D2.3.4
Automation and Evaluation 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 / D2.3.4 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.3
Due date:	September 2013 – M36
Actual submission date:	30th September, 2013

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

Abstract:	This report presents revisions and extensions to components for security management, automation of security configuration, and security evaluation.
Keywords:	Security management, isolation, modelling and verification, trust anchors, provenance.

Editor

Sören Bleikertz (IBM)

Contributors

Sören Bleikertz (IBM)

Jan-Niklas Fingerle, Norbert Schirmer (SRX)

Imad M. Abbadi (OXFD)

Roberto Sassu (POL)

Mihai Bucicoiu (TUDA)

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

Executive Summary

In this deliverable we describe the revisions and extensions of the TClouds subsystems that deal with security management, automation of security configuration, and security evaluation. In particular, we extended the automated security analysis of virtualized infrastructures to handle dynamic infrastructures and propose a new proactive analysis approach. Furthermore, the central management component of the TrustedInfrastructure Cloud has been revised in terms of usability, e.g., by a new clean separation of security and infrastructure management in the user interface, and in terms of end-to-end security, in order to support multi-tenant environments.

The automated deployment of a security network configuration that implements a concept of Trusted Virtual Domains has been finalized and integrated into the OpenStack networking service. In the area of trust establishment and trust anchors, we discuss solutions for implementing large-scale cloud infrastructure requirements for trust anchors, in particular related to scalability and remote attestation. Finally, we propose a cloud provenance framework in order to support the trust establishment.

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. In the final and third year, the subsystems have been revised, extended if necessary, and they have been finalized.

1.4 Deliverable 2.3.4 — Automation and Evaluation of Security Configuration and Privacy Management

Overview. This deliverable describes the revisions and extensions of the subsystems relevant to security management, automation of security configuration, and security evaluation. In particular, the work on the automated security analysis of virtualized infrastructures (cf. D2.3.1, Cha. 8; D2.3.2, Cha. 4) has been revised and extended to cope with dynamic infrastructures. The central management component of the TrustedInfrastructure Cloud has been revised in terms of usability as well as end-to-end security. Furthermore, the automated deployment of a security network configuration has been finalized and integrated into OpenStack.

In terms of trust establishment and trust anchors, we discuss solutions for implementing the requirements of large-scale cloud infrastructures, and we propose a provenance framework as an extension of secure virtual layer management (D2.3.2, Cha. 5.1).

Structure. The deliverable is structured in the following way: Chapter 2 introduces the security analysis of dynamic virtualized infrastructure as an extension to previous automated analysis approaches for static infrastructures. In Chapter 3 the central management component of the

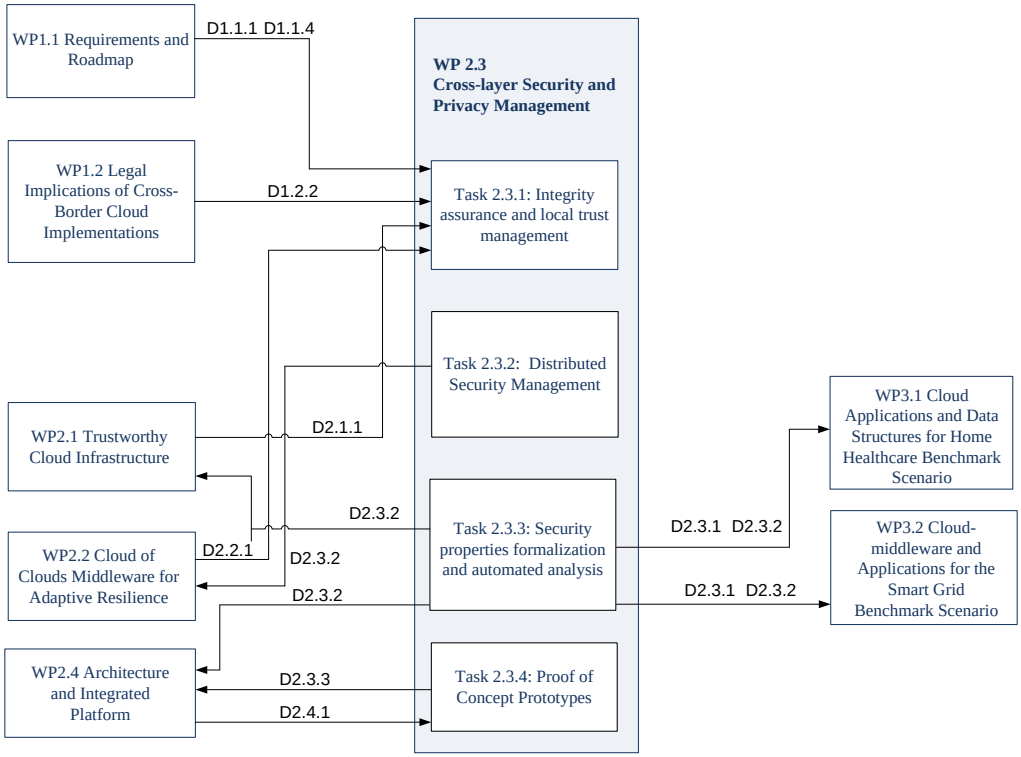

Figure 1.1: Graphical structure of WP2.3 and relations to other workpackages.

TrustedInfrastructure Cloud is revised in terms of usability and end-to-end security. Solutions for the large-scale requirements of trust anchors are discussed in Chapter 4. An automated security network configuration approach is presented in Chapter 5. Finally, Chapter 6 describes a cloud provenance framework for establishing trust. This chapter appeared also as [Abb13].

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).

Chapter 2

Security Analysis of Dynamic Infrastructure Clouds

Chapter Authors:
Sören Bleikertz (IBM)

2.1 Introduction

Multi-tenant infrastructure clouds can bear great complexity and can be exposed to security issues. Even if one only analyses the static configuration of hosts, VMs, network, and storage as well as their inter-connectivity, one faces a complex system. Configuration and topology changes make the system a moving target and can introduce violations of a security policy, for instance, manifest in incorrect deployment or isolation breaches. Although there have been analyses of isolation failures in complex static configurations of clouds, such as Bleikertz et al. [BGSE11] (D2.3.1, Cha. 8), the analysis of dynamic configuration changes is largely missing [BGM11] (D2.3.2, Cha. 4).

Misconfigurations and Insider Attacks Even if committed unintentionally, misconfigurations are among the most prominent causes for security failures in infrastructure clouds. The ENISA report on cloud security risks [ENI09] names isolation failure as major technical risk, with misconfiguration, lack of resource isolation, and ill-defined role-based access policies as notable root causes. If committed intentionally by a malicious insider, misconfigurations expose the infrastructure to greater risks. The CSA threat report [CSA10] as well as the ENISA report agree to insider attacks as a TOP 10 cloud security risk as well as malicious insiders as a "very high impact" technical risk [ENI09]. The analysis of all configuration changes is crucial here, as the insider could create a transient insecure state to attack the system and change it to a secure configuration before the next security analysis. Therefore, there is a need to analyse management operations for their security properties before they are applied, and to achieve overall accountability for administrator actions. The question is: How can we model configuration changes induced by management operations of cloud administrators and check these changes for violations of a security policy?

Analysis of Dynamic Infrastructure Clouds The concrete aim of our research is to mitigate the security impact of misconfigurations: (a) We enable honest administrators to create a change plan for an infrastructure in advance and have this change plan checked for violations of the

security policy in a what-if analysis. (b) We establish an authorisation proxy to have all configuration changes independently checked for violations of the security policy, creating a run-time audit for misconfigurations and their security impact. (c) We direct this research towards the run-time mitigation of misconfigurations and enforcement of security policies. Whereas (b) only establishes an audit log of misconfigurations causing security problems, (c) is aiming at enforcing the security policy.

To achieve this analysis, we need multiple ingredients. First, we need a faithful representation of the topology and configuration of the virtualised infrastructure, which we call a realisation model. Second, we need a description of information flow traversal of infrastructure components to evaluate information flow as the key security aspect. We draw upon our previous work for both points. Third, we need a language to specify security goals for virtualised infrastructures, where we resort to the language *VALID* [BG11], already used in research prototypes for that very purpose.

A crucial piece missing for a dynamic system analysis, however, is a formal model of topology and configuration changes in infrastructure clouds. The model needs to capture how relevant operations change the topology and its security properties. Such a model needs to capture basic operations, such as VMware's UpdatePortGroup as an operation that changes the VLAN configuration, or larger asynchronously executed tasks, such as creating or migrating a virtual machine. Expressing such cloud operations in a formal system is challenging: If the model is too abstract, it may fail to capture many significant attacks, while if it is too detailed, formal reasoning about security in the model becomes infeasible—both for manual and for automated verification. The main contribution of this chapter is to develop an *operations model*, which enables us to establish an analysis system for misconfigurations.

Our approach builds upon graph rewriting, where topology, security goals, and configuration changes are expressed as graphs and transformations of graphs. This methodology allows us to check efficiently whether configuration changes applied to a given topology will violate the security goals or not. We realise a prototype tool for this analysis as well as application cases in change planning and run-time audit of misconfigurations and establish this analysis in a practical environment with VMware.

2.2 System and Threat Model

2.2.1 A Graph Model of Virtualised Infrastructures

A graph-based model of static virtualised infrastructures has been proposed in [BGSE11]. The vertices of such a graph represents the virtualised infrastructure elements, e.g., physical servers or virtual machines, and the edges model the relationship among the elements, thereby capturing the topology of the system. Furthermore, nodes are typed, e.g., vMachine, and attributed to capture further properties and configuration aspects of each element. We consider the model as a directed, node and edge typed, and attributed graph.

In order to build the model we require sufficient information on the topology and configuration of a virtualised infrastructure. Two steps are required for the model construction: *Discovery* (§4.1) and *Translation* (§4.2). The discovery extracts the configuration from the hypervisors or management system of heterogeneous virtualised infrastructures, and translates the extracted configuration data into the model.

2.2.2 Threat Model

As core threat, we model non-malicious (deliberate and non-deliberate, accidental and incompetence) human-made faults. Even if administrators are honest, they can still make mistakes that lead to a security breach. Therefore, their behavior is not malicious and byzantine, but comes with some fairness constraints: A provider administrator will attempt to issue commands in a well-defined way through the service interface. A provider's behavior will take feedback from a security analysis or an audit into account. As part of the system foundations, we require that the analysis probes discovering the infrastructure configuration get an authentic view of the configuration.

In general, the analysis method proposed is capable to handle malicious, byzantine adversaries as well. To protect against those, the infrastructure needs to be modified to enforce sole access through the management interface (and prevent circumvention approaches, such as a direct SSH log-in to the physical hosts). Also, the security validation needs to be mandatory for all management operations, which is part of future work.

2.3 Modeling and Analysis of Dynamic Infrastructure Clouds

2.3.1 A Model of Dynamic Virtualised Infrastructures

The core of our model are graph transformations and we introduce a novel modelling of management operations and their impact on the configuration and topology of a virtualised infrastructure given as a graph. Furthermore, we integrate existing approaches for the formalisation and analysis of security goals for virtualised infrastructure in our model, thereby establishing a unified approach. Additionally, these existing approaches are extended to enable the security analysis of dynamic virtualised infrastructures.

The basic idea of graph transformation is as follows. We have a graph transformation rule p, also called a *production*, in the form of $p : L \xrightarrow{r} R$, where graphs L and R are denoted the left hand side (LHS) and right hand side (RHS), respectively. A partial graph morphism r, the *production morphism*, establishes a partial correspondence between elements in the LHS and the RHS of a production, which determines the nodes and edges that have to be preserved, deleted, or created. A *match* m finds an occurrence of L in a given graph G, then $G \overset{p,m}{\Longrightarrow} H$ is an application of a production p, where H is a derived graph. H is obtained by replacing the occurrence of L in G with R.

Our unified model forms a *graph grammar* that consists of a *start graph* and a collection of *productions*, which transform the start graph. In our case, the start graph is a graph model of the virtualised infrastructure, and the productions represent our model of topological changes, information flow analysis, and policy specification.

2.3.1.1 Modelling of Infrastructure Changes

The *Operations Transition Model* captures the changes to the topology and configuration of a virtualised infrastructure through management operations. Our goal is a practical security system for virtualised infrastructures, therefore we focus our modelling efforts, as an example, on VMware and its management operations. Each management operation is modelled as a graph production that transforms the virtualised infrastructures, which is modelled as a graph, into a modified one.

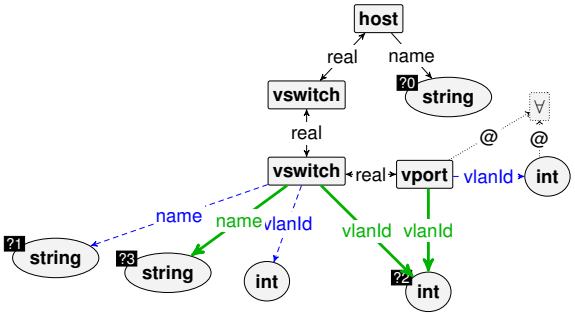

Figure 2.1: PortGroup Operation in *GROOVE*: UpdatePortGroup

For any existing real-world virtualised infrastructures like VMware, the API documentation does not provide a precise formal description and model, but rather a semi-formal description of the operations, parameters, as well as the preconditions and effects that the operations have. A contribution of this work is to obtain a formal model that allows for precise statements to be made and proved or refuted. It is of course not possible to formally prove the correctness of such a model itself, however there is a good methodology to obtain a "good" model by combining two directions.

The first direction is to follow the documentation and translate the documented effects into our (abstract) graph model. The second is to experiment with the real implementation, to verify that the operations indeed do have the effect on the infrastructure that our model predicts. To study these experiments we need to translate the real infrastructure topology into our abstract graph before and after the operation has been performed, and check that the resulting graph transformation coincides with our model of the operation.

An example of a operations model for the VMware operation UpdatePortGroup is given in Figure 2.1. Using this operation, an administrator can change the configuration on an existing portgroup. The portgroup is identified by its name, as well as the host where it resides on, and the operation allows to change the portgroup's name and VLAN ID. Changing attributes is modelled as changing the edges to different data nodes based on the input parameters. In order to maintain compatibility with the existing graph model, not only does the portgroup node contain the VLAN ID, but also the associated vport nodes, i.e., virtual switch ports. Therefore, changing the VLAN ID of the portgroup also requires to change the VLAN ID of all virtual ports associated to that portgroup. For this we use a universal quantifier ∀ that updates the vlanId attributed of all matching vport nodes [RK09].

2.3.1.2 Dynamic Information Flow Analysis & Security Policies

An approach that has been presented in [BGSE11] performs an information flow analysis on a graph-based model of a virtualised infrastructure, in order to detect isolation failures. The approach consists of a set of *traversal rules* that captures how elements in the infrastructure provide isolation, e.g., VLANs provide network isolation. A graph traversal guided by the set of traversal rules computes the transitive closure and determines the information flow in the system.

This information flow analysis is so far limited to a static snapshot of the network and thus unable to deal with the dynamic nature and frequent changes of such infrastructures. We need to integrate this information flow analysis into our dynamic graph model and thus obtain a dynamic information flow analysis. We do so by expressing the traversal rules of the information flow analysis as graph production rules. We use the graph rewriting control language of GROOVE

to compose the information flow analysis from the rules.

The final part of our unified model deals with the formalisation of security policies, which describe properties of the topology and configuration of an infrastructure cloud. We follow here the approach of the policy language *VALID* [BG11]. Such policies can also be expressed as graph production rules, where a rule matches parts of the graph with potential additional conditions on this match. Typically, one would express a *violation* of a security policy as a graph production rule, and try to match the rule on the evolving graph. Once a match is found, a violation of the security policy is found.

2.3.2 Automated Analysis using *GROOVE*

Many graph transformation tools have been developed in the past, among them: *GROOVE* [GdR+11, Ren], AGG [Tae03], GRGen [GBG+06], and PROGRES [SWZ99]. For this work, we decided to use *GROOVE* as our graph transformation environment. *GROOVE* is a general-purpose graph transformation tool that enables an expressive specification of production rules, e.g., by providing nested quantifications and path constructions using regular expressions on edge labels. Furthermore, an imperative control language allows to schedule the application of rules, which also allows more complex control flow constructions, as well as enabling parametrised rules, where parameters are passed from a control program to a production rule. We refer to a detailed comparison between different graph transformation tools to [GdR+11].

2.3.3 Application Scenarios of our Analysis System

2.3.3.1 Change Planning

The administrator specifies the sequence of change requests (either directly as a change program in *GROOVE*'s control language or in a proprietary provisioning language, translated to it). Once the change program is submitted to *GROOVE*, the tool will apply the changes to the realisation model, derived from the actual infrastructure. By that, the tool can establish a what-if analysis and determine what security impact the intended changes will have on the infrastructure.

If the new realisation model obtained from the execution of the change program violates the *VALID* security goals, the tool notifies the administrator to reject the proposed change requests and provides the *GROOVE* output of the matched attack sub-graph as diagnosis. Otherwise, the tool returns that the intended changes are compliant with the security goals, after which the administrator can provision the changes to the infrastructure.

2.3.3.2 Runtime Audit of Misconfigurations

Run-time audit of misconfigurations expands on the principles of the change planning. Whereas change planning requires the administrator to devise the changes in advance and have them checked by our analysis statically, the run-time audit intercepts change requests dynamically at an authorisation proxy and checks them concurrently as they occur and before they are applied. The idea of the run-time auditing is to establish accountability for administrator actions: administrator's configuration changes are validated against the security policy and the results of these checks entered into the audit logs along with the administrator's username and the committed commands. We introduce an authorisation proxy as wrapper of the administration API, which acts as policy enforcement point (PEP) and auditor on configuration changes, and employ our analysis as part of the policy decision mechanism.

2.4 Conclusions

In this work, we started to tackle the problem of misconfigurations in virtualized infrastructures. Our first solution consists of a practical security system that employs a formal model of cloud management operations in order to proactively assess their security impact. Building our modeling efforts upon graph transformation, we offer a unified approach in an intuitive and expressive form. We propose two application scenarios for our system: change planning and run-time auditing.

Chapter 3

Revised Security Management for the Trusted Infrastructure Cloud

Chapter Authors:
Jan-Niklas Fingerle, Norbert Schirmer (SRX)

3.1 Introduction

In the TrustedInfrastructure Cloud (cf. Deliverable D2.1.1 Chapter 13) a central management component, called TrustedObjects Manager (TOM), manages a set of appliances, in case of TClouds these are the TrustedServers (cf. Figure 3.1).

The TOM offers a web based user interface towards the administrator to manage the infrastructure and the security policies. A central security concept of the TrustedInfrastructure cloud is the Trusted Virtual Domain (TVD): a trustworthy virtual overlay on the physical resources (computing, networking, storage) providing strong isolation guarantees. In this chapter we describe a revision of the TOM which resulted from the feedback and experience we made in the first two project years. The main drawbacks of the old version were, that there was no strict separation between TVDs and other components like VPNs and that their was no proper support for a multi-tenant and multi-product environment. The revision done in the third year improved this in two directions:

- Usability: the interface and the database has been reworked to separate the general security management (TVDs) from the infrastructure management. This design highlights the fact, that the security settings should be centrally managed as they have effect on all entities of the infrastructure. Therefore it is preferable to have them accessible on a single spot in the interface.

- End-to-end security: to get the most security benefits out of the concept of the TVDs the endpoints accessing the cloud have to be taken into consideration as well (cf. Deliverable D2.1.2 Chapter 5, Deliverable D2.1.3 Chapter 2.2). Only then a trustworthy end-to-end security can be established. The core components of the TOM were revised to support this multi-tenant, and multi-product environment.

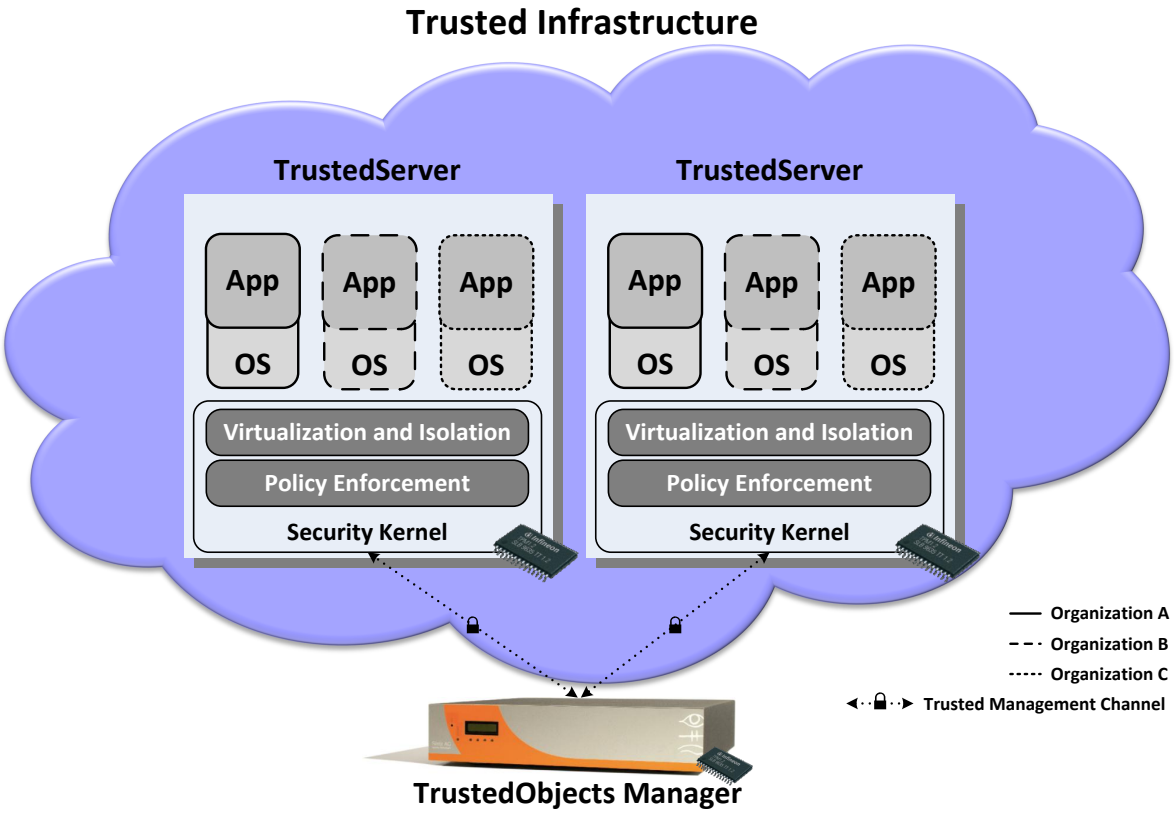

Figure 3.1: TrustedInfrastructure Cloud

3.2 TrustedObjects Manager User Interface

3.2.1 Integrated Configuration

The current version of the central configuration server TOM provides for an integrated configuration approach. The TrustedObjects Manager has been designed to work with multiple products like TrustedServer, TrustedDesktop, TrustedVPN, TrustedDisk, etc. In the old version, this was achieved by using the TrustedObjects Manager as a base platform and installing different functional packages on top of it. The revised TrustedObjects Manager has all the available functionality integrated, restricted by licenses and choices on a per organization basis (cf. Figure 3.2).

This integrated approach allows for a generalized, global configuration of the security policy. Functionality has been moved from the specialized products to the general, global TrustedObjects Manager layer to enable reuse and interconnection between different products, especially with regards to the security configuration. The configuration is globally defined and is propagated to the product specific settings. This ensures that every product is aware of the global settings, especially the security policy.

The TVD integrates all security needs and is enforced throughout the TrustedObjects manager and in all products.

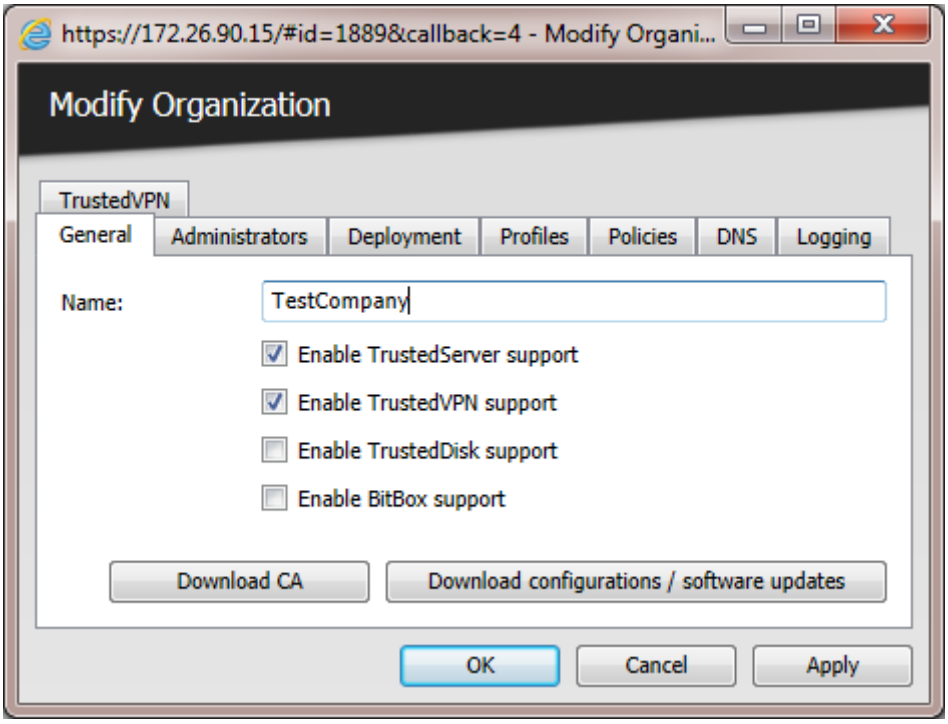

Figure 3.2: Multiproduct support for a Organisation

All configuration of interoperation between different products is based on Trusted Virtual Domains that have been extended to contain network encryption settings for a generalized virtual network approach (cf. Figure 3.3). Virtual networks – the former VPNs – now exist as pure network routing configuration inside a TVD (cf. Figure 3.4).

The TrustedObjects Manager will allow the flow of information between different TVDs only if this is explicitly configured. This can be done with the configuration of Information Flows. TVDs and Information Flows capture the security policy of an infrastructure on the most abstract level (cf. Figure 3.5). The default is that information flow is allowed within the same TVD and prohibited between TVDs. As such, each product has to comply to this policy. Each product may enable the user to do at most what is allowed in the TVD and Information Flow setting, but may opt to allow less.

3.2.2 Configuration of TrustedServer components

The revised TrustedObjects Manager interface allows a more user friendly configuration of almost every configuration aspect of the infrastructure (cf. Figure 3.6). This had been of much concern in older versions of the TrustedObjects Manager, as the administrator had to provide many idiosyncratic and internal configuration options. This was unnecessarily complex and prohibited automatic enforcement of the security policy.

One central service – used not only by TrustedServer but also by other products out of the scope of this document – is the “Disk Manager” (cf. Figure 3.7). Disk images (of virtual machines) are uploaded in one central repository and are referenced when needed. This virtual machines are named compartments in the context of TVDs.

IP pools are used to automatically assign IPs from a valid range. This is needed to automate the Integration off multiple installations of the same compartment template. Individual IPs no

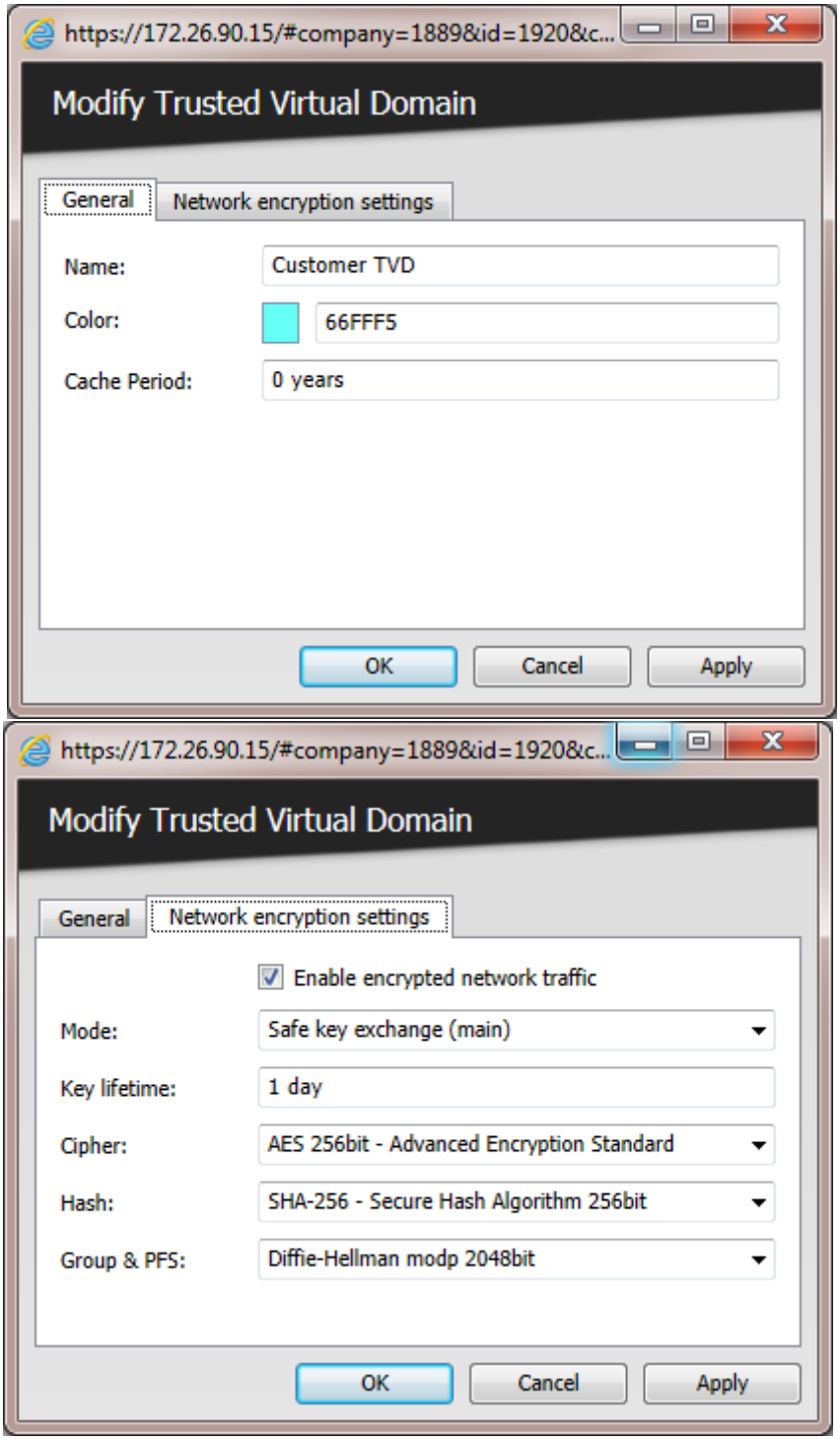

Figure 3.3: TVD settings

longer have to be assigned to a specific installation (while the administrator may still chose to do so anyway) (cf. Figure 3.8 on page 16).

In this way we can configure compartment templates that provide a blueprint for individual compartment installations (cf. Figure 3.9). One can use the same disk image for multiple compartment templates. A compartment templates pre-sets almost all needed configuration for a compartment installation, leaving out only the owning user and the machine(s) where the

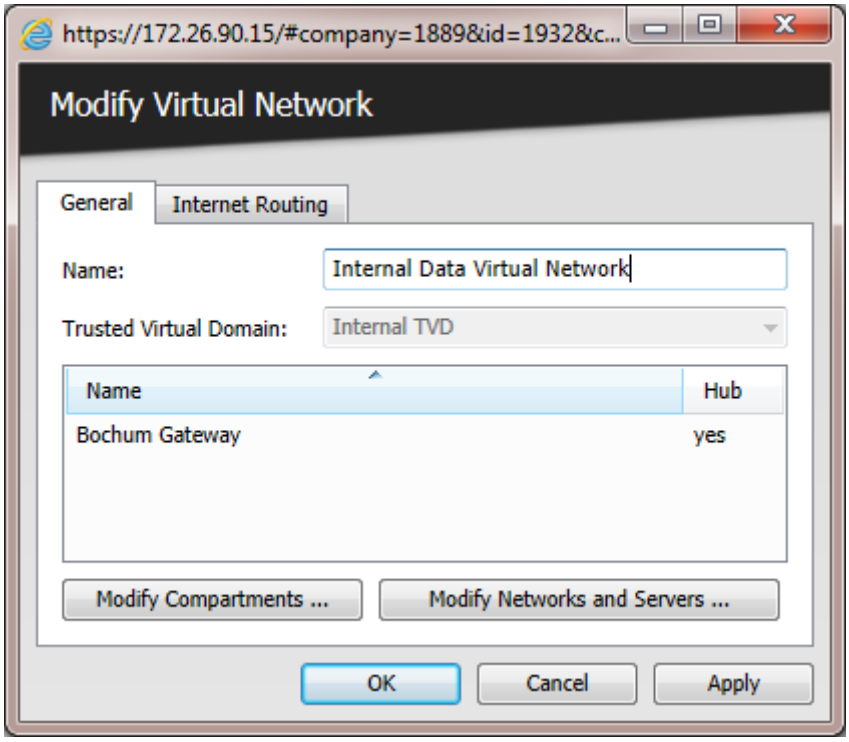

Figure 3.4: Virtual Network

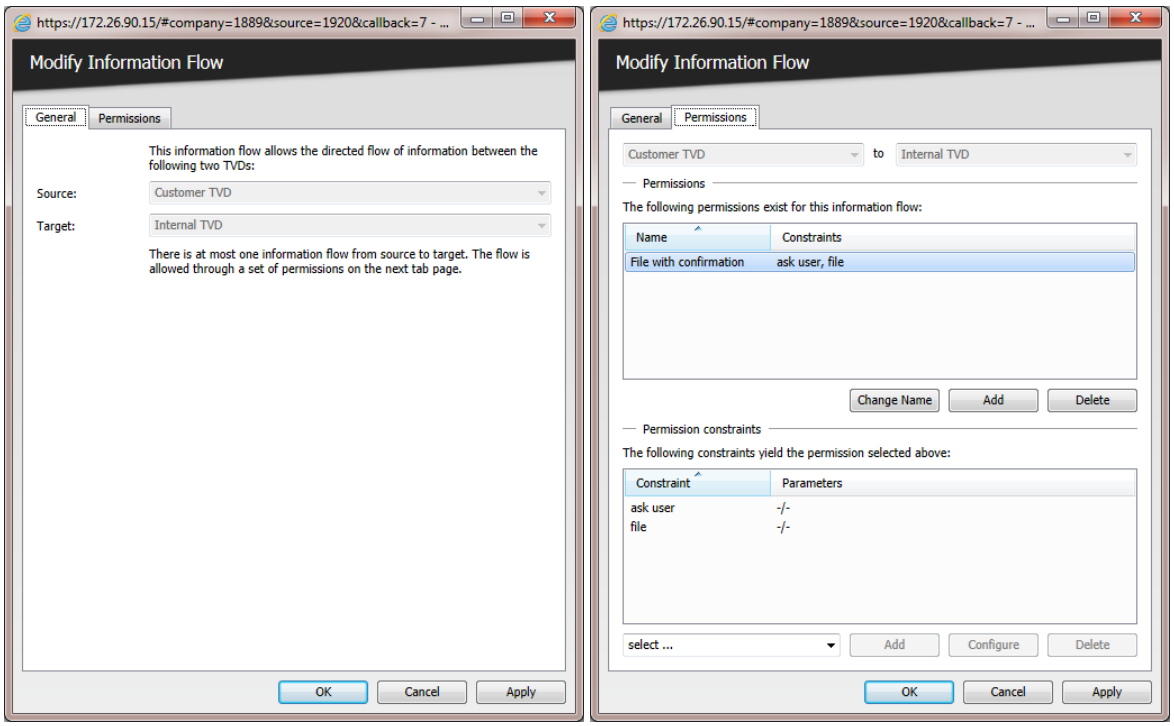

Figure 3.5: Information Flow Settings

installation will be installed.

Compartment templates are used to add a compartment to a user (cf. Figure 3.10). On a

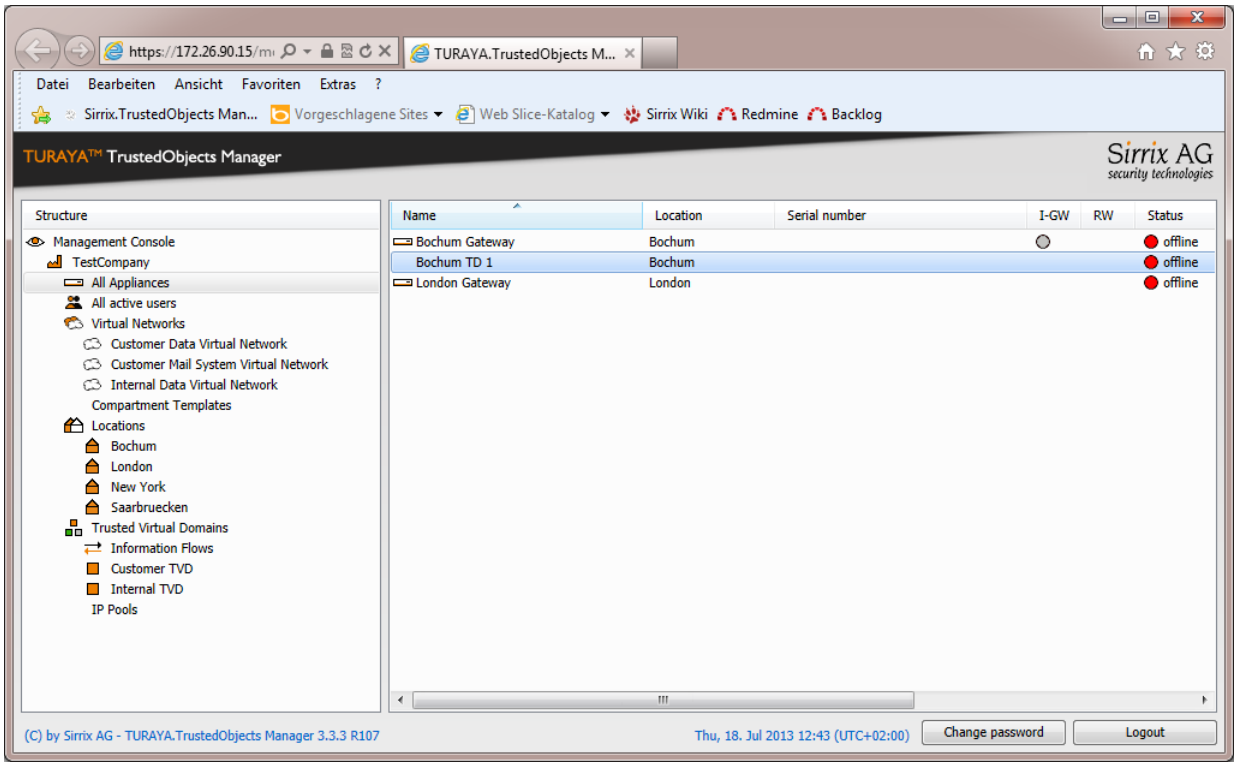

Figure 3.6: Overview of an Organisation

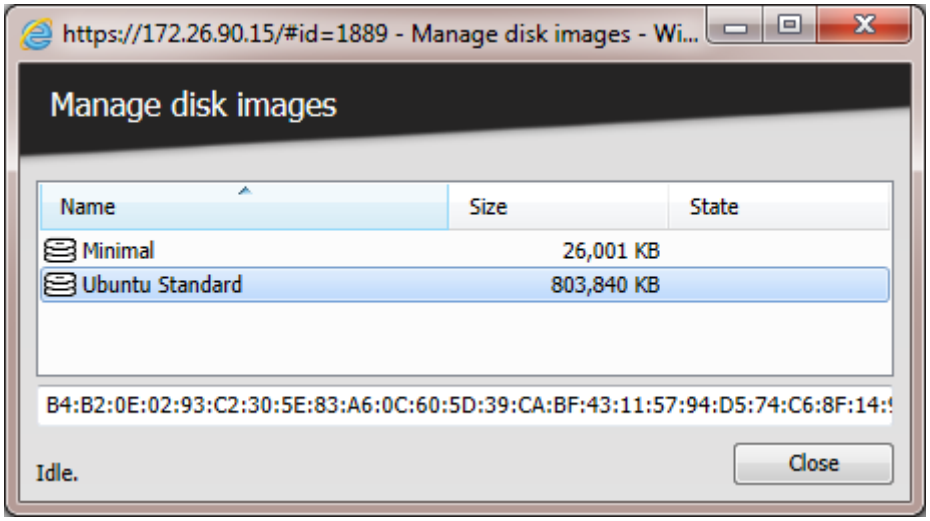

Figure 3.7: Disk Manager

TrustedServer this is usually a system user, as a VM on a server typically implements a service. In contrast on a TrustedDesktop the user identifies the user of the desktop system, in case the desktop system is shared among multiple users.

A compartment template may be configured to be member of a virtual network, together with other virtual network components like VPN networks and servers.

The key to user friendliness in the revised TrustedObjects Manager is generalization: The

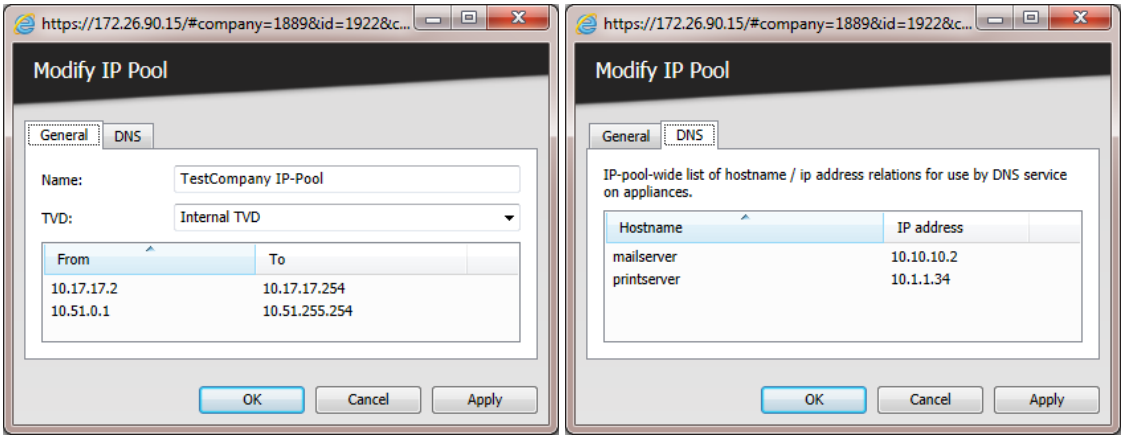

Figure 3.8: IP Pool Settings

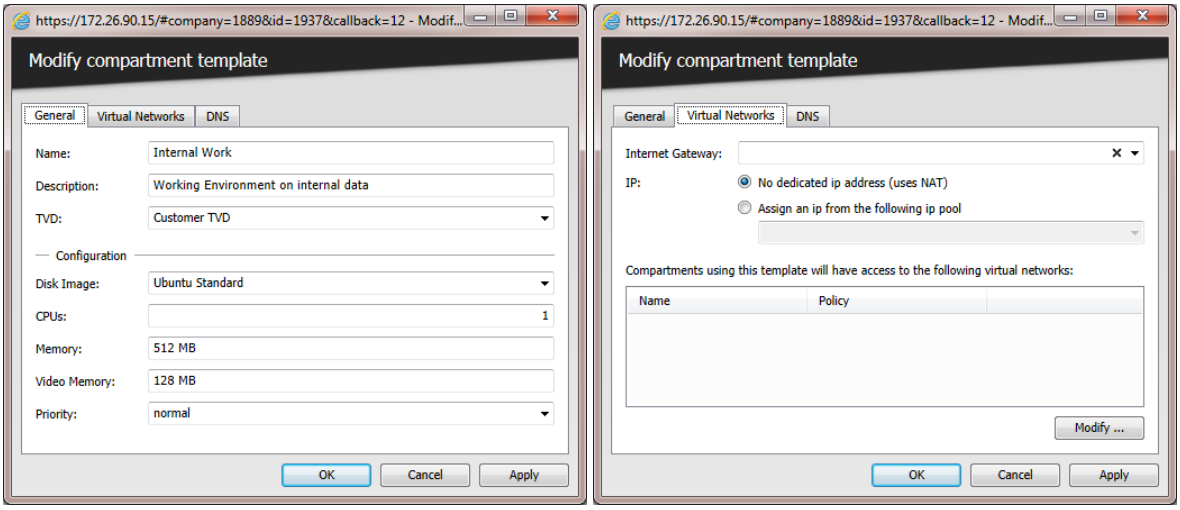

Figure 3.9: Compartment Template

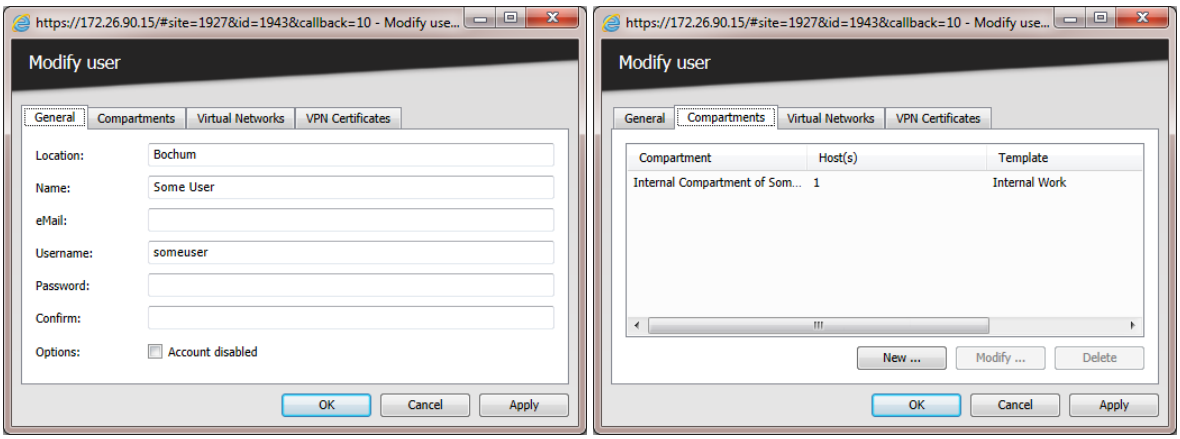

Figure 3.10: User Settings

parallel installations of virtualized systems (“compartments”) should not require the administrator to configure every aspect about each compartment separately (cf. Figure 3.11). Instead, we base individual compartment installations (“an instance of a system that may live on one machine or multiple machines with migratable state”) on compartment templates, generic descriptions, which allow for multiple compartment installations to be based upon. These compartment templates, again, refer to common, generalized objects like disk images or IP pools, as shown below.

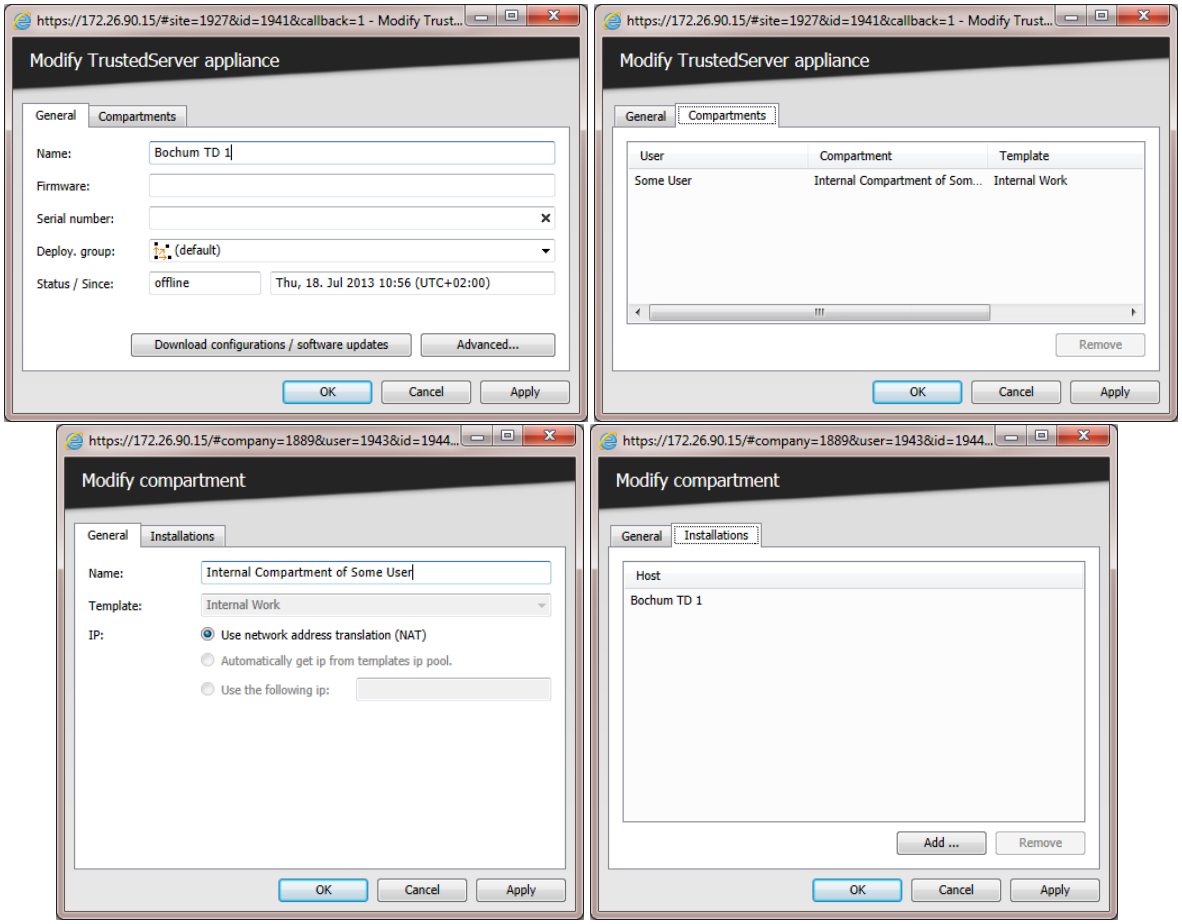

Figure 3.11: Server and Compartment Settings

This installation has then to be added to one or more TrustedServer hosts. If it is added to multiple hosts, migration of system state between theses hosts may be used. All these changes lead to a simpler and more consistent configuration compared to the old TrustedObjects Manager:

- The generalized configuration reduces the number clicks needed significantly.
- TVD security is enforced throughout the whole system, ensuring interoperability between different products.
- The administrator no longer needs to configure technical idiosyncrasies like tun-devices with special IPs for VPN integration.

To achieve these changes, not only the GUI was modified; much more effort went into the TrustedObjects Manager backend as well as in improvements on the TrustedChannel management protocol, which are described in the next chapter.

3.3 Appliance Management

All TrustedObjects Manager managed appliances are managed through the TrustedChannel management protocol. This protocol is organized in several layers.

- Encryption and Remote Attestation Layer: This layer sets up the trusted channel in a trustworthy manner (cf. Deliverable D2.4.1 Chapter 9.3)
- Binary Data Stream Layer: This layer deals with the data transfer for the management communication between TOM and TrustedServer, once the trusted channel above has be established
- Functional layer: The management protocol for the TrustedServer itself.

The revision has affected the Functional layer and reflects the newly achieved modularity which we have described for the user interface before: The clean separation of TVDs as global setting, and the multi-tenant and multi-product support.

3.3.1 Encryption and Remote Attestation Layer

The outer layer is based on TLS. It extends TLS by including remote attestation through Trusted Platform Modules. Therefore the TrustedObjects Manager can trust the client not to be compromised. Some of our product clients are not managed appliances, but unmanaged legacy systems, eg. Windows or Linux machines (cf. Deliverable D2.3.2 Chapter 5.2). In these special cases, the remote attestation will be omitted, falling back to the regular TLS handshake. The TrustedObjects Manager keeps a list of machines which are allowed do use the legacy protocol, so this won't compromise the security of remote attestation.

3.3.2 Binary Data Stream Layer

Inside the established TLS channel, this layer establishes the message passing layer. The data is transmitted in a sparse binary data stream. It uses multiple techniques like command dictionaries and ASN.1-like data encoding, to keep the bytes sent at a minimum.

3.3.3 Functional Layer

The messages are dispatched to a kind of remote object invocation. Messages in the TrustedChannel are transferred between two objects on both ends of the communication channel. The protocol identifies on the functional layer the communication between two such communicating objects as independent "jobs". Calling a method of an interface may create a new object and therefore create a new job. This allows for an extensible approach, which is used to support the TrustedObjects Managers multi-product design. In Appendix A, you find the functional protocol for the TrustedServer. It shows, which jobs are created and which messages are sent ("<-" being the direction from the TOM (Server) to the Client (TrustedServer) and "->" the other way round). You find equivalents for the before mentioned management elements in this protocol,

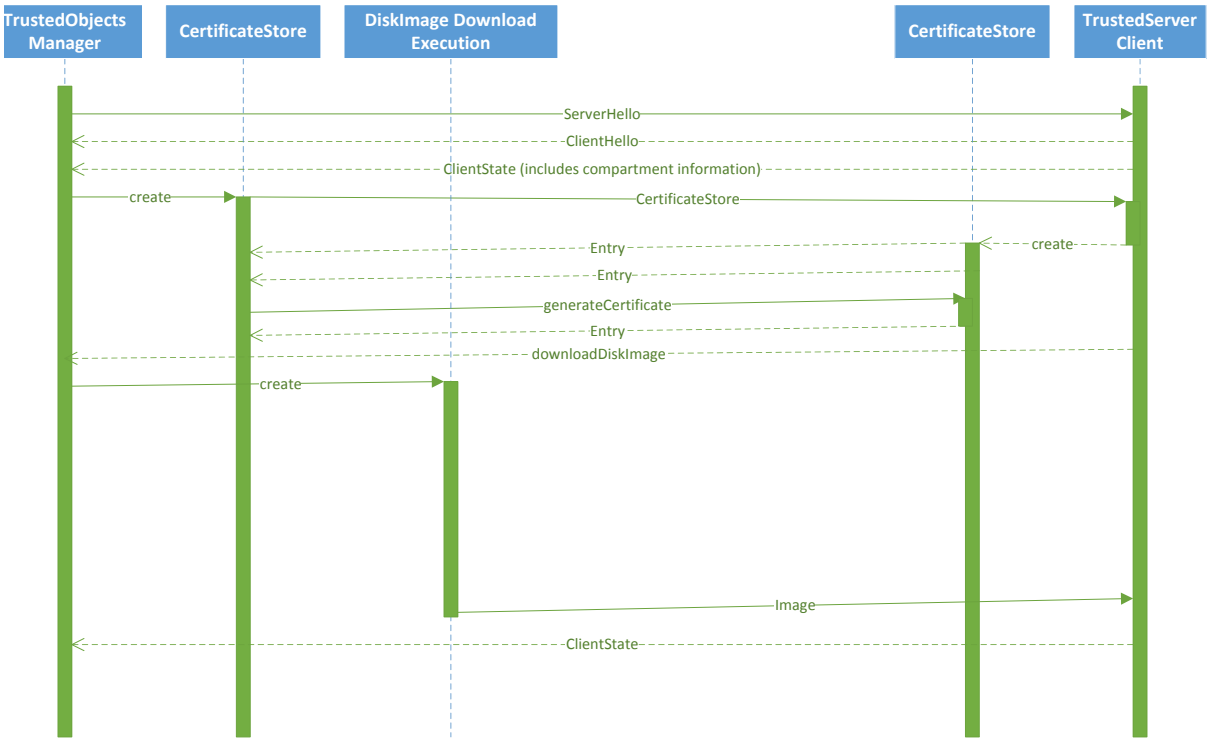

Figure 3.12: Management Protocol Execution Example

like security policies, which combine TVDs and Information Flows and download jobs for disk images. The remote attestation is outside the scope of this layer. The sequence diagram in Figure 3.12) illustrates the execution of a portion of that protocol. The entities "CertificateStore" and "Diskimage Download Execution" on the TOM side and "CertificateStore" on the TrustedServer side refer to internal jobs which handle the respective portions of the protocol.

Chapter 4

Adaptation of Configurable Trust Anchors for Large-scale Infrastructures

Chapter Authors:
Mihai Bucicoiu (TUDA)

4.1 Introduction

The cloud consists of several key concepts, like virtualization, distributed storage, management and so on and so forth. While proofing one node in the cloud in terms of trust is a very challenging task, trusting the entire infrastructure is even harder to achieve. In this report we shall present our vision to expand Cryptography-as-a-Service, a solution of single node trust, in order to create a trusted infrastructure. We will look at the challenges that rise when building such an infrastructure and discuss some solutions to it. Moreover, we shall present the partial setup of CaaS on the AWS EC2 cloud.

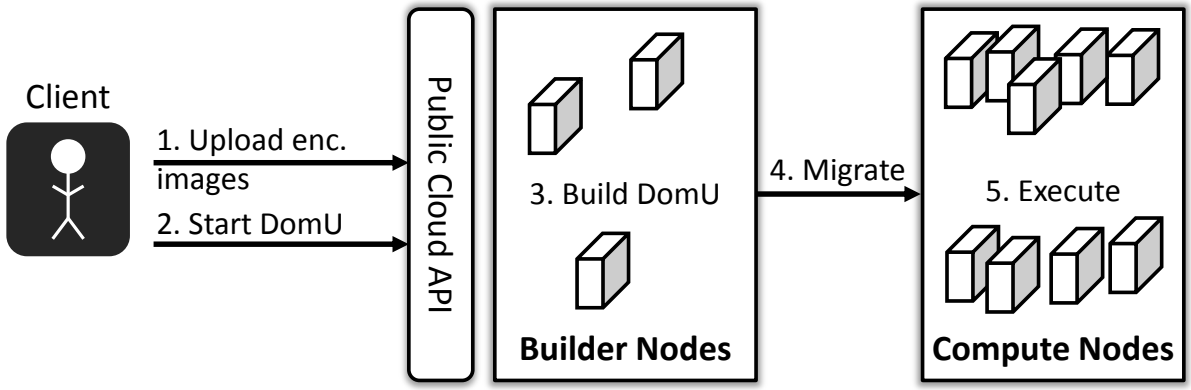

Figure 4.1: Scaling Trusted Computing for infrastructure clouds

Cloud computing offers IT resources like storage, networking and computing platforms on an on-demand and pay-as-you-go basis, which makes them very appealing to clients. The problem with such services is that the customer must trust the administrator of the cloud with his personal (and private) data. In our Cryptography as a Service (CaaS), we specifically aim at providing client-controlled trust for a node. We want to expand CaaS to the cloud infrastructure with multiple nodes, allowing the client to use a simple and compact interface (Figure 4.1)

only validating the cloud once and trusting it when her VMs are managed (e.g., migrated) by the cloud administrator.

4.2 Scalability challenges and possible solutions

The CaaS design relies on the fact that the client is based on the trustworthiness of the cloud node (i.e., our extended hypervisor) on which she deploys her virtual machines (VMS). This trust is established using TPM (Trusted Platform Module) sealing functionality to the trusted state S. However, the standard TCG (Trusted Computing Group) concepts have crucial drawbacks when applied in the context of *real-life* cloud computing, i.e., large-scale infrastructures:

1. The platform state S is reported using binary attestation, hence identified as a cryptographic hash (SHA1) of the software components comprising the state. Thus, minimal changes to the software stack (e.g., a security update) result in a different hash. Consequently, the client has to be aware of all possible trustworthy hash values. Moreover, determining the trustworthiness of a hash value requires knowledge of the code base the hash is derived from and thus the client gains full insight in the cloud provider's infrastructure. This is, on one hand, infeasible for the provider, since his code base is his trade secret, and on the other hand unnecessary complexity for the client, who is rather interested in the software stack fulfilling certain security properties (e.g., protection of her secrets) instead of having full insight.

2. Leveraging a *certified* TPM binding key always requires at one point in the certification chain a TPM-dependent, i.e., platform-dependent, certification key[1] (independently from being a migratable or non-migratable key). Hence, the client must be able to verify that she is communicating with a genuine TPM, requiring a certification of the TPM's credentials by a certification authority, denoted in TCG terminology as *privacy CA*. However, cloud infrastructures can include hundreds of thousands of nodes where the client's VMs can be deployed, requiring the client to verify hundreds of thousands of platforms.

3. Successfully verifying the attestation of a platform, only informs the client about the trustworthiness of the platform's software stack, but does not provide important meta-information such as the physical location of the platform. In our adversary model, the client has to be assured that the attested platform is on the premises of the cloud provider, thus under physical control of a trusted staff of hardware administrators. Otherwise, a *logical* attacker can trick the client into deploying her private information onto a platform running a trusted software stack, but deployed outside this trusted perimeter, which is easily prone to physical attacks by outsiders.

While providing solutions to the above mentioned scalability, challenges of Trusted Computing are out of scope of this report. Nevertheless, we do want to briefly mention here our approaches towards resolving these issues. Problem 1 is a long standing problem of TCG proposed Trusted Computing and a possible solution is *property-based attestation* (see, e.g., [SSW08]), which abstracts binary measurements of software to their desired security properties.

A possible solution for Problems 2 and 3 would be to establish the cloud provider as trusted Certification Authority, which provides the required certification for all its platforms within the trusted perimeter. To conclude, if the provider certifies a platform, the client is assured

[1] This key is of type *Attestation Identity Key* (AIK) [tpm].

that she is not tricked into revealing her secrets to an outside attacker. Considering the large number of platforms in a cloud (and its strong variances due to maintenance) and the consequent enormous complexity of managing this list, this approach is in practice infeasible. In CaaS, we conceptually solve this problem more efficiently, leveraging transitive trust (cf. migration in Section 4.3 and [SMV+10][SGR09]). The client initially instantiates her VM images only on a node belonging to a fixed subset of the cloud nodes (*"builder nodes"*), which can be publicly attested by the client. From these nodes, the instantiated VM is securely migrated to *"computation nodes"* for actual execution.

4.3 Fulfilling cloud requirements

In this section we will look into more details of CaaS design and if it can be scaled to large-infrastructures as cloud environments provide.

4.3.1 Design

Figure 4.2 illustrates the CaaS architecture in the default Infrastructure as a Service cloud model. For the sake of brevity, we only discuss each component role, while the architecture of the CaaS is detailed in D.2.1.2. The main goal of CaaS is to provide confidentiality for the customer VMs, called DomUs in Xen terminology, from the cloud provider administrator. It achieves this by two separate means: 1) removing all memory management from the management domain (e.g., Dom0 for Xen) into a new domain, called DomT, that can be verified using a TPM, and 2) adding a new encryption layer, called DomC, used for accessing the encrypted HDD of the virtual machine. By these means, the administrator cannot access the HDD of the VM, which is stored encrypted, and cannot read the memory of the running VM. For storing the key of the encrypted HDD, CaaS makes use of the hardware, namely the TPM, and binds this key to it.

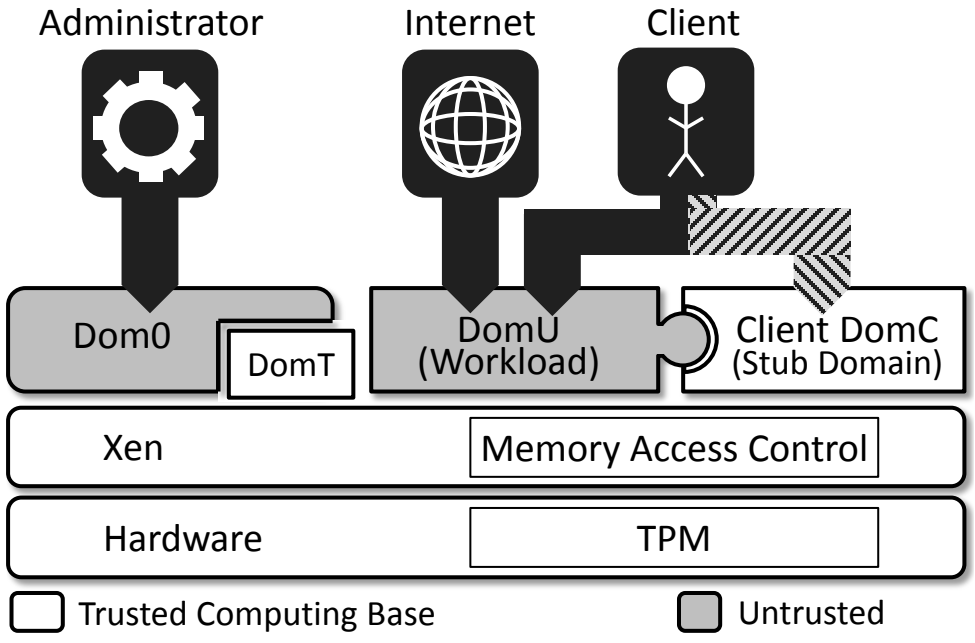

Figure 4.2: CaaS Design: Establishment of a separate, coupled security-domain, denoted as DomC, for critical cryptographic operations.

CaaS can be viewed as a "Local Service", described in Section 2.1.2.1 of D.2.3.2, that provides encryption to individual VMs using the host's TPM. As CaaS only impacts virtual machines operations (e.g., creation or migration), we will only consider these in the remaining of this section.

4.3.2 Analysis for transfer to large scale scenarios

Section 2.1.4 of D2.3.2 defines two important requirements that need to be fulfilled by any algorithm in order to be capable of adaptation for a cloud environment: scalability and elasticity.

First, let's have a look at DomT component. A cloud consists in a number of hosts, on each running a hypervisor responsible for VM management. Each hypervisor needs to be expanded by adding the DomT functionality for Dom0 and the required access control for memory management. As this is done independently for each hypervisor, this operation can scale at the magnitude of cloud and it has no other requirements for the cloud infrastructure. This technique, called *domain disaggregation*, is already used by cloud provider to brake Dom0 into several privileged service domains.

If we assume that all the hypervisors in the cloud have been extended with DomT, adding DomC to each VM require no further modification of the hypervisor. Because this function is atomic, i.e., each VM has his own DomC, scaling to every VM is only a matter of resources availability. The elasticity of the algorithm also holds due to the one to one relationship to each VM, not requiring a fixed number of resources available. If parts of the cloud have not been extended with DomT, they can still be used for starting "vanilla" VMs, but will not benefit from the encryption provided by CaaS. Moreover, migrating encrypted VM to such a host is not possible.

4.3.3 Cloud provider requirements

There are three major requirements, as depicted in Section 2.1.3 of D2.3.2, that a cloud provider will ask before deploying any new system: impact on the current cloud infrastructure, demand for the new system, and market penetration.

The first and most important aspect is the impact of the new system. CaaS is designed to improve the security of the cloud by limiting the access to the customer VMs from the cloud administrator, adding to the overall security of the cloud. In terms of resources required, CaaS adds only 6.5MB for DomT (both on local disks and memory) and 1MB for each running VM. As shown in Section 3.3.3, D.2.1.2, the impact on disk access performance is only 3.2%, with no additional impact on the networking or other I/O operations. In terms of reliability, CaaS does not change the cloud as a whole.

Our system was created based on a major demand and to solve the main argument when talking about migrating to cloud services, i.e., the customer needs to trust the cloud provider with its data. Moreover, if we look at sensitive information, like that saved by hospitals for their patients, there are several laws that prohibit the release of such information to outsiders, making the power of cloud computing unusable. In D1.3.3 we go into the details of how CaaS can be used to enable hospitals to make use of the power provided by cloud computing.

In terms of market penetration, CaaS can be provided as a separate service to only those that need extra protection of their data. Being the first product that can offer isolation from the cloud provider itself, we believe that it can be of great benefit to those that adopt it.

4.4 Real-world implementation design

Cloud clients of existing public infrastructure clouds can already use our security architecture to protect their high-value credentials against end-users compromising their workload VMs.

We describe how our implementation can be adapted for the Amazon EC2 cloud, which we chose due to its usage of Xen and its popularity. The first step would be to extend the Xen hypervisor with CaaS, namely to add the DomT and remove unnecessary management access from Dom0. This can be done through a normal update routine and should be done only once for each host. We now look at how daily VM operation can be done in order to maintain the security of the VM and his attached domC.

Suspension and live migration are fairly similar in the sense that suspension saves the current execution state of a VM to storage and can restore it at any time later, whereas live migration transfers the execution state from one physical machine to another while minimizing the down-time. We only cover migration here, as this technically implies suspension.

In order to support live migration, the usual migration protocol ([CFH+05, SCP+02]) needs to be wrapped, but in essence it works unaffected by client and DomU input. Instead of migrating plaintext VM memory from one host to another, the memory must be encrypted, since migration requires the involvement of Dom0 in order to distribute the saved state to the target platform. Thus, before granting Dom0 the required access to DomU's memory, DomT encrypts this memory in-place. Then, DomU's state is transferred to the target node using protocols from traditional live migration.

To restore the transferred state on the target platform, DomC has to be migrated as well to decrypt the migrated DomU state on the target platform. Restoring a VM state requires platform-dependent modifications to the state, such as rebuilding the memory page-tables. DomT's domain building code performs these modifications on DomU during DomU's resumption. While it would be possible to delegate this task to our trusted hypervisor or a central trusted domain building VM ([MMH08, BLCSG12]), this has the drawback of bloating the Xen hypervisor code base and introducing unnecessary complexity at that level or removing the client's full control over the DomU domain building process. However, before DomC can perform this task, its own state has to be restored.

In our design we opted for DomC *program* state migration instead of VM state migration. We therefore only transfer the state of the cryptographic programs in DomC and do not migrate the DomC VM state (e.g., CPU state). To migrate the program state, the target platform instantiates a new DomC and updates its state with the DomC state of the source platform. Afterwards, the new DomC is able to decrypt and resume the DomU state on the target platform and the old DomC on the source platform can be discarded. To achieve the protection of the transferred DomC state, this state is encrypted under the public TPM key, pk_{TPM}, of the system where the domain is transfered to. This can only be decrypted by the TPM itself with the private-key. Thus, only a target node running our trustworthy hypervisor is able to decrypt and resume the DomC state. All cloud nodes running our hypervisor form a *trusted network*, in which the client transitively trusts all nodes to securely distribute her DomU and DomC, after she has successfully verified the node on which she instantiated her DomU.

In case of suspension, the protocol works identical, except that the "target platform" is cloud storage to which the protected DomC and DomU states are saved by Dom0.

Chapter 5

Secure VM Containers

Chapter Authors:
Roberto Sassu (POL)

5.1 Overview

"Secure VM Containers" is a new extension that aims to enhance the security properties offered to virtual machines by the bare OpenStack. Since the Folsom release, Quantum, a core component of OpenStack, allows tenants to configure logically isolated virtual networks which connect the network interfaces of virtual machines. Quantum can accomplish this task by using different technologies, like VLAN, GRE and OpenFlow, through a plugin mechanism that allows developers to provide different implementations of the same abstract API.

Our new extension does more than providing isolation to tenants' virtual networks as it applies the Trusted Virtual Domain (TVD) concept to the Cloud. The TVD model was defined by Bussani et al. [BGJ+05] in order to address a serious concern that is present in distributed environments: the difficulty to apply a security policy uniformly across all the platforms that compose a distributed environment. The difficulty resides in the fact that the configuration task is error prone and that the necessity of managing several platforms will often cause inconsistencies that lead to a violation of the security policy requested by a user.

The TVD model defines a container of Execution Environments or EE (which in practice are equivalent to virtual machines) on which the following security properties must be guaranteed:

- **Isolation**: TVD members can communicate only among themselves regardless of the network topology that connects EEs;
- **Confidentiality/integrity**: Communications among TVD members cannot be intercepted or modified by unauthorized entities;
- **Trust**: An EE can join a TVD only if the host which it is running on satisfies the integrity properties specified in the TVD security policy.

5.2 Ontology-based Reasoner/Enforcer

Ontology-based Reasoner/Enforcer is a new subsystem developed by POL and comprised of the Reasoner component (not delivered for the TClouds project), which is responsible to check

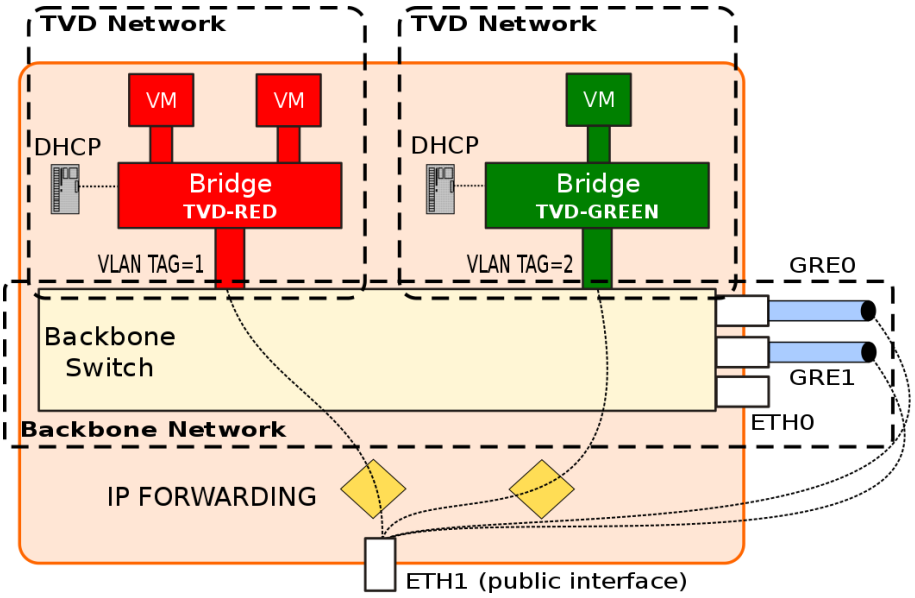

Figure 5.1: Extended Libvirt virtual networks

whether the network configuration matches the desired security goals, and the Enforcer component, which configures the virtualization software according to the desired security goals[1].

The Enforcer partially implements the TVD model by guaranteeing the first three security properties on the containers. As already mentioned in Section 3.4.2 of the D2.4.2 deliverable [Rob12], this subsystem is composed by two parts:

- **Extended Libvirt**: enhanced version of Libvirt that allows to describe TVDs by using the XML language;
- **Open vSwitch**: virtual switch that ensures network isolation through VLANs and GRE.

With Extended Libvirt, it is possible to configure the virtualization components to enforce TVD security properties, as depicted in Figure 5.1. In this figure, two different types of virtual networks have been represented:

- **Backbone Network**: This network consists of a virtual switch (Backbone Switch) that grants connectivity to VMs running on remote hosts through GRE tunnels or physical network interfaces. As part of the definition of this network, it will be possible to include the parameters of a pair of IPSec security associations to protect the confidentiality and integrity of data exchanged between hosts but this feature currently is not implemented;
- **TVD network**: This network contains a bridge that connects all VMs within a TVD and is connected to the Backbone Switch through an access port with the VLAN TAG associated to the TVD. Optionally, it offers to VMs a connection to the outside networks e.g. through NAT.

As mentioned above, it is possible to create these virtual networks by generating appropriate XML configuration files. Extended Libvirt will translate these files into commands to the hypervisor, Open vSwitch and iptables (to configure the firewall).

[1]For the TClouds project, POL is releasing only the latter, as the Remote Attestation Service was delivered in place of the Ontology-based reasoner component.

```
<network>
  <name>net-backbone</name>
  <bridge name='br-backbone' type='openvswitch' />
  <forward mode="none" />
  <tunnel>
      <remoteip address='10.0.0.2'/>
      <device name='gre-1'/>
  </tunnel>
  <tunnel>
      <remoteip address='10.0.0.3'/>
      <device name='gre-2'/>
  </tunnel>
</network>
```

Listing 5.1: Sample XML configuration file for a Backbone Network

```
<network>
  <name>net-tvd-daefbdfb-0fb3-4cd8-b677-8b4a5575f3af</name>
  <bridge name='br-tvd-daefb' type='openvswitch' ↩
     sourcebridge='br-backbone' />
  <portgroup name='pgroup-tvd' default='yes'>
    <vlan>
      <tag id='1'/>
    </vlan>
    <virtualport type='openvswitch'/>
  </portgroup>
</network>
```

Listing 5.2: Sample XML configuration file for a TVD Network

The Listings 5.1 and 5.2 show respectively a sample configuration file of a Backbone Network and a TVD network. From the former, Extended Libvirt creates a virtual network with a Backbone Switch named `br-backbone` and two GRE tunnel endpoints to the hosts with IPs 10.0.0.2 and 10.0.0.3. Instead, from the latter, it creates a virtual network with a bridge named `br-tvd-daefb`, connected to `br-backbone` through a VLAN access port with TAG set to 1.

5.3 Integration into OpenStack Quantum

While using the Enforcer alone is sufficient to ensure the TVD security properties on virtual machines, it must work together with Quantum in order to gain the same advantages in Cloud infrastructures based on OpenStack. To achieve this goal, the plugin mechanism has been leveraged to convert commands sent by a client utility or other OpenStack services (like Nova) into calls to the TClouds subsystem, i.e. the Ontology-based Reasoner/Enforcer.

As it can be seen in Figure 5.2, the Controller node runs the Quantum server, which has been configured to use the Open vSwitch plugin. The latter interacts with agents running on the Clouds' Compute Nodes responsible to deploy users' virtual machines. Since the goal of the *Secure VM Containers* extension is to configure Compute Nodes as depicted in Figure 5.1, the work done by POL consisted in replacing the agent developed as part of the Open vSwitch plugin with a new agent, called Libvirt Agent, capable of generating the TVD XML configuration and sending it to Extended Libvirt. Then, the latter will send commands to Open vSwitch as it would happen in the original version of the agent.

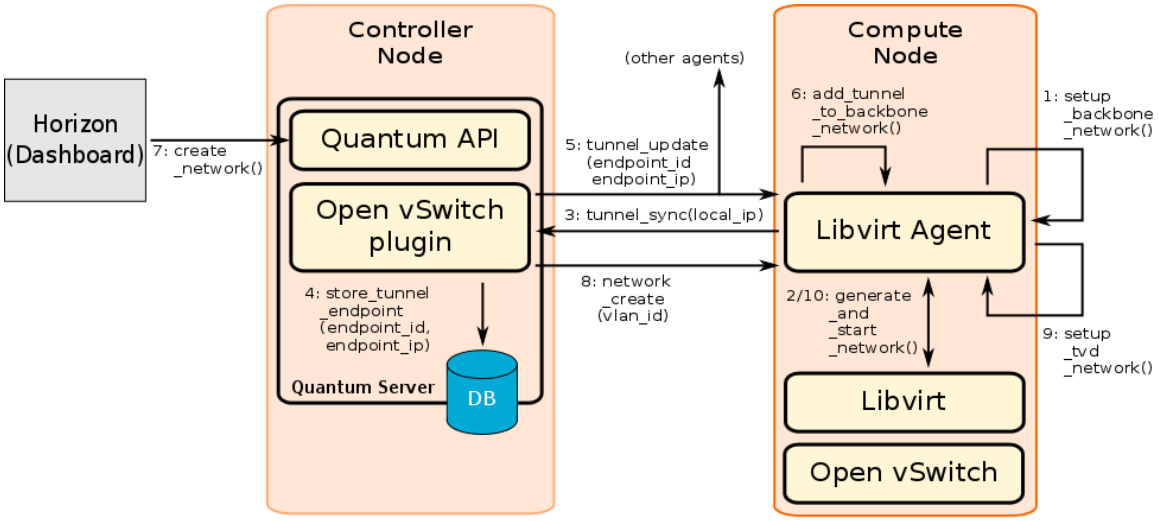

Figure 5.2: Integration into OpenStack Quantum

While this choice seems redundant, as the new security extension introduces a new layer to perform similar tasks, the TClouds solution has the following benefits:

- **Interoperability with other Cloud software stacks (e.g. Open Nebula)**: Since the TVD security properties are enforced by generating the appropriate XML configuration and sending it to Extended Libvirt, it is easy to integrate the TClouds extension into other Cloud management software, as they are usually based on Libvirt;
- **Easy discovery of the network state**: Since the network state can be derived by dumping the XML configuration from Extended Libvirt, third party software (e.g. SAVE subsystem developed by IBM in TClouds [BGSE11] and Chapter 8 of the D2.4.2 [Rob12] deliverable) can easily check for security problems in the infrastructure;
- **Enforce the confidentiality/integrity TVD security properties**: In future versions, it will be possible to use the XML configuration to set the parameters of IPSec security associations. This allows to protect the confidentiality and integrity of the communications between Compute Nodes;

Referring to the Figure 5.2, the interaction between Quantum and Ontology-based Reasoner/Enforcer can be split in two phases:

- **Setup phase**: During this phase, the Libvirt Agent is started on Compute Nodes and configures the Backbone Network, without tunnel endpoints, (steps 1 and 2). Then it notifies its presence to the Open vSwitch plugin (step 3), which stores this information in a database (step 4). Once the Libvirt Agents in other nodes are running, the local Libvirt Agent receives a notification of their IP address from the plugin (step 5) and updates the Backbone Network configuration by creating a new tunnel endpoint for each remote agent (step 6);
- **Tenant network creation phase**: In this phase, a tenant creates a new virtual network, e.g. from the Dashboard. The latter invokes the `create_network()` API function of the Quantum Server (step 7), which in turn sends a notification of this event to all Libvirt Agents (step 8). Finally, the agents configure a new TVD network with the VLAN TAG received by the plugin (steps 9 and 10).

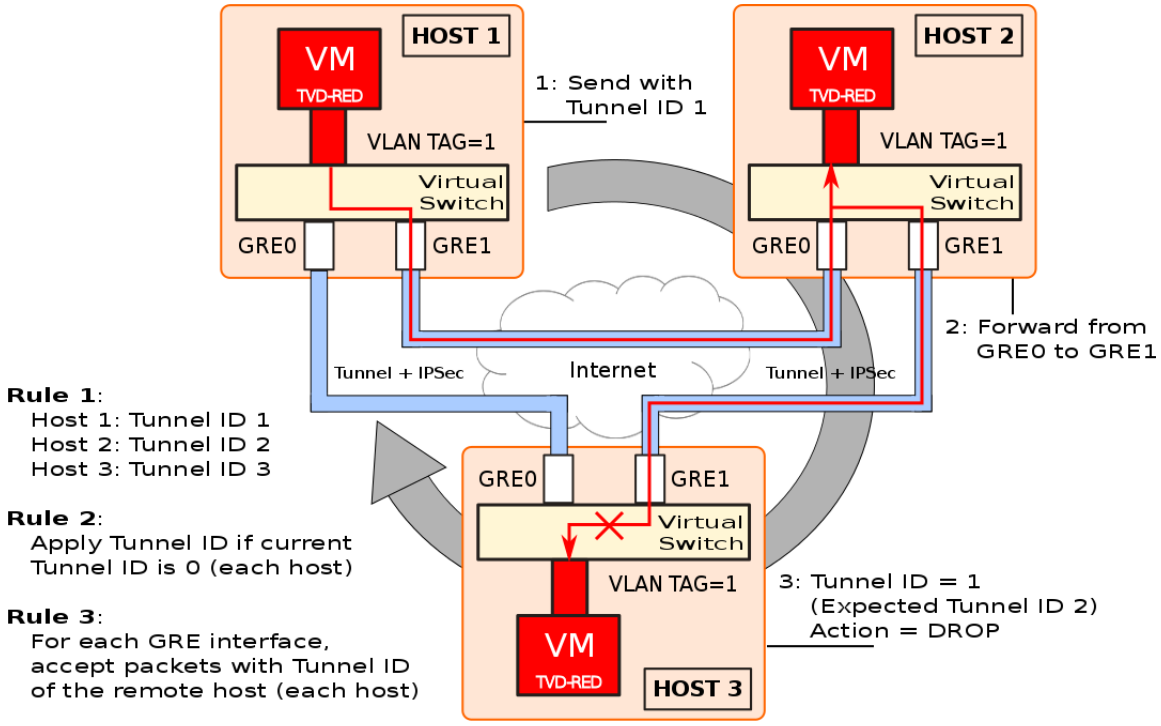

Figure 5.3: Loop prevention

From this description, it can be inferred that each host (i.e. a Compute Node) is connected through a tunnel with each other host (Compute Node) in order to allow virtual machines of a virtual network (i.e. a TVD) talk to each other even if they are spread in different hosts. This results in the quadratic growth of tunnel endpoints on each host as the number of the hosts present in a cloud zone increases: $n-1$ tunnel endpoints on each host for n hosts. The ensemble of the physical hosts and the tunnels between them form a fully connected mesh network, which is the basis for an overlay mesh network for connecting the related VMs within each TVD.

The high number of tunnel interfaces set up on a single instance does not cause any problem as Open vSwitch has been properly designed to scale. Instead, the resulting topology (a sample scenario is depicted in Figure 5.3) may cause network problems due to the creation of loops between hosts. This issue has been solved by developers of the Open vSwitch Quantum plugin by defining, for the virtual switch that contains tunnels to other hosts, a set of flow rules that allows the communication only between virtual machines network interfaces and tunnel endpoints and vice versa. However, this solution does not apply to the *Secure VM Containers* extension, as the network configuration defined for each Compute Node is slightly different: in the TClouds solution, only one virtual switch is used to connect virtual machines network interfaces with tunnel endpoints, instead of two.

For this reason, a different solution has been identified to avoid connection loops when using the TClouds extension. It consists in avoiding that a packet sent by a virtual machine traverses more than one tunnel segment through the definition of three rules that must be applied by each host of the Cloud infrastructure. First, each tunnel endpoint is associated to a tunnel ID (a 32 bit identifier defined in the header of GRE packets). Second, a tunnel endpoint labels outgoing packets only if packets coming from another virtual switch port have tunnel ID set to zero[2]. Finally, a tunnel endpoint accepts incoming packets from a tunnel endpoint only if their

[2]This condition is true only for packets sent by virtual machines.

embedded tunnel ID is equal to the one associated to the remote endpoint. This association is done by Quantum during the setup phase.

It becomes clear from Figure 5.3 why this set of rules allows to achieve the stated goal. Consider, for example, the situation where a packet sent by a virtual machine running on `HOST 1` in the Red TVD traverses more than one tunnel segment and reaches `HOST 3` through `HOST 2`. The target host discards that packet because its tunnel ID differs from the one expected (tunnel ID of `HOST 2`).

On the other hand, the packet sent by the VM running on `HOST 1` reaches directly `HOST 3` through the tunnel segment between the two hosts: this flow is not represented in Figure 5.3.

In conclusion, although the Open vSwitch Quantum plugin realizes a network topology which likely contains loops, our agent implementation (the Libvirt Agent) ensures, as the same as the original Open vSwitch Agent, that virtual machines communicate among themselves properly, by allowing packets sent by a host to be transmitted only to the adjacent ones.

Chapter 6

A Framework for Establishing Trust in Cloud Provenance

Chapter Authors:
Imad M. Abbadi (OXFD)

6.1 Introduction

Cloud computing is relatively a new term in mainstream IT, first popularized in 2006 by Amazon's EC2[Ama10]. It has emerged from commercial requirements and applications [JNL]. Establishing trust in cloud architectures is an important subject that is yet to receive adequate attention from both academia and industry [JNL, Abb11c, AFG+, Mic09]. Logging, auditing and historical data are of tremendous importance for establishing trust in clouds. This data has different usage, e.g. pro-active service delivery (incidents and security monitoring), billing, error and forensic investigation. For convenience in this chapter we refer to this data as log records. Almost all of clouds' resources generate this data in some way. The importance of such data and its usage is based on the following resource types. *Physical resources* generate log records related to physical resource status, security and incident reporting. The generated data helps in the direction of finding the cause of incidents and for security monitoring. *Virtual resources* generate log records related to virtual resource status, security and incident reporting. They also generate usage data, which are used for billing customers using IaaS clouds. Finally, *Application resources* generate log records related to application resource status, security and incident reporting. They also generate usage data that are used for billing customers using PaaS and SaaS clouds.

Establishing trust in cloud systems (as we discussed in [Abb11b]) requires two mutually dependent elements: (a) support infrastructures with trustworthy mechanisms and tools to help cloud providers automate the process of managing, maintaining, and securing their systems (what we referred to as self-managed services [Abb11a]); and (b) develop methods to help cloud users and providers establish trust in the operational management of the infrastructure. Our previous work ([AAM11]) focusses on both points (a and b); specifically, it establishes offline chains of trust across the distributed elements of clouds physical infrastructure helping self-managed services to securely exchange management data, and it provides a mechanism enabling users to attest to the way clouds infrastructure is managed. The framework presented in this chapter focusses on point (a) by supporting self-managed services with a trustworthy provenance system. This chapter, in addition, extends our previous framework's entities to be provenance aware; i.e. establish offline chains of trust between cloud entities and the prove-

nance system, collect log records from the distributed elements of clouds infrastructure, associate important identification metadata with such records in clouds context, and securely push the result to the proposed provenance system. The integrated framework helps in our long vision of establishing trustworthy clouds.

6.1.1 Log and Provenance

Logs and *provenance data* are distinctly different. Logs provide a sequential history of actions usually relating to a particular process. Provenance generally refers to information that 'helps determine the derivation history of a data product, starting from its original sources' [SPG05]. Provenance goes beyond an individual application or a process and may refer to many pieces of equipment as well as people. Throughout this chapter we refer to logs as being a source of provenance, primarily because in cloud logs are used in combination for a similar purpose.

The provenance is provided on clouds through linking together log records, collected from multiple resources, to provide the complete history of an event or result. cloud provenance, at present, is associated with the following limitations [AL11]: the methods followed by clouds to support provenance queries are basic and, in many cases, such methods are developed on an ad-hoc basis by cloud system administrators using customized scripts to address a specific event. In addition, current provenance mechanisms are object specific; i.e. they do not automate the process of managing different log and audit files and linking dependent log and audit records together. Current log and audit records are not reasonably protected, which in turn affects the creditability of provenance in the cloud. Moreover, current cloud provenance mechanisms are deployed and fully controlled by cloud providers; i.e. cloud users do not have control over such mechanisms, and neither can they access log and audit records.

The identified limitations motivate the need to establish a trustworthy secure cloud provenance, which we next discuss the complexities exposed in this.

6.1.2 Problem Description and Objectives

We believe that establishing trustworthy secure cloud provenance requires great efforts from both academia and industry. One of the main reasons for the complexity of cloud provenance is that it uses log records which are associated with the following issues: i) log records are not properly managed and are dispersed amongst clouds complex and distributed infrastructure, e.g. most of log records are scattered all around the infrastructure using unstructured and unrelated text files; and ii) log records do not adhere to any standard format (this covers both the ones generated by different processes and the ones generated by similar processes but from different manufacturers). Also, such log records do not have semantics explaining the meaning of the items of log records.

Provenance in clouds with the above problems is not practical considering clouds enormous number of applications, complex infrastructure, and huge number of users. In addition, cloud provenance is even much more complicated than traditional enterprises considering cloud dynamic nature [Abb11a, AN11]. The dynamic nature of clouds results in its desired properties, e.g. resource consolidation, resilience, scalability and high availability; however, this dynamism results in new challenges, e.g. building a logical sequence of events to investigate an incident for any one application requires data from many sources, which include the application itself, all logs for possible virtual resources that the application could have used, and logs of all physical resources that virtual resources could have used. Administrators must then combine this data correctly by identifying all time intervals when an application used a specific virtual resource,

all possible time intervals when these virtual resources used physical resources and then all relevant log files from all related resources. Collecting and combining data from these resources is not easy or practical considering the potential scale of cloud systems. These, in turn, increase insider threats in clouds and reduce its trustworthiness, which discourage critical infrastructures to outsource their resources to public clouds.

We believe that the foundation of providing cloud provenance requires following key elements: i) establish semantics and standards of log records which enable the automated understanding of log records as generated by multiple processes; ii) store log records in a structured, highly available, and centralized repository which enable provenance tools to easily and quickly find log records, query them, and bind related events together; iii) provide security measures for storing, querying, transferring, and managing log records; and iv) establish trust in the operation of the processes managing log records which help end users to establish trust in cloud provenance.

Providing trustworthy secure cloud provenance is a complex problem that requires lots of efforts. This chapter focuses on point (iii) above. In order to clarify the overall picture and to put the proposed scheme in context, the chapter also partially discusses points (ii and iv). In addition to points i, ii, and iv this chapter does not cover the details of the following: a) a detailed database management system design for supporting provenance application requirements, b) a detailed design of the provenance application itself, c) policy management and enforcement (e.g. log retention policy), d) detailed discussion about VM agents that manage provenance data inside a VM (we only outlined one aspect of this, i.e. secure storage and transfer of provenance data), e) protecting provenance data and domain credentials once decrypted in memory, and f) key management.

6.1.3 Organization of the chapter

The chapter is organized as follows. Section 6.2 discusses related work and our contribution. Section 6.3 introduces clouds structure and management services. Section 6.4 presents motivating scenarios. Section 6.5 discusses the management of provenance data and then extracts the system requirements. Section 6.6 defines our proposed domain architecture. Section 6.7 identifies the software services and their functions. Section 6.8 provides our framework workflow. Section 6.9 provides an informal threat analysis of the proposed workflow. Finally, we discuss and conclude the chapter in Section 6.10.

6.2 Related Work and Contribution

The need for additional provenance information in cloud computing storage has been well established by Muniswamy-Reddy et al. [MRMS10, MRMS09]. The authors have discussed the requirements for adding data provenance to cloud storage systems and have analysed several alternative implementations. This is in contrast to our work, which considers the entire cloud infrastructure and proposes a framework for that. The use of provenance for fault tolerance has also been proposed before for grid computing [CA08, Xu05]. One aim is to avoid common modes of failure when attempting to use multiple composite web services. This work provides useful motivation for the collection of provenance data, but the move to cloud computing requires a new analysis of current problems in the collection of provenance data.

There are many promising tools which could be adapted for use in cloud environments. Muniswamy-Reddy et al. [MRMS10, MRMS09] have already evaluated the use of PASS – the

Provenance-Aware Storage System – for cloud provenance. Reilly and Naughton [RN09] have proposed extending the Condor batch execution system to capture data on execution environments, machine identities, log files, file permissions and more. While there are significant new challenges on a cloud infrastructure, the Provenance-Aware Condor system certainly collects the right kind of provenance data.

The need to protect the security and privacy of applications' log records has been discussed in [HL09, HM08]. These papers primarily focus on protecting applications' log records generated by virtual machines which are hosted at a specific physical machine. The authors identify the importance of (but did not address) the distributed log management, as generated by distributed systems, and leave it as future work. clouds infrastructure is distributed and dynamic by nature which necessitates the need of distributed log management which covers both application and infrastructure management logs.

The authors could not find other related work which use logs as a source of provenance in cloud environment neither they could find related work covering log management in distributed systems. The idea of this research, in fact, is based on a real problem in clouds computing (inherited from enterprise infrastructure, the predecessor of cloud computing). The research problem does not have a practical solution at the time of writing[1]. Cloud providers currently relies on security administrators to manually (supported with basic management tools and in house scripts) perform the cloud provenance job. Trustworthy provenance is a key requirement for establishing trustworthy clouds' self-managed services that support our global vision of establishing trust in clouds. Our novel contribution in this chapter is about covering the foundations of this topic; i.e. identify the requirements of cloud provenance, propose a provenance framework, and address some of the identified requirements. In addition, the proposed provenance framework extends our previous work ([AAM11]), which is also part of our contribution to this chapter.

6.3 Cloud Structure and Management Services

Providing cloud provenance requires careful understanding of the cloud taxonomy and management services. This section briefly summarizes our previous work in this direction ([Abb11a, Abb11c]).

6.3.1 Cloud Structure

Cloud environment is composed of enormous *resources*, which are categorized based on their types and deployment across cloud infrastructure. Cloud environment conceptually consists of multiple intersecting layers as follows: i) *Physical Layer* — This layer represents the physical resources which constitute cloud physical infrastructure; ii) *Virtual Layer* — This layer represents the virtual resources which are hosted by the *Physical Layer*; and iii) *Application Layer* — This layer runs the applications of cloud's customer which are hosted using the *Virtual Layer*. We identify an entity *Layer* as the parent of the three Cloud 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

[1]The author has more than 15 years of industrial expectance covering most technologies behind today's cloud infrastructure.

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.

Cloud resources communicate in a well organised way, either horizontally and/or vertically ([Abb11c]). *Horizontal communication* is where cloud resources communicate as peers within a layer, domain, or group. *Vertical communication*, on the other hand, is where cloud resources communicate with other cloud resources in the same layer or another layer following a process workflow in either an up-down or down-up direction.

Cloud resources are dynamic which means the following: a) a specific virtual resource can be hosted at many different physical resources at different times according to a policy; b) similarly a specific application resource can run on multiple virtual resources that are increased or decreased based on load and a predefined policy controlling such behaviour (i.e. elasticity property [Abb11a]); and c) from (a and b) we can conclude that a specific application can be hosted under different physical servers.

6.3.2 Virtual Control Centre

This section 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 [VMw12] and OpenStack [Ope]. For convenience we call such tools using a common name Virtual Control Centre (VCC), which is a Cloud device[2] 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) [Abb11c]. 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 establishing communication channels with physical servers to manage Cloud's Virtual Machines (VMs). VCC establishes such channels by communicating with the virtual machine manager running at each server. Such management helps in maintaining the agreed service level agreement with customers.

Trust establishment requires automated self-managed services that can manage Cloud infrastructure (considering both user properties and infrastructure properties) with minimum human intervention [Abb11a]. VCC will play a major role in providing Cloud's automated self-managed services, which are mostly provided manually at the time of writing. In previous work ([Abb11a]) we focused on defining the functions of self-managed services. In this chapter we propose a provenance framework which is a key requirement for having automated and trustworthy self-managed services. Provenance helps self-managed services to reason about the changes across the distributed elements of Clouds; e.g. it helps such services to understand the consequences of a decision and to realize the right action plan to be considered.

[2]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.

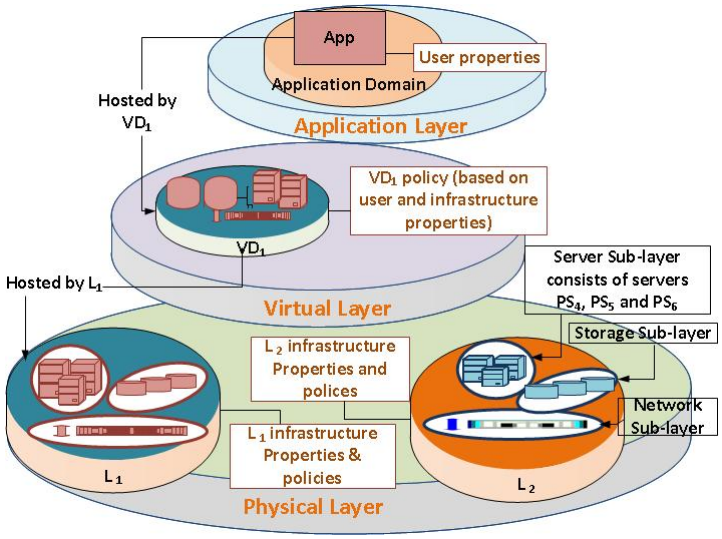

Figure 6.1: Provenance Scenario

6.4 Motivating Scenarios

We now discuss the importance of provenance in a Cloud using two simple example scenarios, as illustrated in Figure 6.1. We assume that a Cloud provider has six physical servers PS_1 to PS_6, and two physical domains L_1 and L_2. L_1 is allocated physical servers PS_1 to PS_3, and L_2 is allocated physical servers PS_4 to PS_6. We also assume that the Cloud provider hosts an application App. The Cloud provider creates a virtual domain VD_1 in the virtual layer to run App. VD_1 is initially allocated a one virtual resource, VR_1, to host App. VD_1 is associated with a policy allowing it to scale its resources when there is an increase in demand using resources from physical domain L_1.

Our first example demonstrates how a simple increase in load, and the corresponding reaction from the Cloud, can result in a loss of provenance data. Assume the load on App has dramatically increased, the following steps then apply: i) VD_1 responds by instantiating a new virtual resource VR_2 replicating VR_1 inside VD_1; ii) now both VR_1 and VR_2 process App, which are hosted using L_1 — assume that VR_1 is hosted by PS_1 and VR_2 is hosted by PS_2; iii) PS_2 has hardware problems, which results in incorrect results being generated by App; iv) load returns to normal and so VD_1 downscales by removing VR_2; and v) Cloud customers discover the problem and call the Cloud provider. If the Cloud provider only examines the logs of files generated by VR_1 and PS_1, then they will not find the root case of the problem or how to rectify it.

Our second scenario focus on forensic provenance in the Cloud, as follow: i) a security administrator reads the policy for VD_1 and understands that App can only be hosted using L_1 resource; ii) the administrator updates the VD_1 policy to force VD_1 to use L_2 resources; iii) the administrator then connects to L_2 physical resources and finds out that VD_1 resources are running on PS_4, meaning that App is hosted there. The security administrator connects to PS_4 and indirectly extracts important information from App. PS_4 logs this activity; and iv) The administrator restores the original policy, which forces VD_1 resources to switch back to L_1. If the Cloud provider only examines log files generated by L_1 resources, then they will not discover who performed the attack or, even worse, they might never discover that an attack has happened in the first place. This is one of the main challenges that shows the importance of

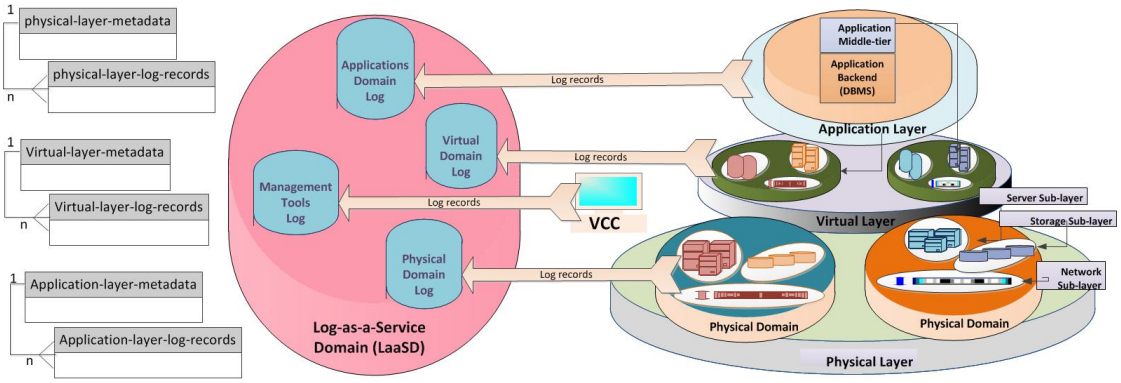

Figure 6.2: High Level Architecture of LaaS DBMS

provenance considering the complex Cloud infrastructure and enormous distributed resources.

6.5 Log Records Management and Requirements

In this section we outline a possible approach for managing log records (i.e. partially cover point (ii) discussed in Section 6.1.2). Following that we identified the system requirements.

6.5.1 Database Design

As explained in Section 6.3, Cloud computing is composed of enormous processes running at distributed and heterogeneous resources. Various processes communicate horizontally and/or vertically amongst each other. Log records generated by such processes require experts in the domain to interpret and establish relationships amongst them, especially they are stored in a scatted log files. Our design objective is to address these problems and fulfil the following:

- Move log records from their originating distributed processes to a centralized repository, as illustrated in Figure 6.2. By centralized we do not mean a single storage neither we mean any restrictions on geographical locations. Specifically, we mean moving disperse log records to a centralized provenance system, which we also refer to as Log as a Service (LaaS). The LaaS should be protected against single points of failures, e.g. replicated across different geographical locations such that each replica is supported by high availability infrastructure.

- The log records should be easily queried using standard mechanisms; e.g. ANSI SQL-92 [Dig92].

- Associate individual log records with metadata. The metadata associates items of log records with labels which explain the original source of log records based on the outlined Cloud taxonomy. The metadata also establishes the relationship between different log records in the Cloud. These help tracking log records considering both vertical and horizontal communication channels amongst Cloud components, and, also, considering Cloud dynamic nature.

Figure 6.2 illustrates a simplified schema design of LaaS database whereby we transfer and store log records to a centralized LaaS repository. LaaS repository, for example, could be com-

posed of a web-based log application supported by an appropriate distributed database management system (DBMS) (e.g. Oracle RAC and DataGaurd [Ora11b]). We propose categorizing log records into four parts, each is stored in a dedicated set of tables. Three categories cover the three horizontal layers of Cloud taxonomy, while the last category for management tools' log. Each category is composed of two types of provenance data: i) log records and ii) a metadata describing the details behind each item of log records in the context of the discussed Cloud taxonomy. The transfer of the provenance data to the LaaS repository and the management of the LaaS repository itself are controlled by trustworthy services which we cover in Section 6.7.

The LaaS DBMS is expected to be highly transactional with enormous size, considering the huge number of elements of the Cloud. These properties require a careful distributed system design that maintains reliability, eliminates any single point of failure, and maintains overall high system performance. Such properties also necessitate a log retention policy controlling the lifetime of log records (a retention policy should consider the type of log records, user requirements, and other legislative measures). It is outside the scope of this chapter to discuss these important issues and the LaaS DBMS schema design in further details, and we are currently working on a chapter supported by a prototype covering such details.

6.5.2 Security Requirements

The previous section outlines a simplified design of Cloud's log records management. We now identify the security requirements to transfer and manage such data, as follows (these are related to our stated objectives in points (iii) and (iv) discussed in Section 6.1.2): i) provide assurance measures to the LaaS that the log records are generated and transferred from their source by trustworthy processes; ii) provide assurance to the LaaS that the metadata associated with each item of the log records is correct; iii) Provide assurance measures to the processes which generate the log records that the Clouds' management processes are trusted to provide the correct metadata, and, in addition, the LaaS is trusted to protect the log records and the associated metadata; and iv) provide assurance measures to interested parties (e.g. Cloud customers, auditors, and even Cloud providers) about the trustworthiness and reliability of the LaaS mechanism to protect the log records and associated metadata. Subsequent sections focus on these points, which cover our objectives discussed in Section 6.1.2.

6.5.3 Other Requirements and Device Properties

The above identified properties of the LaaS (i.e. enormous size and high transaction rates) and the identified requirements (e.g. highly available and reliable system with no single point of failure) necessitate careful design at the infrastructure and application levels, which we do not cover in this chapter. LaaS, as a result, is expected to fully utilize multiple and redundant physical servers. Our proposed scheme requires the LaaS application to be installed and managed at dedicated physical servers which are physically separate from the other Cloud resources. In addition, we require LaaS to be managed by a dedicated provenance security administrators who do not manage the Cloud infrastructure, as illustrated in Figure 6.3. This is to enforce the separation of duty principle and to not have same people who manage both the Cloud infrastructure and the LaaS.

We require that physical layer's 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 [Tru07]. Trusted computing systems are platforms whose state can be remotely tested, and which can be trusted to store

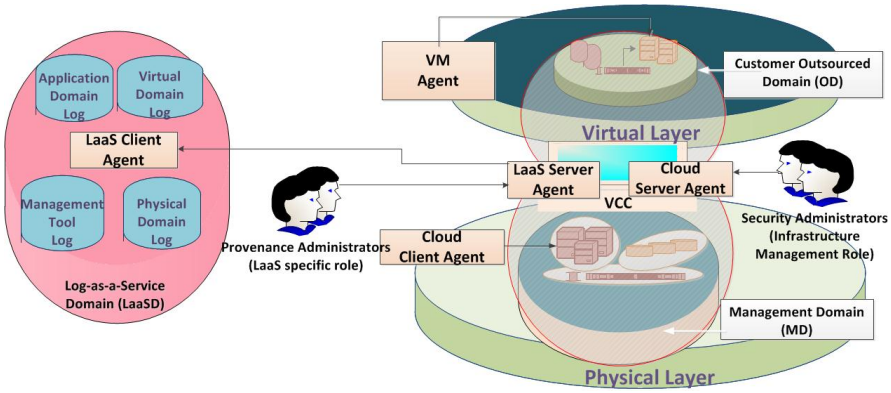

Figure 6.3: Domains and Software Agents in Clouds Taxonomy

security-sensitive data in ways testable by a remote party. TCG is a wide subject and has been discussed by many researchers; we will not address the details of TCG specifications in this chapter for space limitations (see, for example, [Sad08] for further details about this subject). The TCG specifications require each trusted platform (TP) to include an additional hardware component, the TPM, to establish trust in that platform [Sad08]. TPM has protected storage and protected capabilities. 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. Storage and retrieval are carried out by the TPM.

6.6 Framework Domain Architecture

In this section we propose a LaaS domain architecture which forms the foundation for addressing the identified objectives. The architecture uses the dynamic domain concept which is proposed in [AA08]. We start by defining the dynamic domain concept, and then discuss the adaptation of such concept to architect the framework.

6.6.1 Dynamic Domain Concept

Definition 6.6.1 ***A Dynamic Domain*** *represents a group of devices that need to securely 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. The dynamic domain protocols are discussed in detail in [AA08].*

6.6.2 Domain Architecture

The framework is composed of the following types of domains (see Figure 6.3): *Log as a Service Domain* (LaaSD), *Management Domain* (MD), *Collaborating Management Domain*

(CMD), *Outsourced Domain* (OD), and *Collaborating Outsourced Domain* (COD). We now map these domains using the Cloud infrastructure taxonomy concept, which we summarize in Section 6.3. An MD and CMD represent a Physical Domain and Collaborating Physical Domain at the Physical Layer. An OD and COD represent a Virtual Domain and Collaborating Virtual Domain at the Virtual Layer. LaaSD is composed of the LaaS-specific servers which hosts the LaaS system.

As we discussed earlier, the proposed framework extends some of the functions provided in our previous work ([AAM11]) to make them provenance aware. The previous work established a trustworthy and controlled environment for the management of MD/CMD when hosting OD/-COD. Subsequent sections identify the additional functions which we introduced at MD — to simplify the proposed scheme, this chapter does not cover the integration of the provenance system with OD, COD and CMD, as these will increase the complexity of the chapter and divert the focus.

Definition 6.6.2 ***LaaS Domain (LaaSD):*** *Consists of platforms that host Cloud LaaS application. Section 6.5, outlines the design requirements of LaaSD's hosting platforms. LaaSD has a unique identifier i_{laas}, two shared unique keys k_{laas} and $k_{laas-cca}$, and a specific PKL_{laas} composed of all devices in the LaaSD. k_{laas} is used to protect log records when transferred within LaaSD, while $k_{laas-cca}$ is used to protect log records when transferred from Cloud entities to LaaSD (specifically, between cloud client agent and log client agent as will be explained latter). The credentials of LaaSD are defined in Definitions 6.6.3, 6.6.4, 6.6.5 and 6.6.6. LaaSD is associated with a provenance policy, which controls LaaSD behaviour and manages the provenance data, e.g. data retention policy outlined in Section 6.5.*

Definition 6.6.3 ***LaaSD identifier*** *i_{laas} is a unique number that we use to identify LaaSD. It is securely generated and protected by the TPM of VCC.*

Definition 6.6.4 ***LaaSD key*** *k_{laas} is used to protect provenance data. k_{laas} is a symmetric key that is securely generated and protected by the TPM of VCC. k_{laas} is not available in the clear, it is shared between all devices member of LaaSD, and it can only be transferred from VCC to a device when it joins the LaaSD.*

Definition 6.6.5 ***LaaSD public key list (PKL_{laas})*** *is a LaaSD-specific list that is composed of the public keys of all devices of LaaSD. Provenance administrators assign devices to each LaaSD by providing each device public key to VCC in a form of PKL. The PKL_{laas} is securely protected and managed by VCC.*

Definition 6.6.6 ***LaaSD key*** *$k_{laas-cca}$ (also called LCA-CCA key) is used to protect the provenance data when transferred from Clouds' distributed elements to LaaS provenance application. $k_{laas-cca}$ is a symmetric key that is securely generated and protected by the TPM of VCC. $k_{laas-cca}$ is not available in the clear, it is shared between devices member of LaaSD and MD, and it can only be transferred from VCC to a device when the device joins MD or LaaSD.*

Definition 6.6.7 ***Management Domain:*** *MD represents a group of devices at the Physical Layer. The capabilities of devices member of MD and their interconnections reflect the overall properties of the MD. Such properties enable the MD to serve the part of user requirements which can be matched only at the physical layer (e.g. location restrictions, resilience and scalability properties). An MD has a policy which manages the behaviour of its members and controls the behaviour of collaborating MDs, and, in addition, the policy controls the transfer*

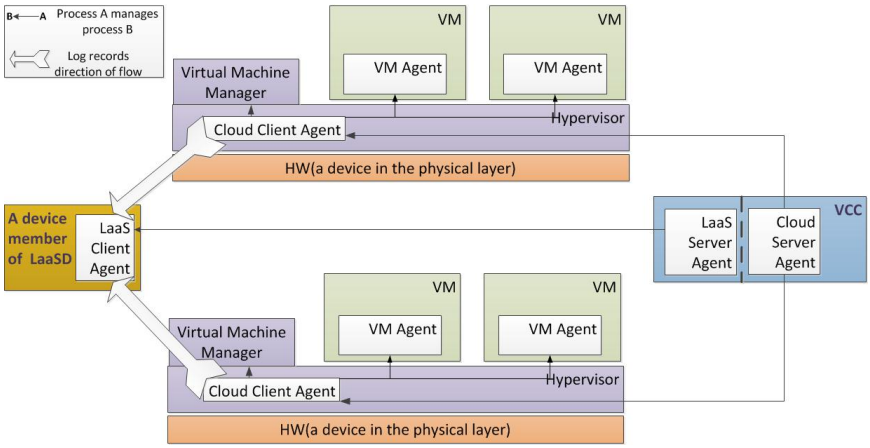

Figure 6.4: Software Agents for Cloud Provenance and Management Services

of log records and the association of metadata to LaaSD from across the distributed elements of Cloud infrastructure. The MD has credentials consisting of a unique identifier i_{md}, a unique symmetric key k_{md}, a public key list (PKL_{md}), and the shared LCA-CCA key provided by LaaSD, which have similar definitions to those provided in Definitions 6.6.3, 6.6.4, 6.6.5 and 6.6.6 respectively; however, i) MD is not managed by the provenance administrators, and ii) k_{md} is used to protect infrastructure management data.

6.7 Framework Software Agents

The proposed framework architecture is composed of a set of software agents which are required to implement the functions of the framework (see Figure 6.3 and 6.4). The software agents are as follows: i) Cloud client agent (CCA), ii) Cloud server agent (CSA), iii) LaaS server agent (LSA), iv) LaaS client agent (LCA), and v) virtual machine agent (VMA). In our previous work ([AAM11]) we provided the required protocols for CCA and CSA which control the management of OD/COD at MD/CMD. As we discussed earlier, the objectives of our previous work are not the same as the objectives of this chapter which necessitate introducing changes on CCA and CSA to provide an integrated framework. In the remaining part of this section we discuss in details the functions of these agents, and the changes we introduced at CCA and CSA to be provenance aware.

Assumption 6.7.1 *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. Achieving this is one of our long term objectives.*

6.7.1 Cloud Server Agent

CSA is a trusted management agent that runs at VCC, and has the following functions: a) install CCA on physical devices excluding the ones related to LaaS servers (covered in [AAM11]); b) manage MD/CMD policies and provenance policies; provenance policies provide assurance that log records are securely generated and transferred to authorized entities (MD/CMD policies are discussed in ([AAM11])). These policies also provide assurance that trustworthy metadata is generated and associated with log records; c) establish offline chains of trust between Cloud entities which include the following: i) CSA and CCA (covered in [AAM11]), and ii) collaborate with LSA to establish chains of trust between CSA and LSA, and CCA and LCA; and d) create and manage MD/CMD (the creation is covered in [AAM11], while the management lacks parts of points b and c, as discussed above).

6.7.2 LaaS Server Agent

LSA is a trusted provenance agent which runs at VCC. LSA has the following functions: a) install and manage LCA; b) manage provenance policies which provide assurance that provenance data are only accessible to authorized entities and control provenance data retention; c) establish offline chains of trust between provenance management agents (i.e. LSA and LCA), and between provenance management agents and other agents (i.e. LSA and CSA, and CCA and LCA); and d) create and manage LaaSD which includes the following: i) securely generating and storing LaaSD protection keys; ii) attesting to the execution environment status of devices' LCA whilst being added to the domain and ensuring they are trusted to execute as expected; hence trusted to securely store the domain key and to protect domain content; and iii) add and remove devices to a domain by releasing the domain-specific key to the LCA running on devices joining the LaaSD.

6.7.3 LaaS Client Agent

LCA is a trusted provenance agent which runs at physical platform member of the LaaSD. LCA has the following functions: a) intermediate the communication between CSA/CCA and the provenance system, and between provenance security administrators and the provenance system; b) assure verifiers that the provenance system operates in a trusted environment; i.e. can access provenance data when its execution environment is trusted; and c) manage and enforce organization policy related to the provenance operations as distributed by LSA.

6.7.4 VM Agent

The VM agent is a trusted agent running at all virtual machines which are organized into ODs and CODs. The VM agent intermediates the communication between running processes inside the virtual machine and CCA — this chapter only covers the secure storage of provenance data. The VM agent attests to the execution status of all running processes inside the VM and ensures that they are trusted to behave as expected. It then securely transfers the log records to CCA. The CCA, as explained next, is in charge of adding and binding the metadata to log records and then transferring the result to the LCA.

6.7.5 Cloud Client Agent

CCA is a trusted client-management agent running at resources of a physical layer (excluding the ones member of LaaSD). CCA has the following functions which are related to provenance system (these are additional functions to the ones discussed in our previous work, [AAM11]): a) enforce provenance policy as distributed by the LSA via the CSA; b) intermediate the communication between all processes running at the physical platform and the LCA. Specifically, it grabs the log records as forwarded from inside the VM and other processes in the hypervisor, and then associates them with the required metadata. Subsequently, it sends the result to its allocated LCA; and c) it sends its own log records (i.e. log records related to the management of virtual resources at physical resources) to its allocated LCA.

6.8 Framework Workflow

This section discusses a possible workflow of the proposed system framework. The chapter does not discuss OD/COD (i.e. it does not discuss VMs and the details of their agents), neither it discusses applications' provenance management. Discussing such details will drag us into extra complexities that diverts the focus of the chapter. At this early stage of our work we propose a set of protocols as a proof of concept with an informal security 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 would likely to be updated.

6.8.1 Cloud Server Agent Initialization

This section describes the procedure of initializing the CSA discussed in Section 6.7. Following are the notations used in this section: **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 [Tru07]: $\text{TPM}_{\text{C}reateWrapKey}$, $\text{TPM}_{\text{L}oadKey2}$, $\text{TPM}_{\text{S}eal}$, and $\text{TPM}_{\text{U}nseal}$.

The main objective of initializing the CSA is to prepare it to implement the framework of the proposed scheme. This includes the following: i) Cloud security administrators install the CSA on VCC — the installation of the CSA includes generating a non-migratable key pair (Pr,Pu) to protect domain secrets; and ii) the CSA manages security administrator(s) credentials and securely stores them to be used whenever administrator(s) need to be authenticated to CSA.

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

The CSA then needs to ensure that only security administrators can use the CSA. For this the CSA instructs security administrators to provide their authentication credentials (e.g. password/PIN), as described by Algorithm 2. The objective of this algorithm is to enrol security administrators into the CSA. The CSA then requests the TPM to store the authentication credentials of the Cloud 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 server agent. The authentication credential is used to authenticate security administrators before using the CSA; 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 CSA secrets. The protocols are used by security administrators when interacting with the server agent.

Algorithm 1 CSA initialization

1. CSA→ TPM: $\text{TPM}_{CreateWrapKey}$.
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→ CSA: TPM_KEY12[Pu, Encrypted Pr, TPM_KEY_STORAGE, tpmProof=TPM (NON-MIGRATABLE), S, Auth_data]

Algorithm 2 Administrators registration

1. CSA→ Administrators: Request for security administrators authentication credentials.
2. CSA→ TPM: $\text{TPM}_{LoadKey2}$(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, CSA returns an appropriate error message.
3. CSA→ TPM: TPM_{Seal}(Authentication_Credential).

Algorithm 3 Authentication verification

1. CSA→ Administrators: Request for authentication credentials.
2. CSA→ TPM: $\text{TPM}_{LoadKey2}$(Pr). TPM on CSA's device 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, CSA returns an appropriate error message.
3. CSA→ TPM: TPM_{Unseal}(Authentication_Credential).
4. TPM: Decrypts the string Authentication_Credential and passes the result to CSA.
5. CSA: Authenticates the administrators using the recovered authentication credentials. If authentication fails, CSA returns an appropriate error message.

6.8.2 LaaS Server Agent Initialization

The process of initializing LSA exactly follows the same process and algorithms described for initializing CSA in Section 6.8.1. The main differences are as follows: i) LaaS should be managed by provenance security administrators who should not have access to the CSA. Similarly, CSA security administrators should not have access to the LSA; ii) LaaS should have its specific non-migratable key pair which is independent from CSA key pair; and iii) as we outlined in Section 6.3.2, although both LSA and CSA run at VCC; however, this does not mean that VCC is a single entity. It is most likely to be the opposite (as currently implemented, for example, in OpenStack) has multiple different entities each could be allocated a specific function for scalability, performance, and security reasons.

6.8.3 LCA and CCA Initialization

This section describes the procedure of initializing client agents, which could be LCA or CCA. The goal of this procedure is to prepare devices to participate in Clouds. This covers generating

a non-migratable key to protect important credentials at client devices.

The protocol of initializing a LCA and CCA is described by Algorithm 4. The objective of this algorithm is to install a copy of the agent, which generates a non-migratable key to protect device's credentials. **TPM**, S and **(Pu, Pr)** have the same meanings provided earlier.

Algorithm 4 LCA Initialization — this equally applies to CCA initialization

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

6.8.4 LaaS Domain Establishment

In this section we discuss the procedure of establishing LaaSD, which are managed by the LSA. In the provided protocol we use the same notations described earlier. In this subsection we require that the LSA has already been installed and initialized, exactly as described earlier in Section 6.8.2. This includes installing the LSA, which interacts with the TPM to generate a non-migratable key pair that can be only used by the agent. This key pair is used to protect LaaS secrets.

LaaSD establishment begins when provenance security administrators want to establish a LaaSD. The administrators instruct the LSA to create a new LaaSD. The server agent authenticates administrators as described by the Algorithm 3. If authentication succeeds the server agent interacts with the TPM to securely generates the LaaS specific domain key k_{laas} and identifier i_{laas}, and a specific key $k_{laas-cca}$ to be used to establish trusted channel between LCA and CCA. These are described by Algorithm 5.

At the successful completion of this protocol LaaS credentials are initialized, which include the domain key, the domain identifier, the LCA-CCA key and an empty PKL. These are protected by LSA running at VCC, which manages LaaSD membership.

Provenance security administrators assign selected physical devices to LaaSD based on the devices properties that could fulfil the required overall LaaSD properties. As we discuss in Section 6.8.5, the LSA securely transfers the domain credentials to joining log devices. It also transfers the key $k_{laas-cca}$ associated with i_{laas} to the CSA. The CSA in turn transfers the key to joining CCA (see Section 6.8.7), which would establish an offline chain of trust between CCA and LCAs.

Algorithm 5 LaaSD establishment

1. LSA→ TPM: $\text{TPM}_{GetRandom}$.
   TPM generates a random number to be used as a LaaSD key k_{laas}.
2. TPM→ LSA: k_{laas}
3. LSA→ TPM: $\text{TPM}_{GetRandom}$.
   TPM generates a random number to be used as a LCA-CCA key $k_{laas-cca}$.
4. TPM→LSA: $k_{laas-cca}$
5. LSA→ TPM: $\text{TPM}_{GetRandom}$.
   LSA generates a unique number to be used as LaaSD identifier i_{laas}.
6. TPM→LSA: i_{laas}
7. The LaaSD credentials k_{laas}, i_{laas}, $k_{laas-cca}$, and an empty PKL_{laas} are stored in VCC protected storage, and sealed to the LSA so that only the LSA can access these credentials when its execution status is trusted. This is achieved as follows.
   LSA→ 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, LSA returns an appropriate error message.
   LSA→ TPM: $\text{TPM}_{Seal}(k_{laas}||i_{laas}||\text{PKL}_{laas})$.
   TPM securely stores the string $k_{laas}||i_{laas}||\text{PKL}_{laas}$ using the platform protected storage, such that they can only be decrypted on the current platform by LSA, and only if the platform runs as expected (when the platform PCR values matches the ones associated with Pr, i.e. S).

6.8.5 Adding Devices to LaaSD

This section describes the process for adding a device to a LaaSD. Following notations are used in the provided protocol: $\mathbf{TPM}_{LCA}$ is the TPM of the device running the LCA; $\mathbf{TPM}_{LSA}$ is the TPM of the device running the LSA; S_{LCA} is the platform state at release as stored in the PCR inside the TPM_{LCA}; S_{LSA} is the platform state at release as stored in the PCR inside the TPM_{LSA}; **(Pu**$_{LCA}$**, Pr**$_{LCA}$**)** is non-migratable key pairs such that the private part of the key Pr_{LCA} is bound to TPM_{LCA} and to the platform state S_{LCA}; **(Pu**$_{LSA}$**, Pr**$_{LSA}$**)** is non-migratable key pairs such that the private part of the key Pr_{LSA} is bound to TPM_{LSA} and to the to the platform state S_{LSA}; i_{laas} is LaaSD specific identifier; **PKL** is the LaaSD public key list; k_{laas} is the LaaSD-specific content protection key; $k_{laas-cca}$ is the LCA-CCA specific key for protecting content transferred between CCA and LaaS and to establish trust between both entities; $\mathbf{Cert}_{LSA}$ is the LSA device certificate; $\mathbf{Cert}_{LCA}$ is the joining LCA device certificate; A_{LSA} is an identifier for the LaaS server device included in Cert_{LSA}; A_{LCA} is an identifier for the LaaS client device included in Cert_{LCA}; $\mathbf{Pr}_{LSA-AIK}$ is the corresponding private key of the public key included in Cert_{LSA}; $\mathbf{Pr}_{LCA-AIK}$ is the corresponding private key of the public key included in Cert_{LCA}; N_1 is a randomly generated nonce; N_2 is a randomly generated nonce; $e_{Pu_{LCA}}(Y)$ denotes the asymmetric encryption of data Y using key Pu_{LCA}, 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 LCA sends a join domain request to the LSA. This request includes the LaaSD specific identifier i_{laas} this is achieved as follow.

LCA→LSA: Join_Domain

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

Adding a device into a domain uses Algorithm 7, which starts upon successful completion

of Algorithm 6. The objective of Algorithm 7 is to securely transfer the key k_{laas} and $k_{laas-cca}$ to the LCA. Both keys are sealed on the device hosting the LCA, so that they are only released to the LCA when its execution environment is as expected. If the execution status of the device running LCA is trusted, LSA 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_{laas} and the LCA-CCA specific key to LCA using Algorithm 7. The keys are sealed on LCA's device, so that they are only released to LCA when its execution environment is as expected.

Upon the successful completion of the above algorithms the LaaS client and server agents establish a trusted secure communication channel that is used to transfer the LaaSD key and policy to the LCA. The established secure channel, importantly, provides the assurance to the LSA about the state of the client agent and forces the future use of the transferred key to the agent on specific trusted state. The device hosting LCA is now part of the domain, as it possesses a copy of the key k_{laas}, and its public key matches the one stored in the server agent. Member devices of the domain can access the domain log records which are now shared by all devices member of the LaaSD.

Algorithm 6 LCA and LSA mutual authentication

1. LSA→ TPM$_{LSA}$: TPM$_{GetRandom}$.
2. TPM$_{LSA}$ →LSA: Generates a random number to be used as a nonce N_1.
3. LSA→ TPM$_{LSA}$: TPM$_{LoadKey2}$(Pr$_{LSA-AIK}$);
   Loads the server agent hosting device AIK in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr$_{LSA-AIK}$.
4. LSA→ TPM$_{LSA}$: TPM$_{Sign}$(N$_1$).
5. TPM$_{LSA}$ → LSA →LCA: N_1||Cert$_{LSA}$||Sign$_{LSA}$(N_1).
6. LCA: verifies Cert$_{LSA}$, extracts the signature verification key of LSA from Cert$_{LSA}$, and checks that it has not been revoked, e.g. by querying an OCSP service [MAM+99]. LCA then verifies message signature. If the verifications fail, LCA returns an appropriate error message.
7. LCA→ TPM$_{LCA}$: TPM$_{GetRandom}$.
8. TPM$_{LCA}$ →LCA: Generates a random number N_2 that is used as a nonce.
9. LCA→ TPM$_{LCA}$: TPM$_{LoadKey2}$(Pr$_{LCA-AIK}$);
   Loads the private key Pr$_{LCA-AIK}$ in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr$_{LCA-AIK}$.
10. LCA→ TPM$_{LCA}$: TPM$_{CertifyKey}$(SHA1(N_2||N_1||A_{LSA}||i_{laas}),Pu$_{LCA}$). TPM$_{LCA}$ attests to its execution status by generating a certificate for the key Pu$_{LCA}$.
11. TPM$_{LCA}$ →LCA: N_2||N_1||A_{LSA}||Pu_{LCA}||S_{LCA}||i_{laas}|| Sign$_{LCA}$(N_2||N_1||A_{LSA}||i_{laas}||Pu_{LCA}||S_{LCA}).
12. LCA→LSA: N_2||N_1||A_{LSA}||Pu_{LCA}||S_{LCA}||i_{laas}||Cert$_{LCA}$|| Sign$_{LCA}$(N_2||N_1||A_{LSA}||i_{laas}||Pu_{LCA}||S_{LCA}).
13. LSA verifies Cert$_{LCA}$, extracts the signature verification key of LCA from the certificate, and checks that it has not been revoked, e.g. by querying an OCSP service. LSA 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_{LSA}. LSA determines if LCA is executing as expected by comparing the platform state given in S_{LCA} with the predicted platform integrity metric. If these validations fail, then LSA returns back an appropriate error message.

Algorithm 7 Sealing LaaSD credentials to LCA

1. LSA→ TPM$_{LSA}$: TPM$_{LoadKey2}$(Pr$_{LSA}$).

   TPM on LSA loads the private key Pr$_{LSA}$ in the TPM trusted environment, after verifying the current PCR value matches the one associated with Pr$_{LSA}$ (i.e. S_{LSA}). If the PCR value does not match S_{LSA}, the server agent returns an appropriate error message.

2. LSA→ TPM$_{LSA}$: TPM$_{Unseal}$($k_{laas}||k_{laas-cca}||i_{laas}||$PKL).

3. TPM$_{LSA}$ →LSA: decrypts the string $k_{laas}||k_{laas-cca}||i_{laas}||$PKL and passes the result to LSA.

4. LSA verifies i_{laas} matches the recovered domain identifier and Pu$_{LCA}$ is included in the PKL. If so LSA encrypts k_{laas} and $k_{laas-cca}$ using the key Pu$_{LCA}$ as follows $e_{Pu_{LCA}}(k_{laas}||k_{laas-cca})$.

5. LSA→ TPM$_{LSA}$: TPM$_{CertifyKey}$(SHA1($N_2||A_{LCA}||e_{Pu_{LCA}}(k_{laas}||k_{laas-cca})$),Pu$_{LSA}$).

6. TPM$_{LSA}$ →LSA: attests to its execution status by generating a certificate for the key Pu$_{LSA}$, and sends the result to LSA.

7. LSA→LCA: $N_2||A_{LCA}||Pu_{LSA}||S_{LSA}||e_{Pu_{LCA}}(k_{laas}||k_{laas-cca})||\text{Sign}_M(N_2||A_{LCA}||e_{Pu_{LCA}}(k_{laas}||k_{laas-cca})||Pu_{LSA}||S_{LSA})$.

8. The device LCA verifies message signature, it is the intended recipient by checking the value of A_{LCA}, and verifies message freshness by checking the value of N_1. If verifications succeed, LCA stores the string $e_{Pu_{LCA}}(k_{laas}||k_{laas-cca})$ in its storage.

6.8.6 Establishing Trust between Server Agents

Before establishing an MD domain we should first establish a chain of trust between both CSA and LSA. This would help in establishing a transparent chain of trust between the CCA running at each member device of MD and LCA that runs at each member device of LaaSD, as we discuss it latter. For clarity we do not assume that the LSA and CSAs are hosted at a single VCC (as we indicated earlier VCC could be composed of multiple, but collaborating entities). Following notations are used in the provided protocol: **TPM**$_{LSA}$ is the TPM of the device running the LSA; **TPM**$_{CSA}$ is the TPM of the device running the CSA; S_{LSA} is the platform state at release as stored in the PCR inside the TPM$_{LSA}$; S_{CSA} is the platform state at release as stored in the PCR inside the TPM$_{CSA}$; **(Pu**$_{LSA}$**, Pr**$_{LSA}$**)** is non-migratable key pairs such that the private part of the key Pr$_{LSA}$ is bound to TPM$_{LSA}$ and to the platform state S_{LSA}; **(Pu**$_{CSA}$**, Pr**$_{CSA}$**)** is non-migratable key pairs such that the private part of the key Pr$_{CSA}$ is bound to TPM$_{CSA}$ and to the to the platform state S_{CSA}; i_{laas} is LaaSD specific identifier; i_{md} is MD domain specific identifier; k is a specific shared key between Cloud and LSAs; **Cert**$_{CSA}$ is the LSA device certificate; **Cert**$_{LSA}$ is the CSA device certificate; A_{CSA} is an identifier for the LSA device included in Cert$_{CSA}$; A_{LSA} is an identifier for the CSA device included in Cert$_{LSA}$; **Pr**$_{CSA-AIK}$ is the corresponding private key of the public key included in Cert$_{CSA}$; **Pr**$_{LSA-AIK}$ is the corresponding private key of the public key included in Cert$_{LSA}$; N_1 is a randomly generated nonce; N_2 is a randomly generated nonce; $e_{Pu_{LSA}}(Y)$ denotes the asymmetric encryption of data Y using key Pu$_{LSA}$, 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 LSA sends an establish trusted channel request to the CSA as follow.

LSA→CSA: Establish_Trusted_Channel

Two algorithms are then initiated to establish the trusted channel and to transfer management data across. The first algorithm involves the LSA and CSA to mutually authenticate each other conforming to the three-pass mutual authentication protocol [Int98]. The agents attest each other to prove their trustworthiness. These steps are achieved using an algorithm which is exactly the same as the one in 6. The second algorithm (Algorithm 8) starts upon successful completion of Algorithm 6. The objective of this algorithm is to securely establish a shared key k that can only be accessed by both agents when their execution status is as expected. Upon the successful completion of the two algorithms the LSA and CSAs establish a trusted secure

communication channel that is used to transfer the related provenance policy and other secret data between both agents. In addition, such a trusted channel, as we discuss latter, would help in establishing a transparent chains of trust between LCAs and CCAs. The established trusted secure channel provides the assurance to both agents about their states and forces the future use of the transferred key to be on specific trusted state. Next sections build on successful completion of the provided protocols when storing and querying log records, and when validating the trustworthiness of the log management processes.

Algorithm 8 Sealing the LSA-CSA shared key to LSA/CSA server agents

1. Note that, and as indicated in the text, for this algorithm to make sense it must be read after the attestation algorithm to show the reader how both entities (i.e. LSA and CSA) attest to each other execution environment and exchange their certificates.

2. CSA→ TPM: $\text{TPM}_{GetRandom}$.

   TPM generates a random number to be used as a shared key k.

3. TPM→ CSA: k

4. k is stored in CSA protected storage, and sealed to the CSA so that only the CSA can access the key when its execution status is trusted. This is achieved as follows.

   CSA→ 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, CSA returns an appropriate error message.

   CSA→ TPM: $\text{TPM}_{Seal}(k)$.

   TPM securely stores the key k using the platform protected storage, such that they can only be decrypted on the current platform by CSA, and only if the platform runs as expected (when the platform PCR values matches the ones associated with Pr, i.e. S).

5. CSA then encrypts k using the key Pu_{LSA} as follows $e_{Pu_{LSA}}(k)$.

6. CSA→ TPM_{CSA}: $\text{TPM}_{CertifyKey}$(SHA1($N_2||A_{LSA}||e_{Pu_{LSA}}(k)$),$\text{Pu}_{CSA}$).

7. TPM_{CSA} →CSA: attests to its execution status by generating a certificate for the key Pu_{CSA}, and sends the result to CSA.

8. CSA→ LSA: $N_2||A_{LSA}||Pu_{CSA}||S_{CSA}||\ e_{Pu_{LSA}}(k)||\text{Sign}_{CSA}(N_2||A_{LSA}||e_{Pu_{LSA}}(k)||\ Pu_{CSA}||S_{CSA})$.

9. LSA verifies message signature, it is the intended recipient by checking the value of A_{LSA}, and verifies message freshness by checking the value of N_1. If verifications succeed, LSA stores the string $e_{Pu_{LSA}}(k))$ in its storage.

   As in the case of CSA, the key k can only be decrypted on the current platform by CSA, and only if the platform runs as expected.

6.8.7 MD establishment and Management

In this subsection we require that the CSA has already been installed and initialized, LaaSD has been established, and a trusted channel between LSA and CSA has been established, exactly as described earlier in Sections 6.8.1, 6.8.4, and 6.8.6, respectively. The establishment of an MD follows similar steps to those provided in Algorithm 5 with the following changes: i) the CSA does not generate the shared LCA-CCA key, it rather requests it from the LSA using the trusted channel established in Algorithm 8; and ii) after the CCA receives this key, it securely stores the key along with other MD credentials.

Adding a device to MD also follows similar steps to those provided in Algorithms 6 and 7 with the following changes: i) the mutual authentication protocol (Algorithms 6) needs to be updated to establish a chain of trust between CSA and CCA rather than LSA and LCA; ii) a chain of trust need to be established between LCA and CCA in Algorithm 7. This is transparently established when CSA sends the shared LCA-CCA key to CCA (how this is achieved is discussed in Section 6.10); iii) the CSA regularly receives changes related to provenance management and policies from LSA using the trusted channel established in Algorithm 8; and iv) the CSA (by collaborating with the LSA, as in point iii) sends to CCA the metadata to use with the log records such as: physical device-id reflecting the CCA's device identifier at VCC database, MD-id the CCA is a member of, CMD-ids the MD is a member of, VMs that the CCA

would manage, and the policy that controls how the CCA interacts with the LCA.

6.8.8 Secure Log Storage

In this section we discuss a possible approach for storing Cloud provenance data using the proposed LaaSD. We, now, list the main steps for storing a log record generated by a process P which is hosted at physical device D. Whenever a process P generates a log record, LOG, it sends the LOG to the CCA running at D as follows:

1. $P \rightarrow$CCA: $LOG||APP_{ID}$ (where APP_{ID} is the process unique identifier which produces the LOG)

The CCA, as discussed earlier, is assigned to a LaaSD and a pre-agreed shared CCA-LCA specific key, $k_{laas-cca}$. Such key can only be accessed by the assigned agents when their execution environment is as expected. We assume, for performance reasons, that the CCA and LCAs keep such keys pre-loaded in memory (we assumed in Assumption 6.7.1 that a mechanism is in place to protect sensitive data whilst being in memory). Loading such key is done as follows.

1. $CCA \rightarrow$ TPM: $\text{TPM}_{\text{L}oadKey2}$(Pr). TPM on D 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, CCA returns an appropriate error message.
2. $CCA \rightarrow$ TPM: $\text{TPM}_{\text{U}nseal}(k_{laas-cca})$.

CCA associates additional metadata representing the virtual and physical layer details (i.e. virtual domain id (VD_{ID}), virtual machine id (VM_{ID}), physical machine id (PH_{ID}), and physical domain id (PHD_{ID})), and then encrypts the string using the shared key key $k_{laas-cca}$. CCA then sends the result to the LaaS agent as follows.

$$CCA \rightarrow LaaS\text{: } e_{k_{laas-cca}}(LOG||APP_{ID}||VM_{ID}||VD_{ID}||PH_{ID}||PHD_{ID})$$

As discussed above, we require that $LaaS$ pre-loads the shared key $k_{laas-cca}$. LaaS then decrypts the string, and re-encrypts only the LOG field using the LaaSD specific key k_{laas} as follows.

1. $LaaS \rightarrow$ TPM: $\text{TPM}_{\text{L}oadKey2}$(Pr). TPM on a LaaS device 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, LaaS returns an appropriate error message.
2. $LaaS \rightarrow$ TPM: $\text{TPM}_{\text{U}nseal}(k_{laas}||K_{laas-cca})$.
3. $LaaS$ decrypts the string $e_{k_{laas-cca}}(LOG||APP_{ID}||VM_{ID}||VD_{ID}||PH_{ID}||PHD_{ID})$
4. $LaaS$ then encrypts the LOG field as follows: $e_{k_{laas}}(LOG)$

Finally, LaaS stores the encrypted LOG record and the extracted metadata in a set of tables inside the provenance DBMS (identified in Section 6.5). We require that the LaaS DBMS provides additional protection measures of the stored provenance data. Example of this is what is known by Oracle Wallet [Ora11a]. In this the DBMS automatically stores the data encrypted inside the DBMS. It is outside the scope of this chapter to discuss or analyze the process of securely storing data inside the DBMS.

6.9 Threat Analysis

In this section we informally analyse the threats, services and mechanisms for the provenance framework workflow proposed in section 6.8. We focus on the threats, services and mechanisms that apply to provenance and management data, and the MD and LaaSD domains' credential.

Provenance and Cloud security administrators when interacting with the server agents running at VCC could violate their privileges by adding unauthorized devices to a domain or even an unauthorized party could steal security administrators authentication credentials to add an unauthorized device into a domain. The *administrators authorization violation threat* can be mitigated by combining different measures, for example: (a) requiring that N out of M administrators successfully authenticate themselves directly to the VCC for request authorization; (b) using logging and auditing mechanisms that could detect abnormalities in the system; and (c) using the policy of separation of duty, for example, prevent administrators (both provenance and Cloud) from accessing log files, which are routinely examined by auditors. The *stealing of administrators credentials*, on the other hand, can be mitigated by using strong authentication measures which involve a combination of "something the administrator has" e.g. a smart card; "the security administrator is", e.g. biometric verification; and/or "the security administrator knows", e.g. a password or PIN. At this foundation stage, the chapter does not cover the implementation and enforcement of such mechanisms.

The server software agents running at VCC raise the following security threats when processing and storing system credentials: *unauthorized manipulation of system credentials during use in the VCC*, and/or *unauthorized manipulation of system credentials whilst stored in the VCC*. The *confidentiality and integrity protection of system credentials during execution in a VCC* requires process isolation techniques, in which software agents run in isolation, free from being observed or compromised by other processes running in the same protected partition, or by software running in any insecure partition. This chapter does not cover this point, however, we assumed in Assumption 6.7.1 that such a protection mechanism is in place. The *confidentiality and integrity of system credentials whilst stored in the VCC* requires protected storage capabilities, as discussed in section 6.5.3 and Algorithm 5. The protected storage capabilities uses TPM functions to protect domain credentials. TPM is tamper evident and so it is not easy for the protected credentials to get hacked in normal circumstances. However, TPM cannot protect itself from physical attacks and, in addition, domain keys could possibly be revealed in different ways such as brute-force attack. Lessening the impact of such threats requires key management. The Cloud policy makers decide on the key management policy (e.g. frequency of refreshing domain keys, what should happen if a device is hacked, etc). In this chapter we do not cover the key management part, neither we consider policy management and enforcement mechanisms.

The interaction between a client software agent running on a device joining a domain and the corresponding server software agent running at VCC raises the following threats to the corresponding domain key whilst in transit: *unauthorized reading or alteration of the domain key whilst in transit, the VCC wittingly/unwittingly sending the domain key to a malicious entity, a device wittingly/unwittingly receiving the domain key from a malicious entity, and a replay of communications between the VCC and the added device*. The *confidentiality and integrity of the domain whilst in transit*, as discussed in Section 6.8.5, is provided by the use of asymmetric encryption where we assume that the encryption primitive in use provides non-malleability. *Entity authentication of a device to a VCC* involves a protocol exchange between the device and the VCC, as discussed in Algorithm 6. It is initiated when the VCC and the joining device mu-

tually authenticate to each other. This mutual authentication attests to the scheme applications execution status and whether the platform is trusted. By this the VCC can only communicate with a trusted entity, and so cannot unwittingly send the domain key to a malicious entity. Similarly, the device agent, if it is not operating properly, cannot get the domain key and so it cannot wittingly send it to a malicious entity (see Algorithms 7 and 8). Similar discussion also applies to *entity authentication of a VCC to a device*. *Prevention of replay of communications between a VCC and a device* is provided by the inclusion of nonces in protocol messages (see Section 6.8.5).

Domain devices raise the following threats to the processing and storage of the domain key and content: *unausthorized reading or alteration of the domain key during use in the device, unauthorized reading or alteration of the domain key whilst stored in the device, unauthorized reading or alteration of content during use in the device, and unauthorized reading or alteration of content whilst stored in the device*. The *confidentiality and integrity of the domain key during execution on a device* is covered in Assumption 6.7.1 as discussed above for the VCC. The *confidentiality and integrity of the domain key whilst stored in a device*, as discussed above, does not only require protected storage capabilities but also key and policy management and enforcement mechanisms. The *confidentiality and integrity of domain content during execution on a device* follows the same discussion as of the point of protecting the domain key during execution in the device. The *confidentiality and integrity of domain content* is protected by encrypting it using the domain key whilst stored on a device where we assume that the encryption primitive in use provides authenticated encryption. The encryption key is bound to the device's trusted environment, as discussed in Section 6.8.8.

6.10 Discussion, Future Directions, and Conclusion

6.10.1 Establishing Trust

In this part we discuss the foundation of trust establishment between different CloudâĂŹs entities. A client or a verifier (which could, for example, be a Cloud customer, Cloud employee, or a third party) needs to assess the trustworthiness of a running application in the Cloud. This includes assessing the trustworthiness of a Cloud to manage the infrastructure and the provenance system. If the result is positive, the verifier can then trust the operation of Clouds and would only need to assess the trustworthiness of the running application. We now discuss how the proposed framework goes in this direction in more details — it is outside the scope of this chapter to go in the details of trust measurement.

As we discussed earlier one of the responsibilities of LSA is to establish a trustworthy LaaSD to manage the provenance data of Cloud elements. The first step is to install LCAs at carefully selected log-specific devices. LSA then verifies the trustworthiness of LCA and assures users about the trustworthy behaviour of LCA when managing the LaaSD. In other words, untrusted LCA will automatically be evicted from managing the LaaSD. Thus, a verifier only needs to measure and then assess the trustworthiness of LSA. If trusted, the verifier can then implicitly assume that LCA (which is managed by LSA) is trusted to manage the LaaSD. Assessing the trustworthiness of LSA is not enough by itself. This is because the operation of Clouds infrastructure (e.g. hosting of billing application) is managed by the CSA and CCA, while the log records is managed by LSA and LCA. Therefore, a verifier would also need to measure and then verify the trustworthiness of the CSA as well as the LSA. As in the case of assessing the trustworthiness of log management, a verifier does not need to measure and assess

the trustworthiness of the CCA. It is rather the opposite as the verifier should not, indeed, get involved into understanding complexities of Cloud infrastructure [AN11]. As in the case of LSA, one of the key functions of the CSA is to assure users that only trustworthy CCA can manage Cloud infrastructure and untrusted agents will automatically be evicted from the MD.

A chain of trust is also required between both CCA and LSA, which is provided based on the above chains of trust, as follows: i) we established a chain of trust between LSA and LCA; ii) we established a chain of trust between LSA and CSA; and iii) we established a chain of trust between CSA and CCA. Using these chains of trust, we transparently established a chain of trust between CCAs and LCAs.

To conclude, a verifier should not (read as must not) get involved into understanding the details of Cloud infrastructure. The identified chains of trust help in this direction, as a verifier only needs to attest to the trustworthiness of the requested application and the VCC which runs both the LSA and CSA.

6.10.2 Achievement of Objectives

Section 6.1.2 identifies four key requirement for trustworthy secure Clouds provenance which we now discuss the ones covered in this chapter. We partially address requirement (ii) as follows: a) provide a high level design of a provenance system which is built on distributed DBMS engine; b) associate each item of log record with a metadata identifying the recorded log in context of Cloud taxonomy; and c) identify the provenance system requirements. We covered requirement (iii) as follows: a) establish LaaSD which manages the secure sharing of provenance data between LaaSD member devices; b) update our previous work on Cloud infrastructure management ([AAM11]) to associate provenance metadata with log records; and c) integrate our previous work with this chapter framework enabling the secure transfer of log records from their originating processes to the log repository. The previous subsection discusses how we partially cover point (iv) which is related to trust management — more work are still needed on this point which is related to trust evaluation in Clouds.

Section 6.8.8 provides a possible approach of how the integrated framework could possibly work. However, this is not enough by itself to assure provenance data integrity and confidentiality whilst being stored and processed within the LaaSD. For example, this chapter do not discuss key management, policy management and protecting sensitive data whilst being processed. These are complex subjects, especially in Cloud context, to be covered in this chapter.

6.10.3 Conclusion

This chapter proposes a framework for trustworthy Cloud's provenance. Cloud provenance is a key requirement to establish the foundation for providing trust in the Cloud. Establishing trust in the Cloud requires trustworthy self-managed services that can automatically and with minimal human intervention manage Cloud users' resources at the Cloud infrastructure. Such self-managed services require trustworthy Cloud provenance as it helps in taking the right action on changes and incidents. Cloud provenance has many additional advantages, e.g. a key requirement in forensic investigation. This chapter does not provide an exhaustive secure framework neither we provide a formal security analysis of the framework. For example, we do not cover key management neither we cover database security subjects. This is because discussing such topics is a whole area of research in Clouds context.

Appendix A

Specification of the functional layer of the TrustedServer management protocol

```
AnyType := volatile type *

Array := volatile type 1

Boolean := volatile type 3

Integer := volatile type 4

String := volatile type 14

ByteArray := volatile type 15

Binary := volatile AnyType

Identifier := volatile type 16

Timestamp := volatile type 17

IPv4 := volatile type 18

IPv4withNetwork := volatile type 19

ASN1Encoded := volatile type 21

PublicKey := volatile ASN1Encoded

Certificate := volatile ASN1Encoded

CertificateChain := volatile [ Certificate ]

AES128Key := volatile ByteArray

AES128Encrypted := volatile ByteArray

RSAEncrypted := volatile ByteArray
```

```
SHA256CheckSum := volatile ByteArray

CodebookEntry := volatile type 22

Color := volatile ByteArray

Priority := volatile enum( lower, medium, higher, highest )

ExchangeMode := volatile enum( main, aggressive )

Cipher := volatile enum( 3des, aes128, aes256, blowfish, ↩
   cast )

Hash := volatile enum( sha1, sha256, sha512 )

DHGroup := volatile enum( modp768, modp1024, modp1536,
  modp2048, modp3072, modp4096, modp6144, modp8192 )

Network::TraceRouteExecution := job(
  -> object(
       ip -> IPv4
       ttl -> Integer
       delay? -> Integer
     )
) : ()

VirtualNetwork::PeerServer := object(
  id -> Identifier
  ip? -> IPv4
  policy -> NetworkPolicy::PolicyReference
  priority -> Priority
  internetAccess -> Boolean
)

VirtualNetwork::PeerNetwork := object(
  id -> Identifier
  network? -> ( IPv4withNetwork | [ IPv4 ] | IPv4 )
  servers? -> [ IPv4 ]
  policy? -> NetworkPolicy::PolicyReference
  priority -> Priority
  internetAccess -> Boolean
)

VirtualNetwork::Peer := object(
  id -> Identifier
  networks -> [ VirtualNetwork::PeerNetwork ]
  servers -> [ VirtualNetwork::PeerServer ]
  trustedNetworks? -> [ IPv4withNetwork ]
```

```
  ip? -> IPv4
  serial? -> ( Integer | [ Integer ] )
)

VirtualNetwork::Network := object(
  id -> Identifier
  tvd -> SecurityPolicy::TVDReference
  peers -> [ VirtualNetwork::Peer ]
  internetNat? -> Boolean
  if "defined internetNat then one of these" (
    internetRoutingIPs -> [ IPv4 ]
    internetRoutingPeer -> ref VirtualNetwork::Peer.id
  )
)

SecurityPolicy::TVDDescriptor := object(
  id -> Identifier
  name -> String
  color -> Color
  validityPeriod? -> Integer
)

SecurityPolicy::TVDReference := ref ↩
   SecurityPolicy::TVDDescriptor.id

SecurityPolicy::TVD := SecurityPolicy::TVDDescriptor + object(
  ikeSettings? -> object(
    identity -> CertificateStore::Handle
    ca -> CertificateStore::Handle
    exchangeMode -> ExchangeMode
    cipher? -> Cipher
    hash -> Hash
    dhGroup -> DHGroup
    pfsGroup? -> DHGroup
    lifetime -> Integer
  )
)

SecurityPolicy::IFConstraint := abstract object(
  id -> Identifier
  type -> CodebookEntry
)

SecurityPolicy::AskUserConstraint
  := SecurityPolicy::IFConstraint < "askUser" >

SecurityPolicy::VirusCheckConstraint
  := SecurityPolicy::IFConstraint < "checkVirus" >
```

```
SecurityPolicy::CheckUserConstraint
  := SecurityPolicy::IFConstraint < "checkUser" > + object(
  users -> [ User::UserReference ]
)

SecurityPolicy::IFPermission := object(
  id -> Identifier
  constraints -> [ ( SecurityPolicy::AskUserConstraint
                   | SecurityPolicy::VirusCheckConstraint
                   | SecurityPolicy::CheckUserConstraint ) ]
)

SecurityPolicy::InformationFlow := object(
  id -> Identifier
  source -> SecurityPolicy::TVDReference
  target -> SecurityPolicy::TVDReference
  permissions -> [ SecurityPolicy::IFPermission ]
)

Appliance::ClientState := volatile object(
  timestamp -> Timestamp
  compartments? -> [ Compartment::CompartmentState ]
)

Appliance::ServerState := volatile object(
  timestamp -> Timestamp
)

CertificateStore::Handle := Binary

CertificateStore::Entry := object(
  reference -> CertificateStore::Handle
  certificateChain? -> CertificateChain
  publicKey? -> PublicKey
  persistent -> Boolean
)

CertificateStore::CertificateStore := job(
  <- generateKeyPair( object(
       algorithm -> enum( rsa )
       bits -> Integer
       persistent -> Boolean
     ) ) : CertificateStore::Entry
  <- addCertificateChain( object(
       reference? -> CertificateStore::Handle
       certificateChain -> CertificateChain
       persistent -> Boolean
```

```
    ) ) : CertificateStore::Handle
  -> CertificateStore::Entry
  -> ()
) : ()

NetworkPolicy::ICMPType := Integer

NetworkPolicy::Rule := object(
  priority -> Priority
  protocol -> enum( icmp, tcp, udp )
  if "protocol == icmp" (
    types? -> [ NetworkPolicy::ICMPType ]
  ) else (
    port -> Integer
    endPort -> Integer
  )
  direction -> enum( inbound, outbound, bidirectional )
)

NetworkPolicy::Policy := object(
  id -> Identifier
  rules -> [ NetworkPolicy::Rule ]
)

NetworkPolicy::PolicyReference := ref NetworkPolicy::Policy.id

Debug::PTYExecution := job(
  <> ByteArray
  <- object(
       width -> Integer
       height -> Integer
    )
) : ()

NetworkDNSD::Entry := object(
  hostname -> String
  ip -> IPv4
)

Firmware::PackageUUID := ref Firmware::Package.uuid

Firmware::InstallationState := enum( ready, downloadFailed, ↩
   invalidFile )

Firmware::InstallExecution := job(
  <- object(
       uuid -> Firmware::PackageUUID
       key -> AES128Encrypted < AES128Key >
```

```
      url -> String
    )
  -> Firmware::InstallationState
  <- object(
      uuid -> Firmware::PackageUUID
      key -> AES128Encrypted < AES128Key >
      size -> Integer
    )
  <- ByteArray
  -> Firmware::InstallationState
) : Boolean

Firmware::Version := object(
  label -> String
  revision -> Integer
  firmware -> String
  release -> String
)

TimeSynchronizer::TimeSynchronizer := job(
  <> Timestamp
) : ()

DiskImage::DiskImage := object(
  size -> Integer
  sha256? -> SHA256CheckSum
  id -> Identifier
  name -> String
)

DiskImage::DownloadExecution := job(
  <- ByteArray
) : ()

Compartment::Template := object(
  id -> Identifier
  name -> String
  description -> String
  image -> DiskImage::DiskImage
  tvd -> SecurityPolicy::TVDReference
  nat -> Boolean
  dhcpd? -> object(
    dns? -> [ IPv4 ]
    domain? -> String
  )
  dnsd? -> object(
    hostnames? -> [ NetworkDNSD::Entry ]
    dns? -> [ IPv4 ]
```

```
  )
  priority -> Priority
  recommendations -> object(
    memory -> Integer
    cpus -> Integer
    videoMemory -> Integer
    operatingSystem -> enum( defaults, windows5 )
  )
)

Compartment::TemplateReference := ref Compartment::Template.id

Compartment::Host := object(
  id -> Integer
  name -> String
  serial -> String
)

Compartment::HostReference := ref Compartment::Host.id

Compartment::Compartment := object(
  id -> Identifier
  name -> String
  template -> Compartment::TemplateReference
  if "template.nat == false" (
    ip -> [ IPv4 ]
  )
  hosts -> [ Compartment::HostReference ]
  user -> ref Server::User.id
  shares? -> [ ( Server::SSHFSShare | ↩
     Server::AnonymousWebDavShare ) ]
)

Compartment::Reference := ref Compartment::Compartment.id

Compartment::CompartmentState := object(
  id -> Compartment::Reference
  status -> enum( running, stopped )
)

Client::Protocol := object(
  name -> String
  versions -> [ Integer ]
)

Client::ClientHello := volatile object(
  currentFirmware -> Firmware::Version
  protocol -> Client::Protocol
```

```
  supportedFeatures -> [ String ]
  timestamp -> Timestamp
  installationEncryptionKey -> PublicKey
  attachmentIdentifier? -> String
)

Client::RejectReason := enum( factoryReset, detach,
  authorizationPending )

Client::ServerHello := volatile object(
  currentFirmware -> Firmware::Version
  protocol -> Client::Protocol
  supportedFeatures -> [ String ]
  timestamp -> Timestamp
  rejectReason? -> [ Client::RejectReason ]
)

Client::OnlineConfiguration := volatile object(
  intranet -> [ IPv4withNetwork ]
  manager -> object(
    hosts -> [ ( IPv4 | String ) ]
    signatures -> [ CertificateStore::Handle ]
  )
  timestamp -> Timestamp
  events -> object(
    enableUnknown -> Boolean
  )
  system -> ManagedClient::System
  virtualNetworks -> [ VirtualNetwork::Network ]
  securityPolicy -> object(
    tvds -> [ SecurityPolicy::TVD ]
    informationFlows -> [ SecurityPolicy::InformationFlow ]
  )
  networkPolicies -> [ NetworkPolicy::Policy ]
  compartmentConfiguration -> object(
    templates -> [ Compartment::Template ]
    compartments -> [ Compartment::Compartment ]
  )
)

Client::Root := job(
  -> Client::ClientHello
  <- Client::ServerHello
  <- detachManager( Boolean ) : ()
  -> Appliance::ClientState
  <- Appliance::ServerState
  <- pty( object(
       folder -> String
```

```
      command -> String
      arguments? -> String
    ) ) : Debug::PTYExecution
  <- timeSynchronizer() : TimeSynchronizer::TimeSynchronizer
  <- configure( Client::OnlineConfiguration ) : ()
  <- install( RSAEncrypted < AES128Key > ) : ↩
     Firmware::InstallExecution
  <- certificateStore() : CertificateStore::CertificateStore
  -> eventManager() : Events::EventManager
  -> startUserSession( object(
      username -> String
      password -> String
    ) ) : Server::UserSession
  -> downloadDiskImage( object(
      image -> ref DiskImage::DiskImage.id
      resumeSHA256? -> SHA256CheckSum
      resumePosition? -> Integer
    ) ) : DiskImage::DownloadExecution
  <- reboot() : ()
  <- traceRoute( IPv4 ) : Network::TraceRouteExecution
  <- legacyStatus() : String
  <- legacySyslog( Integer ) : String
  <- legacyExecute( String ) : String
  <- legacyTunnelList() : String
  <- legacyTunnelDetails( String ) : String
) : ()

ManagedClient::System := object(
  id -> Identifier
  name -> String
  company -> Identifier
  serial -> String
)

Events::ObjectReference := volatile Integer

Events::EventManager := job(
  <- Integer
  -> object(
      company -> ref ManagedClient::System.company
      appliance -> ref ManagedClient::System.id
    )
  -> object(
      sequenceNumber -> Integer
      timestamp -> Timestamp
      source -> CodebookEntry
      type -> Integer
```

```
        arguments -> Array < ( String | ↩
           Events::ObjectReference ) >
      )
) : ()

TrustedChannel::RemoteAttestation := volatile Binary < ↩
   PublicKey >

User::User := object(
  id -> Identifier
  name -> String
  mail? -> String
)

User::UserReference := ref User::User.id

Server::Share := abstract object(
  id -> Identifier
  name -> String
  type -> CodebookEntry
)

Server::SSHFSShare := Server::Share + object(
  hostname -> String
  path -> String
)

Server::AnonymousWebDavShare := Server::Share + object(
  hostname -> String
  path -> String
)

Server::User := object(
  id -> Identifier
  name -> String
  expireDate? -> Integer
  hosts -> [ Compartment::Host ]
  tvds -> [ SecurityPolicy::TVDDescriptor ]
  templates -> [ Compartment::Template ]
  compartments -> [ Compartment::Compartment ]
)

Server::Reason := enum( noSuchUser, disabled, expired )

Server::UserSession := job(
  <- Server::User
  -> createCompartment( object(
       template -> Compartment::TemplateReference
```

```
      name -> String
    ) ) : Compartment::Reference
  -> copyCompartment( object(
      compartment -> Compartment::Reference
      name -> String
    ) ) : Compartment::Reference
  -> renameCompartment( object(
      compartment -> Compartment::Reference
      name -> String
    ) ) : Boolean
  -> deleteCompartment( Compartment::Reference ) : Boolean
  -> addCompartment( object(
      compartment -> Compartment::Reference
      host -> Compartment::HostReference
    ) ) : Boolean
  -> removeCompartment( object(
      compartment -> Compartment::Reference
      host -> Compartment::HostReference
    ) ) : Boolean
) : Server::Reason

Server::Server :=
  TrustedChannel::RemoteAttestation < "identityKey", ↩
    Client::Root >
```

Bibliography

[AA08] 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.

[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. Operational trust in clouds' environment. In *MoCS 2011: Proceedings of the Workshop on Management of Cloud Systems*, pages 141–145. IEEE, June 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.

[Abb13] Imad M. Abbadi. A framework for establishing trust in cloud provenance. *International Journal of Information Security*, 12(2):111–128, 2013.

[AFG+] 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.

[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.

[Ama10] Amazon. Amazon Elastic Compute Cloud (Amazon EC2), 2010. http://aws.amazon.com/ec2/.

[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.

[BG11] Sören Bleikertz and Thomas Groß. A Virtualization Assurance Language for Isolation and Deployment. In *IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY'11)*. IEEE, Jun 2011.

[BGJ+05] Anthony Bussani, John Linwood Griffin, Bernhard Jansen, Klaus Julisch, GÃijenter 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 *ACM Cloud Computing Security Workshop (CCSW'11)*. ACM, Oct 2011.

[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.

[BLCSG12] Shakeel Butt, H. Andres Lagar-Cavilla, Abhinav Srivastava, and Vinod Ganapathy. Self-service cloud computing. In *19th ACM Conference on Computer and Communications Security (CCS'12)*. ACM, October 2012.

[CA08] Daniel Crawl and Ilkay Altintas. A Provenance-Based Fault Tolerance Mechanism for Scientific Workflows. In *Provenance and Annotation of Data and Processes*, volume 5272 of *LNCS*, pages 152–159. Springer, 2008.

[CFH+05] C. Clark, K. Fraser, S. Hand, J.G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX, 2005.

[CSA10] CSA. Top threats to cloud computing v1.0. Technical report, Cloud Security Alliance (CSA), mar 2010.

[Dig92] Digital Equipment Corporation — Maynard, Massachusetts. Information technology — database language sql, 1992. http://www.contrib.andrew.cmu.edu/s̃hadow/sql/sql1992.txt.

[ENI09] ENISA. Cloud computing: Benefits, risks and recommendations for information security. Technical report, European Network and Information Security Agency (ENISA), nov 2009.

[GBG+06] Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *Third International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer, 2006.

[GdR+11] Amir Hossein Ghamarian, Maarten Mol de, Arend Rensink, Eduardo Zambon, and Maria Zimakova. Modelling and analysis using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)*, March 2011.

[HL09] Jun Ho Huh and John Lyle. Trustworthy log reconciliation for distributed virtual organisations. In *Proceedings of the 2nd International Conference on Trusted Computing*, Trust '09, pages 169–182, Berlin, Heidelberg, 2009. Springer-Verlag.

[HM08] Jun Ho Huh and Andrew Martin. Trusted logging for grid computing. In *Third Asia-Pacific Trusted Infrastructure Technologies Conference*. IEEE, 2008.

[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.

[JNL] Keith Jeffery and Burkhard NeideckerLutz. The Future of Cloud Computing — Opportunities For European Cloud Computing Beyond 2010.

[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.

[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.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving xen security through disaggregation. In *4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments (VEE'08)*. ACM, 2008.

[MRMS09] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo I. Seltzer. Making a cloud provenance-aware. In *TaPP '09: Proceedings of the First Workshop on the Theory and Practice of Provenance*, 2009.

[MRMS10] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *FAST '10: Proceedings of the 8th USENIX conference on File and storage technologies*, pages 15–14. USENIX, 2010.

[Ope] openstack. http://www.openstack.org.

[Ora11a] Oracle. Oracle Advanced Security Administrator's Guide — Using Oracle Wallet Manager, 2011. http://docs.oracle.com/cd/B10501_01/network.920/a96573/asowalet.htm.

[Ora11b] Oracle. Oracle Real Application Clusters (RAC), 2011. http://www.oracle.com/technetwork/database/clustering/overview/index.html.

[Ren] Arend Rensink. GROOVE: GRaphs for Object-Oriented VErification. http://groove.cs.utwente.nl/.

[RK09] A. Rensink and J-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. In A. Boronat and R. Heckel, editors, *Graph transformation and visual modelling techniques, York, U.K.*, volume 18 of *Electronic Communications of the EASST*. EASST, 2009.

[RN09] Christine F. Reilly and Jeffrey F. Naughton. Transparently Gathering Provenance with Provenance Aware Condor. In James Cheney, editor, *TaPP '09: Proceedings of the First Workshop on the Theory and Practice of Provenance*, San Francisco, CA, USA, 2009. USENIX.

[Rob12] Roberto Sassu et al. TClouds – Initial Component Integration, Final API Specification, and First Reference Platform. Deliverable D2.4.2, TClouds Consortium, October 2012.

[Sad08] Ahmad-Reza Sadeghi. Trusted computing — special aspects and challenges. In V. Geffert et al., editor, *SOFSEM*, volume 4910 of *LNCS*, pages 98–117. Springer-Verlag, Berlin, 2008.

[SCP+02] C.P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M.S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, 36(SI):377–390, 2002.

[SGR09] N. Santos, K.P. Gummadi, and R. Rodrigues. Towards trusted cloud computing. In *Hot topics in cloud computing (HotCloud'09)*. USENIX, 2009.

[SMV+10] Joshua Schiffman, Thomas Moyer, Hayawardh Vijayakumar, Trent Jaeger, and Patrick McDaniel. Seeding clouds with trust anchors. In *ACM workshop on Cloud computing security (CCSW'10)*. ACM, 2010.

[SPG05] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[SSW08] Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Property-based TPM virtualization. In *11th International Conference on Information Security (ISC'08)*, volume 5222. Springer, 2008.

[SWZ99] Andy Schürr, Andreas J. Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of graph grammars and computing by graph transformation: vol. 2: applications, languages, and tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[Tae03] Gabriele Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In John L. Pfaltz, Manfred Nagl, and Boris BÃűhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer, 2003.

[tpm] TPM main specification. http://www.trustedcomputinggroup.org/files/static_page_files/72C26AB5-1A4B-B294-D002BC0B8C062FF6/TPM%20Main-Part%201%20Design%20Principles_v1.2_rev116_01032011.pdf.

[Tru07] Trusted Computing Group. *TPM Main, Part 2, TPM Structures. Specification version 1.2 Revision 103*, 2007.

[VMw12] VMware. VMware vCenter Server, 2012. http://www.vmware.com/products/vcenter-server/.

[Xu05] Jie Xu. Provenance-Aware Fault Tolerance for Grid Computing. http://spiderman-2.laas.fr/IFIPWG/Workshops&Meetings/48/RR/03-Xu.pdf, 2005.