}_{K_{EK_i}^{-1}} \tag{8.1}$$

M certifies AIK_i as follows. M verifies $Cert(EK_i)$. If the verification succeeds, M generates a specific-AIK certificate for C_i, and it also generates a unique ID, CID_i, to and sends them over to C_i.

$$M \rightarrow C_i : \{Cert(K_{AIK_i}), CID_i\}_{K_M^{-1}}, Cert(K_M) \tag{8.2}$$

Protocol 9 Trust establishment protocol.

M sends an attestation request to C_i. M first sends a nonce N_a to C_i.

C_i then reports an attestation ticket to M as follows. C_i sends its PCR values, and the measurement log IR back to M, together with N_a. These are signed by C_i's AIK.

$$C_i \rightarrow M : \{N_a, \{PCR\}, IR\}_{K_{AIK_i}^{-1}} \tag{9.1}$$

M then verifies the message sent by C_i as follows. M verifies the AIK_i signature, and N_a value matches the sent nonce. If succeed it examines the consistency of PCR and IR, and then determine the properties of C_i based on the value of IR.

memory and pretend to be on a trusted state. However, small TCB is not a trivial task to implement and lots of efforts are required in this direction, especially, considering the complexity and scalability of the hosting Cloud system (we leave this important subject as a planned future research). As an attempt to lessen the effect of this threat in our prototype we require a periodic attestation which keeps the security properties of a computing node up to date. We implemented this by associating a timer with each computing node. Re-attestation is enforced whenever the timer expires. Discovered un-trusted computing nodes will be immediately removed from the database and they would need to re-enroll into the system. In addition, VMs running on untrusted computing nodes will be forced to migrate to other computing nodes that are member of the same physical domain.

6.5.2.2 Secure scheduling

As we discussed in previous sections, computing nodes are organized into physical domains. Such organization would be based on the properties of each computing node (i.e. security, privacy and other properties) which enable it serve the needs of the domain. Users can specify the expected properties that could host their VMs. Some of these properties could be represented by a set of PCR values. However, PCR values are hard to pre-calculate and manage considering that they represent aggregated hash values of software components when loaded in a specific order. In our proposed prototype users do not really need to specify PCR values, rather they could select their expected hosting environment based a provided sets of white-lists (as discussed earlier). Whilst enrolling a computing node into a domain, the administrators compose and specify the white-list of the computing node, in accordance with its properties. The computing node is then periodically attested based on this assigned white-list. Whenever a computing node's white-list change, it will be immediately remove from the domain. However, a genuine updates on the properties of a computing node, e.g. applying security patch, require administrators to adjust the corresponding entries in the white-list database. This is much simpler in comparison with the need to manage a huge set of possible "good" PCR values. In our prototype, users would identify part of their required properties in a form of a white-list, ACaaS scheduler would then deploy the user VMs on nodes having same properties as requested by the user. ACaaS with collaboration with DC-S and DC-C periodically examine the consistency of such properties.

6.5.3 Preliminary Performance Evaluation

For preliminary evaluations, we measure the overheads introduced by TCG trusted computing infrastructure. Our assessments focus on the critical operations which include: *PCR Quote* instruction, PCR verification instruction, and a full remote attestation procedure. In addition, we assess the additional time which is required when extending RCoT on a Compute node during the bootstrapping process, as discussed earlier. Table 6.2 shows the performance metrics for these operations. Each value in the table represents averaging 15 execution times of every corresponding operation. We also included the standard deviations.

We found that a full attestation requires around 10 seconds. This includes, on the computing node, quoting the PCR values and generating the IRs, and on the VCC, verifying the PCR values and analysing the IRs. As we discussed earlier, our prototype performs off-line attestation. As a result, the attestation values are stored in the database and queried only when necessary. Hence, the impacts on the execution flows of the Clouds management facilities are negligible. For example, the Cloud scheduler when making a scheduling decision could directly

Table 6.2: Trusted computing operations overheads

	Avg Time (sec.)	StDv (sec.)
Full Attestation	10	0.8
PCR Quote	0.759	0.007
Verify Quote	0.0021	0.0006
Trusted Boot	213	5
Regular Boot	56	2

fetch a previously stored attestation results from the database instead of performing an on-line attestation that requires, based on our measurements, around 10 seconds.

An important point to consider is the trade-off between security and performance. Specifically, in our prototype a full attestation required 10 seconds; the longer the gap we leave between attestations the better the performance we got, and the less assurance on changes of device status. Analogously, the more frequent an full attestation is performed the worse performance we got, and the more assurance on changes of device status. In other words, the delays between successive attestations determine the required time for the violation detection. Even if the delay between successive attestations is zero, there is still a 10 seconds time interval where a violation could happen. Within this period, the state of a target computing node could have been changed without reflecting such a change to its security properties as stored in the database.

When we carefully check the constitution of a full remote attestation procedure we find that the consumed time when quoting and verifying the values of PCR is much less than the consumed time for generating and verifying the IRs. However, handling the IRs is only necessary when the state of a computing node get changed, i.e. the security properties of the computing node can safely be assumed identical as long as the subsequent measurements of PCR values are identical, and as a result, having identical PCR values eliminate the need to do further examining the IRs. Hence, the attestations to a computing node can be optimised by first compare the PCR values with the previous ones. Only when these values do not match the computing node generates IR and the VCC verifies it. In practice, platforms' state rarely get changed [LM09]; for example, a computing node configurations only changes when it loads new privileged executable, e.g. loading new security modules, applying patches or launching malicious attacks. Most of the time, the attestation delay can be reduced by only generating and verifying PCR values, which, as shown in 6.2, is less than 0.8 seconds.

Finally, an inevitable bootstrapping delay will be incurred for extending the RCoT, as all software components prior loading should first get measured and extended to the TPM. As illustrated in Table 6.2, this process is 3 times longer than booting a general system. This is not a major issue in Clouds, as in practice computing nodes do not get rebooted frequently. Hence, the bootstrapping delay is negligible.

6.6 Related Work

The issue of establishing trust in the Cloud has been discussed by many authors (e.g. [Aba09, HRM10, HNB11, KM10b, SMV+10]). Much of the discussion has been centered around reasons to "trust the Cloud" or not to. Khan and Malluhi [KM10b] discusses factors that affect consumer's trust in the Cloud and some of the emerging technologies that could be used to es-

tablish trust in the Cloud including enabling more jurisdiction over the consumers' data through provision of remote access control, transparency in the security capabilities of the providers, independent certification of Cloud services for security properties and capabilities and the use of private enclaves. The issue with jurisdiction is echoed by Hay et al [HNB11], who further suggest some technical mechanisms including encrypted communication channels and computation on encrypted data as ways of addressing some of the trust challenges. Schiffman et al [SMV+10] propose the use of hardware-based attestation mechanisms to improve transparency into the enforcement of critical security properties. The work in [Aba09, HRM10] focus on identifying the properties for establishing trust in the Cloud.

Few papers propose the usage of TPM in Clouds for remote attestation (e.g. [RM11, SGR09, SMV+10]). The work in [RM11] proposes a remote attestation mechanism based upon reputation systems and TCG remote attestation concept. This work requires resources, when interacting with other resources, to attest to their trustworthiness, keep a copy of the measured trust values, and to share it with other resources. The trust measurements are associated with timestamp and would need to be revalidated after it expires. The dissemination of such measurements between resources form a web-of trust. We identify the following weaknesses in the scheme: i) the relation between entities in Clouds are identified mainly based on the dynamic behaviour of the entities (i.e. a relationship between entity A and B is established when A exchanges messages with B), and ii) the entities of the web-of-trust (consists of a large and replicated database of trust measurements) that expire and require frequent re-evaluation. Establishing trust between entities based on entities' dynamic behaviour (i.e. point i) is not accurate and might affect Cloud availability and resilience. For example, entities at the physical layer forming a collaborating domain do not communicate frequently, and sometimes never communicate directly. If a physical domain fails then all its hosted resources must start-up immediately at another physical domain. Establishing a chain of trust at this critical stage would affect the timing of service recovery. In addition, point (ii) is time consuming and, by considering the enormous number of resources in Clouds, it is not practical to keep revalidating the trust measurements.

The work on ([SGR09, SMV+10]) provide remote attestation for either the entire Cloud infrastructure or for the the physical resources hosting a specific VMs. However, we argue that it is not practical to attest to the entire Cloud infrastructure considering its huge and distributed resources, neither it is practical to attest to a specific set of physical resources considering the dynamic nature of Clouds where VMs can move between different physical resources. In addition, these papers require users to understand to some extent the Cloud infrastructure, i.e. they do not provide transparent Cloud's infrastructure.

Cloud scheduler is proposed and implemented by industry body such VMWare in their VCenter product [VMw10] and open source tools such as Openstack and OpenNebula [Ope10b, Ope10a]. A scheduler decisions could be based on various factors at the physical layer such as memory, CPU, and load. Currently the main implemented scheduling algorithms are: i) chance: in this method, a computing node is chosen randomly, ii) availability zone: similar to chance, but the computing node is chosen randomly from within a specified availability zone, and iii) simple: in this method, a computing node whose load is the least is chosen to run an instance. Such algorithms are still basic; i.e. they do not consider the entire Cloud infrastructure neither they consider wide users' requirements.

Our scheme is different from the above as it is based on practical understanding of how Clouds work. We explicitly identified the Cloud infrastructure and user properties. We also considers Cloud taxonomy, dynamic nature and the practical relationships between Cloud entities. Understanding these help us in providing a novel Cloud scheduler that matches user properties with infrastructure properties ensuring user requirements are continuously met fol-

lowing pre-agreed SLA. In addition, we developed software agents running on computing nodes to enforce the scheduler decision and also to provide trustworthy report about the computing node trust level. Moreover, we assess the trustworthiness of the infrastructure using the compositional chains of trust scheme, which considers both Cloud dynamic nature and way Cloud infrastructure is managed.

6.7 Conclusion

Cloud infrastructure is expected to be able to support Internet scale critical applications (e.g. hospital systems and smart grid systems). Critical infrastructure services and organizations alike will not outsource their critical applications to a public Cloud without strong assurances that their requirements will be enforced. This is a challenging problem to address, which we have been working on as part of TClouds project. A key point for addressing such a problem is providing a trustworthy Cloud scheduler supported by trustworthy data enabling the scheduler to take the right decision. Such trustworthy source of data is related to both user requirements and infrastructure properties. User requirements and infrastructure properties are enormous, and assuring their trustworthiness is our long term objective. This work covers one of the most important properties which is about measuring the trust status of the Cloud infrastructure, and enabling users to define their minimal acceptable level of trust. We presented our prototype which does not only cover the proposed scheduler but it also covers our related previous work focussing on Clouds trust measurements. The key advantage of our prototype is that it considers critical factors that have not been considered in commercial schedulers, such as: considering the overall Clouds infrastructure and computing nodes' trustworthiness. In addition, our prototype enables users to identify their expected trust level at physical resources hosting their virtual resources without the need for users to get involved into infrastructure complexity. We also prototyped client agents which enforces the scheduler decisions across the wide scale Cloud infrastructure.

This work presents a core component of our long term objective of establishing Clouds' trustworthy self-managed services. We identified different areas of research to expand this work, which we are planning to work on in the near future.

Bibliography

[AA08a] Muntaha Alawneh and Imad M. Abbadi. Preventing information leakage between collaborating organisations. In *ICEC '08: Proceedings of the tenth international conference on Electronic commerce*, pages 185–194. ACM Press, NY, August 2008.

[AA08b] Muntaha Alawneh and Imad M. Abbadi. Sharing but protecting content against internal leakage for organisations. In *DAS 2008*, volume 5094 of *LNCS*, pages 238–253. Springer-Verlag, Berlin, 2008.

[AAM11] Imad M. Abbadi, Muntaha Alawneh, and Andrew Martin. Secure virtual layer management in clouds. In *The 10th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-10)*, pages 99–110. IEEE, Nov 2011.

[Aba09] J. Abawajy. Determining service trustworthiness in intercloud computing environments. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009 10th International Symposium on*, pages 784 –788, dec. 2009.

[ABB+05] Alessandro Armando, David A. Basin, Yohan Boichut, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Paul Hankes Drielsma, Pierre-Cyrille Héam, Olga Kouchnarenko, Jacopo Mantovani, Sebastian Mödersheim, David von Oheimb, Michaël Rusinowitch, Judson Santiago, Mathieu Turuani, Luca Viganò, and Laurent Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In *CAV*, pages 281–285, 2005.

[Abb11a] Imad M. Abbadi. Clouds infrastructure taxonomy, properties, and management services. In Ajith Abraham, Jaime Lloret Mauri, John F. Buford, Junichi Suzuki, and Sabu M. Thampi, editors, *Advances in Computing and Communications*, volume 193 of *Communications in Computer and Information Science*, pages 406–420. Springer Berlin Heidelberg, 2011.

[Abb11b] Imad M. Abbadi. Middleware Services at Cloud Virtual Layer. In *DSOC 2011: Proceedings of the 2nd International Workshop on Dependable Service-Oriented and Cloud computing*. IEEE Computer Society, August 2011.

[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.

[Abb12] Imad M. Abbadi. Clouds trust anchors. In *The 11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-11) (to appear)*. IEEE, Jun 2012.

[AC04] Alessandro Armando and Luca Compagna. SATMC: A SAT-Based Model Checker for Security Protocols. In *Logics in Artificial Intelligence*, 2004.

[AC08] Alessandro Armando and Luca Compagna. Sat-based model-checking for security protocols analysis. *International Journal of Information Security*, 7:3–32, 2008.

[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.

[AL11] Imad M. Abbadi and John Lyle. Challenges for provenance in cloud computing. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP '11)*. USENIX Association, 2011.

[AN11] Imad M. Abbadi and Cornelius Namiluko. Dynamics of trust in clouds — challenges and research agenda. In *The 6th International Conference for Internet Technology and Secured Transactions (ICITST-2011)*, pages 110–115. IEEE, December 2011.

[AS86] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. Technical report, Cornell University, Ithaca, NY, USA, 1986.

[AVA10] AVANTSSAR. ASLan final version with dynamic service and policy composition. Deliverable D2.3, Automated Validation of Trust and Security of Service-oriented Architectures (AVANTSSAR), 2010. http://www.avantssar.eu/pdf/deliverables/avantssar-d2-3.pdf.

[AVI03] AVISPA. The Intermediate Format. Deliverable D2.3, Automated Validation of Internet Security Protocols and Applications (AVISPA), 2003. http://www.avispa-project.org/delivs/2.3/d2-3.pdf.

[BCP+08] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. Tvdc: managing security in the trusted virtual datacenter. *SIGOPS Oper. Syst. Rev.*, 42:40–47, January 2008.

[BDK12] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. DQMP: A decentralized protocol to enforce global quotas in cloud environments. In *Proceedings of the 14th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS '12)*, SSS '12, October 2012.

[BF07] Hitesh Ballani and Paul Francis. Conman: a step towards network manageability. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '07, pages 205–216, New York, NY, USA, 2007. ACM.

[BG11] Sören Bleikertz and Thomas Groß. A Virtualization Assurance Language for Isolation and Deployment. In *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2011)*, 2011. to appear.

[BGJ+05] Anthony Bussani, John Linwood Griffin, Bernhard Jansen, Klaus Julisch, Genter Karjoth, Hiroshi Maruyama, Megumi Nakamura, Ronald Perez, Matthias Schunter, Axel Tanner, and et al. Trusted virtual domains: Secure foundations for business and it services. *Science*, 23792, 2005.

[BGM11] Sören Bleikertz, Thomas Groß, and Sebastian Mödersheim. Automated verification of virtualized infrastructures. In *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*, CCSW '11, pages 47–58, New York, NY, USA, 2011. ACM.

[BGSE11] Sören Bleikertz, Thomas Groß, Matthias Schunter, and Konrad Eriksson. Automated information flow analysis of virtualized infrastructures. In *16th European Symposium on Research in Computer Security (ESORICS'11)*. Springer, Sep 2011.

[Bla01] Bruno Blanchet. An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In *14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 82–96, Cape Breton, Nova Scotia, Canada, June 2001. IEEE Computer Society.

[BMV05a] David Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *International Journal of Information Security*, 4(3):181–208, June 2005. Published online December 2004.

[BMV05b] David A. Basin, Sebastian Mödersheim, and Luca Viganò. OFMC: A symbolic model checker for security protocols. *Int. J. Inf. Sec.*, 4(3):181–208, 2005.

[BN89] D.F.C. Brewer and M.J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 206 –214, may 1989.

[Boi90] J. E. Boillat. Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience*, 2(4):289–313, 1990.

[BSP+10a] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 93–102, New York, NY, USA, 2010. ACM.

[BSP+10b] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 93–102, New York, NY, USA, 2010. ACM.

[Bur06] Mike Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. of the 7th Symp. on Operating Systems Design and Implementation*, pages 335–350, 2006.

[CDE+10] L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A.R. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted Virtual Domains–Design, Implementation and Lessons Learned. *Trusted Systems*, pages 156–179, 2010.

[CDRS07] Serdar Cabuk, Chris I. Dalton, HariGovind Ramasamy, and Matthias Schunter. Towards automated provisioning of secure virtualized networks. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 235–245, New York, NY, USA, 2007. ACM.

[CGJ+09] Richard Chow, Philippe Golle, Markus Jakobsson, Elaine Shi, Jessica Staddon, Ryusuke Masuoka, and Jesus Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, CCSW '09, pages 85–90, New York, NY, USA, 2009. ACM.

[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.

[CLZ99] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1):22–31, 1999.

[Cre09] Mache Creeger. Cloud computing: An overview. *ACM Queue*, 7(5), 2009.

[Cyb89] George Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel Distributed Computing*, 7(2):279–301, 1989.

[DDLS01] Nicodemos Damianou, Naranker Dulay, Emil Lupu, and Morris Sloman. The ponder policy specification language. In *Proceedings of the International Workshop on Policies for Distributed Systems and Networks*, POLICY '01, pages 18–38, London, UK, 2001. Springer-Verlag.

[DG04] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI04, 6th Symposium on Operating Systems Design and Implementation*, page 1, 2004.

[DH04] Scott Douglas and Aaron Harwood. Diffusive load balancing of loosely-synchronous parallel programs over peer-to-peer networks. *ArXiv Computer Science e-prints*, 2004.

[Dmi10] Catuogno L A Dmitrienko. Trusted virtual domains - design, implementation and lessons learned. *Lecture Notes in Computer Science including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*, 6163 LNCS:156–179, 2010.

[DMT10] DMTF. Open virtualization format specification. Technical report, DMTF, 2010.

[GEE+08] Peter Gardfjäll, Erik Elmrothaell, Erik Elmroth, Lennart Johnsson, Olle Mulmo, and Thomas Sandhol. Scalable grid-wide capacity allocation with the SweGrid Accounting System (SGAS). *Concurrency and Computation: Practice and Experience*, 20(18):2089–2122, 2008.

[Goo] Google App Engine. http://code.google.com/appengine/.

[GR05] 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.

[GSS+07] Yacine Gasmi, Ahmad-Reza Sadeghi, Patrick Stewin, Martin Unger, and N. Asokan. Beyond secure channels. In *Proceedings of the 2007 ACM workshop on Scalable trusted computing*, STC '07, pages 30–40, New York, NY, USA, 2007. ACM.

[HKS+08] Felix Hupfeld, Björn Kolbeck, Jan Stender, Mikael Högqvist, Toni Cortes, Jonathan Marti, and Jesús Malo. FaTLease: scalable fault-tolerant lease negotiation with Paxos. In *Proc. of the 17th Intl. Symp. on High Performance Distributed Computing*, pages 1–10, 2008.

[HNB11] Brian Hay, Kara L. Nance, and Matt Bishop. Storm Clouds Rising: Security Challenges for IaaS Cloud Computing. In *HICSS*, pages 1–7. IEEE Computer Society, 2011.

[HR04] Michael Huth and Mark Dermot Ryan. *Logic in computer science - modelling and reasoning about systems (2. ed.)*. Cambridge University Press, 2004.

[HRM10] S.M. Habib, S. Ries, and M. Muhlhauser. Cloud computing landscape and research challenges regarding trust and reputation. In *Ubiquitous Intelligence Computing and 7th International Conference on Autonomic Trusted Computing (UIC/ATC), 2010 7th International Conference on*, pages 410 –415, oct. 2010.

[Int98] International Organization for Standardization. *ISO/IEC 9798-3, Information technology — Security techniques — Entity authentication — Part 3: Mechanisms using digital signature techniques*, 2nd edition, 1998.

[Int06] International Organization for Standardization. *ISO/IEC 18033-2, Information technology — Security techniques — Encryption algorithms — Part 2: Asymmetric ciphers*, 2006.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11:256–290, April 2002.

[JNL10] Keith Jeffery and Burkhard NeideckerLutz. The Future of Cloud Computing — Opportunities For European Cloud Computing Beyond 2010, 2010.

[KFJ03] Lalana Kagal, Tim Finin, and Anupam Joshi. A policy language for a pervasive computing environment. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks*, POLICY '03, pages 63–, Washington, DC, USA, 2003. IEEE Computer Society.

[KLS08] Kfir Karmon, Liran Liss, and Assaf Schuster. GWiQ-P: An efficient decentralized grid-wide quota enforcement protocol. *SIGOPS OSR*, 42(1):111–118, 2008.

[KM10a] Khaled M. Khan and Qutaibah M. Malluhi. Establishing trust in cloud computing. *IT Professional*, pages 20–27, 2010.

[KM10b] Khaled M. Khan and Qutaibah M. Malluhi. Establishing trust in cloud computing. *IT Professional*, 12(5):20 –27, Sept 2010.

[KSS+09] Sunil D. Krothapalli, Xin Sun, Yu-Wei E. Sung, Suan Aik Yeo, and Sanjay G. Rao. A toolkit for automating and visualizing vlan configuration. In *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*, pages 63–70, New York, NY, USA, 2009. ACM.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Trans. Softw. Eng.*, 3:125–143, March 1977.

[LM09] John Lyle and Andrew Martin. On the feasibility of remote attestation for web services. In *SecureCom09: Proceedings of the International Symposium on Secure Computing*, pages 283–288. IEEE, 2009.

[MAM+99] M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 Internet Public Key Infrastructure Online Certificate Status Protocol — OCSP. RFC 2560, Internet Engineering Task Force, June 1999.

[MG09] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, 2009. http://csrc.nist.gov/groups/SNS/cloud-computing/clouddefv15.doc.

[MGHW09] Jeanna Matthews, Tal Garfinkel, Christofer Hoff, and Jeff Wheeler. Virtual machine contracts for datacenter and cloud computing environments. In *Proceedings of the 1st workshop on Automated control for datacenters and clouds*, ACDC '09, pages 25–30, New York, NY, USA, 2009. ACM.

[Mic09] Sun Microsystems. Take Your Business to a Higher Level, 2009.

[MLQ+10] 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.

[Nar05] Sanjai Narain. Network configuration management via model finding. In *Proceedings of the 19th conference on Large Installation System Administration Conference - Volume 19*, LISA '05, pages 15–15, Berkeley, CA, USA, 2005. USENIX Association.

[NCPT06] Sanjai Narain, Y.-H. Alice Cheng, Alex Poylisher, and Rajesh Talpade. Network single point of failure analysis via model finding. In *Proceedings of First Alloy Workshop*, 2006.

[NNS02] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. A succinct solver for alfp. *Nordic J. of Computing*, 9:335–372, December 2002.

[OCJ08] Jon Oberheide, Evan Cooke, and Farnam Jahanian. Exploiting Live Virtual Machine Migration. In *BlackHat DC Briefings*, Washington DC, February 2008.

[OGP03] David Oppenheimer, Archana Ganapathi, and David A. Patterson. Why do internet services fail, and what can be done about it? In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, Berkeley, CA, USA, 2003. USENIX Association.

[Ope10a] OpenSource. OpenNebula, 2010. http://www.opennebula.org/.

[Ope10b] OpenSource. OpenStack, 2010. http://www.openstack.org/.

[Ope11] OpenStack. OpenStack Compute — Administration Manual, 2011. http://docs.openstack.org.

[Ora11] Oracle. Oracle Real Application Clusters (RAC), 2011. http://www.oracle.com/technetwork/database/clustering/overview/index.html.

[Pau94] Lawrence C. Paulson. *Isabelle: a Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer – Berlin, 1994.

[Pea02] S. Pearson. *Trusted computing platforms: TCPA technology in context*. Publisher: Prentice Hall PTR, 2002.

[PEC05] M. Peinado, P. England, and Y. Chen. An overview of ngscb. In Chris J. Mitchell, editor, *Trusted Computing*, pages 115–141. IEE, 2005.

[PLG+07] Kristal T. Pollack, Darrell D. E. Long, Richard A. Golding, Ralph A. Becker-Szendy, and Benjamin Reed. Quota enforcement for high-performance distributed storage systems. In *Proc. of the 24th Conf. on Mass Storage Systems and Technologies*, pages 72–86, 2007.

[RA00] Ronald W. Ritchey and Paul Ammann. Using Model Checking to Analyze Network Vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 156–, Washington, DC, USA, 2000. IEEE Computer Society.

[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.

[RCAM06] Jerry Rolia, Ludmila Cherkasova, Martin Arlitt, and Vijay Machiraju. Supporting application quality of service in shared resource pools. *Communications of the ACM*, 49(3):55–60, 2006.

[RM11] Anbang Ruan and Andrew Martin. Repcloud: achieving fine-grained cloud tcb attestation with reputation systems. In *Proceedings of the sixth ACM workshop on Scalable trusted computing*, STC '11, pages 3–14. ACM, 2011.

[RTSS09a] 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.

[RTSS09b] 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 *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[RVR+07] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proc. of the 2007 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 337–348, 2007.

[Sch04] Jennifer M. Schopf. Ten actions when Grid scheduling: the user as a Grid scheduler. In *Grid Resource Management: State of the Art and Future Trends*, chapter 2, pages 15–23. Kluwer Academic Publishers, 2004.

[SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *In Proceedings of the 2009 conference on Hot topics in cloud computing*, Berkeley, CA, USA, 2009. USENIX Association.

[SMV+10] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *Proceedings of the 2010 ACM workshop on Cloud computing security workshop*, CCSW '10, pages 43–46, New York, NY, USA, 2010. ACM.

[SVJ+05] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Doorn, John Linwood, and Griffin Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. Technical Report RC23511, IBM Research, 2005.

[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 conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 16–16, Berkeley, CA, USA, 2004. USENIX Association.

[TE90] L. Tassiulas and A. Ephremides. Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks. In *Proc. of the 29th IEEE Conf. on Decision and Control*, pages 2130–2132, 1990.

[Trua] Privacy ca. http://www.privacyca.com.

[Trub] Trousers - the open-source tcg software stack. http://trousers.sourceforge.net/.

[Truc] Trusted computing group. http://www.trustedcomputinggroup.org.

[Trud] Trusted grub. http://trousers.sourceforge.net/grub.html.

[Tru06] Infrastructure work group platform trust services interface specification, version 1.0. http://www.trustedcomputinggroup.org/resources/infrastructure_work_group_platform_trust_services_interface_specification_version_10, 2006.

[Tru07a] Trusted Computing Group. *TPM Main, Part 1, Design Principles. Specification version 1.2 Revision 103*, 2007.

[Tru07b] Trusted Computing Group. *TPM Main, Part 2, TPM Structures. Specification version 1.2 Revision 103*, 2007.

[Tru07c] Trusted Computing Group. *TPM Main, Part 3, Commands. Specification version 1.2 Revision 103*, 2007.

[Tru11] Open platform trusted service user's guide. http://iij.dl.sourceforge.jp/openpts/51879/userguide-0.2.4.pdf, 2011.

[Tur06] Mathieu Turuani. The cl-atse protocol analyser. In Frank Pfenning, editor, *Term Rewriting and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 277–286. Springer Berlin / Heidelberg, 2006.

[UOI06] Masato Uchida, Kei Ohnishi, and Kento Ichikawa. Dynamic storage load balancing with analogy to thermal diffusion for P2P file sharing. In *Proc. of the 2006 Work. on Interdisciplinary Systems Approach in Performance Evaluation and Design of Computer & Communications Systems*, 2006.

[VMw10] VMware. VMware vCenter Server, 2010. http://www.vmware.com/products/vcenter-server/.

[WB09] Craig D. Weissman and Steve Bobrowski. The design of the Force.com multitenant Internet application development platform. In *Proc. of the 35th SIGMOD Intl. Conf. on Management of Data*, pages 889–896, 2009.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. Spass version 3.5. In Renate Schmidt, editor, *Automated Deduction CADE-22*, volume 5663 of *Lecture Notes in Computer Science*, pages 140–145. Springer Berlin / Heidelberg, 2009.

[Win] Windows Azure Platform. http://www.microsoft.com/windowsazure/.

[XBL05] Lin Xiao, Stephen Boyd, and Sanjay Lall. A scheme for robust distributed sensor fusion based on average consensus. In *Proc. of the 4th Intl. Symp. on Information Processing in Sensor Networks*, pages 63–70, 2005.

[XZM+04] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks, 2004.

[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.