D2.3.1
Requirements, Analysis, and Design of Security 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:	Deliverable
Deliverable reference number:	ICT-257243 / D2.3.1 / 1.0
Activity and Work package contributing to deliverable:	Activity 2 / WP 2.3
Due date:	September 2011 – M12
Actual submission date:	3rd October, 2011

Responsible organisation:	IBM
Editor:	Christian Cachin
Dissemination level:	Public
Revision:	1.0

Abstract:	This report describes requirements, approaches, concrete interfaces, protocols, and verification methods for managing secure and trustworthy cloud resources according to the TClouds approach.
Keywords:	Interfaces, migration, cryptographic key management, ontology verification.

Editor

Christian Cachin (IBM)

Contributors

Sören Bleikertz, Christian Cachin, Thomas Groß, Matthias Schunter (IBM)

Michael Gröne, Norbert Schirmer (SRX)

Imad M. Abbadi (OXFD)

Daniele Canavese, Emanuele Cesena, Gianluca Ramunno, Jacopo Silvestro, Paolo Smiraglia, Davide Vernizzi (POL)

Johannes Behl, Rüdiger Kapitza, Klaus Stengel (FAU)

Sven Bugiel, Stefan Nürnberger (TUD)

Disclaimer

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

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

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

Executive Summary

This report describes requirements, approaches, concrete interfaces, protocols, and verification methods for managing secure and trustworthy cloud resources according to the TClouds approach.

Security management spans a variety of different approaches. On the one hand, new mechanisms are proposed here that exploit distinctive features of securely managing cloud resources such as to result in an overall increased level of trustworthiness. This is coupled with protocols for managing encryption keys for cloud storage. Encryption is an established technique for protecting data stored in the cloud, but its usability hinges critically on making the necessary encryption and decryption keys available to the involved parties in a secure way. This report proposes a novel architecture and novel solutions for managing keys in IaaS clouds providing access to virtual machines, with their virtual disk images encrypted by keys stored at clients.

In order to automate management of cloud resources in a heterogeneous cloud-of-clouds environment, it is necessary to use a common language across different service providers. In this scenario, ontologies offer a suitable tool for describing the various resources. Since they are formally defined, one can exploit this to infer security properties through automatic reasoning. This report, furthermore, proposes novel ontologies suitable for managing secure and trustworthy cloud computing systems.

One important goal of cloud-infrastructure management is to prevent information leakage between different domains (e.g., tenants or legally separated entities). As another goal, this report shows how to automate an information-flow analysis for large-scale heterogeneous virtualized infrastructures. This work aims at reducing the apparent complexity for human administrators, who should implement a complex system following defined trust assumptions, through automatic tool-based analysis of information properties in the running system.

Contents

List of Figures

List of Tables

Chapter 1

Overview of Security Management

Chapter Authors:
Christian Cachin (IBM)

1.1 Introduction

Cloud computing systems must be managed like any other IT system. In the cloud computing model, a client accesses the resources of a provider. The provider takes care of managing the resources and allocating them to clients. Due to the nature of the resources that are shared, the management functions in these environments differ from those found in traditional IT systems in several ways.

A novel aspect in infrastructure (or platform) management for cloud computing lies in the shared responsibility between provider and client for the system. Whereas traditionally one entity was largely responsible for all aspects, including security, this responsibility is now shared. A clearly defined separation of the respective tasks is required. A second novel aspect is the sharing of infrastructure resources among multiple clients, usually called tenants in this context.

Separating between client-managed and provider-managed operations is crucial also for security management. The introduction of cloud computing breaks up the previously well-defined boundaries of the client infrastructure. For example, IT resources from a potentially untrusted provider in the cloud are now consumed instead of resources produced in-house.

Since cloud computing provides virtualized resources, it is much easier to allocate them than it is to allocate physical resources. This can lead to a much faster introduction and adoption of such resources than previously. Obviously, the low cost of cloud-provided resources is one of the most interesting features of cloud computing. But its ease-of-use also leads to widespread uncontrolled proliferation of resources and overly complex infrastructures, and clients have had difficulty in the past to control their virtual inventory (as characterized by the buzzword "virtual machine sprawl").

1.2 Classification

This report describes requirements, analysis, and design of security management functions for trustworthy cloud computing. According to the general taxonomy of Avizienis *et al.* [ALRL04], the methods for building dependable (i.e., secure and resilient) systems can be grouped into

- prevention mechanisms;
- tolerance mechanisms;

- removal mechanisms; and
- forecasting mechanisms.

Prevention and tolerance mechanisms aim to provide the ability to deliver a service that can be trusted, while removal and forecasting mechanisms are applied after an incident occurred and aim to reach confidence in that ability by justifying that the functional and the dependability and security specifications are adequate and that the system is likely to meet them.

The approaches described in this report qualify mainly as prevention mechanisms, whose goal is to manage secure, dependable, and trustworthy cloud infrastructures. Some technologies, notably those related to the formal verification of security properties, serve as prerequisites to removal mechanisms and as forecasting mechanisms.

1.3 Structure

Chapter 2 contains information about the configuration data needed for the components of the TClouds integrated platform prototype. Whereas the TClouds prototype is the subject of WP 2.4 and the corresponding deliverable D2.4.1, this chapter contains an overview of all configuration data relevant for *managing* the prototype components.

Chapter 3 describes the steps involved to manage the special, tailored Cloud infrastructure components provided by WP 2.1. The goal of this management mechanism is to gather expected usage patterns from applications that want to access certain services and to translate those into a set of actions that need to be carried out in order to provide the requested service. Unfortunately, information regarding service usage is mostly specific to the respective type of service and has to be combined with knowledge about the resource consumption in terms of low-level infrastructure for each tangible service implementation to become useful. The management system described in this chapter is designed to close this gap and derive a set of actions to set up the service components with the desired properties. The actual execution of the steps is left to lower-level infrastructure management already available from most IaaS Cloud providers, except for the interactions with the setup procedure for the tailoring process of the WP 2.1 components.

Chapter 4 presents the design of several self-managed services, whose purpose is to manage the migration of resources inside and across clouds. Self-managed services provide automated capabilities and may act autonomously; such systems have also been called *autonomic systems*. They are an attractive model for building trustworthy cloud services. The chapter focuses on the security and privacy aspects of self-managed services, and furthermore on their availability and resilience features.

Chapter 5 addresses key management for cloud storage. It presents a novel approach to manage encryption keys for storage resources, such as virtual disks, which are accessed by virtual machines running at a cloud provider. Virtual machines are represented as images and stored by a cloud storage service. As demonstrated there, existing technology does not permit that an image is encrypted with a client-managed encryption key, which would prevent the provider from snooping on the image's internals. The method described in this chapter enables this by adding a key-management interface to the virtual machine hosting platform, such that the client may submit encryption keys through a security proxy exactly at the time when the decryption key for an image is needed. At any other time, the image is stored in encrypted form and the cloud provider has no access to the data in the image.

Chapter 6 describes the key management mechanisms within trusted infrastructures as developed in WP 2.1. A central management component is in charge to manage all security aspects of the managed appliances. The management system and concepts described in this chapter build on top of a Trusted Computing infrastructure and provide a seamless integration of cloud resources with on-premise resources of cloud customers.

In Chapter 7, a semantic model of IT and cloud infrastructures is developed, which represents a crucial prerequisite for their automated analysis. Using an ontology-driven approach, the chapter provides an analysis and a logical model of a large-scale IT system. The ontology differentiates between (1) the physical layer, where hardware resources are modeled, (2) the virtual layer, which includes coarse-grained virtualized resources such as otherwise available in hardware form, (3) the software layer containing the basic software components like operating systems and DBMS, and (4) the service layer, which models applications. Last but not least, a security layer (5) is also described, which is composed of non-functional properties, which is orthogonal to the four infrastructure layers.

Chapter 8 tackles the complexity of virtualized environments by formal verification of security properties. In a multi-tenant infrastructure, where resources can be attributed to multiple entities from different security domains, realizing proper isolation mechanisms between the security domains poses a challenge. The design described here proposes a high-level approach for automated discovery of potential information flow between security domains. The resulting flow graph allows for simplified verification of the isolation requirements imposed by a security policy. The chapter also reports about the development of an analysis tool and presents some results from an early case-study done on a mid-sized infrastructure of a production environment in the financial sector.

Chapter 2

Configuration Data Overview

Chapter Authors:
All contributors

2.1 Introduction

This chapter describes information about the configuration data needed for the components of the TClouds integrated platform prototype. The TClouds prototype is the subject of WP 2.4 and the corresponding deliverables. This chapter merely contains an overview of all configuration data relevant for managing the prototype components, since these data are relevant for this report, addressing the design of security management.

The prototype is structured into three broad groups of components:

1. **Trustworthy cloud infrastructure:** these components are the subject of WP 2.1.
2. **Cloud of Clouds Middleware for Adaptive Resilience:** these components are the subject of WP 2.2.
3. **Cross-layer Security and Privacy Management:** these components are the subject of WP 2.3 and further described in this report.

The rest of this chapter is structured along these component groups. For each component, we collect the management aspects of the components contributed by the partners. For each component the following questions are answered:

- Which configuration data (including cryptographic keys) is needed by the component to run?
  - Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest.
  - Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management).
  - Where should the data be kept when the instance is not running?
- Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?
- How is the component being deployed / instantiated?

- Further requirements?

For management components:

- What are they capable of managing?
- Is there a temporary / permanent uni or bidirectional connection to the managed component?
- Further requirements?

2.2 Components of WP 2.1

2.2.1 Logging as a Service (LaaS)

- *Which configuration data (including cryptographic keys) is needed by the component to run?*
  - On the LaaS:
    - URL of S3 persistent storage.
    - optionally, parameters for (Byzantine) fault tolerance, in case we replicate LaaS VM.
    - Log configuration and policy.
    - List of log users and relative keys (public keys).
    - Initial key for log integrity (A_0).
  - On each cloud component, within the LaaS library which is part of the Cloud Framework:
    - URL of LaaS VM.
    - Log configuration.
    - List of users allowed to read the logs created by the cloud component with relative keys (subset of public keys).
    - Current key for log integrity (A_i).
  - On User's management console:
    - URL of LaaS.
    - User's decryption keys.
- *Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest:*
  - Initial key for log integrity (A_0) must be kept confidential and must be confined to the LaaS component.
  - Current key for log integrity (A_i) must evolve over time on each cloud component (e.g. by using one-way functions). As soon as A_i evolves, the old one is destroyed.
  - User's decryption keys must be kept encrypted while not used. When used, they are decrypted and revealed only to the management console. Decrypted User's key must be used as little as possible and destroyed immediately after usage.

  - All the configuration of the logs, both on LaaS and on the cloud components can be public.

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*
  - A_0 in the OS of the LaaS component.
  - A_i in the Cloud Framework of each cloud component.
  - User's decryption keys on the management console.
- *Where should the data be kept when the instance is not running?*
  - A_0 must be in the OS of the LaaS component which is never stopped.
  - A_i must be in the Cloud Framework of each cloud component which is never stopped. In case of stop of the cloud component (e.g. for maintenance), A_i is destroyed and a new key is negotiated at reboot.
  - User's decryption keys are managed by the user.
- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*
  - LaaS overall configuration managed by cloud infrastructure manager.
  - each log may be configured by the user of the log. Possibly some legal constraints may be applied (e.g. minimal or maximal duration of a log).
- *How is the component being deployed / instantiated?*
  - LaaS is always running.
  - each cloud component is provided (in Dom0) with the log library.
- *Further requirements?* None.

2.2.2 Trusted Server

- *Which configuration data (including cryptographic keys) is needed by the component to run?*
  - Connection information to the Trusted Object Manager (TOM), which is managing the TrustedServer, this includes credentials for the TrustedChannel communication (confidential).
- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*
  - Host OS.
- *Where should the data be kept when the instance is not running?*
  - The data is stored on an encrypted partition on the disk of the TrustedServer. Disk encryption is protected by the TPM.

- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*
  - Cloud infrastructure manager.
- *How is the component being deployed / instantiated?*
  - When the TrustedServer is installed the data is deployed on the disk. Note that TOM uses remote attestation to ensure the Integrity of the TrustedServer.

2.2.3 Secure Block Storage (SBS)

Note: Here, we ignore the case of public clouds, in which the SBS has to be part of the client's VM, and focus on the TClouds demo where SBS is part of the cloud infrastructure!

- *Which configuration data (including cryptographic keys) is needed by the component to run?*
  - Cryptographic keys supplied by the client via the API.
  - Configuration data and credentials to assign the above mentioned keys to the corresponding client and to authenticate the client.
- *Requirements on the confidentiality of the data (e.g. for keys) during run-time/at rest?*
  - Both the configuration data/credentials and the storage keys have to be kept confidential.
- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*
  - Ideally in an isolated environment (from the hypervisor/kernel), but for implementation practicability inside a security hypervisor.
- *Where should the data be kept when the instance is not running?*
  - It can be sealed and stored on persistent storage (e.g., by TPM), such that only the legitimate, trusted hypervisor (or code in a trusted execution environment) can unseal it. Alternatively, it can be securely wiped and re-provisioned on next demand.
- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*
  - Cloud infrastructure manager has to deploy the component.
  - Provisioning of storage keys is handled by the cloud user/admin via an API in OpenStack.
- *How is the component being deployed / instantiated?*
  - When the server/hypervisor is set up, the component is deployed and instantiated.

2.2.4 Secure VM Images

Identically to SBS, with the only difference that cloud users/admins can provision/retrieve encrypted VM images (via new interfaces in OpenStack). Images are decrypted on-the-fly, based on the SBS component, during image execution. The extension to SBS for that purpose has to be tightly integrated in the secure hypervisor or must be a highly efficient (in terms of performance) secure environment.

2.2.5 High Availability Basics

Same as the one described in section 2.4.1

2.2.6 Resource-efficient BFT (CheapBFT)

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Secret keys and tamper-proof counter state for signing messages on the FPGA board.
  2. Addresses, ports, and keys of other replicas in the cluster.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime/at rest?*

  1. Signing keys must stay confidential, either by shipping pre-configured FPGA boards to the cloud provider, or going through a setup procedure in a trustworthy environment to install the keys.

- *Where is the data deployed (during runtime of the component)*

  1. Location of other hosts in the cluster stored inside the VM.
  2. Secret keys are deployed on the FPGA boards.

- *Where should the data be kept when the instance is not running?*

  Secret keys and counters must stay on the FPGA boards. Addresses of other replicas statically stored on the disk image for the VM or configured at startup.

- *Who is responsible for the configuration?*

  1. User of the service generates and installs keys.
  2. Static placement by user or management infrastructure responsible for dynamic allocation of replicas.

2.2.7 Simple Key/Value Store

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Access policies (based on IP address, shared secret, . . .).
  2. Storage size.
  3. Feature subset configuration: Encryption, persistent storage, synchronization.

4. Addresses, ports, and keys of other replicas in the cluster.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime/at rest?*
  1. Shared secrets and encryption keys must be transmitted securely.
  2. Cloud provider should not be able to intercept/steal secrets from the VM.
- *Where is the data deployed (during runtime of the component)*
  1. Inside the VM.
- *Where should the data be kept when the instance is not running?*

  Depends on the persistence level required. If the key/value store is merely used as a cache, there is usually no need to keep any data. Otherwise the configuration should be kept in the management layer.
- *Who is responsible for the configuration?*
  1. User of the service configures access policies, keys and required features.
  2. Management infrastructure responsible for coordination of related hosts.

2.3 Components of WP 2.2

2.3.1 Storage Cloud-of-clouds

The system accesses n cloud providers assuming that at most f of them can be malicious. The idea is to trust none of them completely.

The component runs at the client side (not on the cloud).

- *Which configuration data (including cryptographic keys) is needed by the component to run?*
  - Credentials for the n cloud providers (confidential).
  - It is also necessary to define the access control for reading and writing data units, but it's still an open problem how to configure several clouds using the same policy in a simple way.
- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*
  - At the client process accessing the storage system.
- *Where should the data be kept when the instance is not running?*
  - In a digital wallet that can be accessed by the client process when using the storage service. Another option is to give a password to the system user that can be used to derive all these credentials and access the system.
- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*

  - The cloud customer setting up the system.

- *How is the component being deployed / instantiated?*

  - It is an infrastructure service that can run at cloud custumers side or inside of a trusted cloud to make use of a set of unstrusted storage clouds for its storage services.

We think there are important research questions about the configuration of a cloud-of-clouds service (too many third-party services to configure consistently) that need to be answered.

2.3.2 Confidentiality Proxy for S3

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  - Credentials for the S3 provider (confidential).
  - Encryption key for the TVD (confidential).
  - Mount information: mapping of S3 file system on host OS to shared folder in the VM (public).

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*

  - Host OS.

- *Where should the data be kept when the instance is not running?*

  - Management component (TOM).

- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*

  - Cloud customer administrator: S3 credentials, mount point inside the VM.
  - TOM: TVD key, mount point host.

- *How is the component being deployed / instantiated?*

  - The component has a service component on the host (TrustedServer) and a management component inside of TOM. Communication between both is across the TrustedChannel. When a VM is started via TOM on the TrustedServer the configuration is transferred via the TrustedChannel to the TrustedServer and handled accordingly by the service component.

2.3.3 Fault-tolerant Workflow Execution (*flexible* FT-BPEL)

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Zookeeper instance to use for service coordination.
  2. BPEL engine scripts/settings.
  3. Auxiliary web service addresses.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime/at rest?*

  1. Connecting to Zookeeper may be protected with a shared secret key that must not leak to any 3rd party.
  2. The BPEL engine scripts may contain additional secret information, depending on the application scenario.

- *Where is the data deployed (during runtime of the component)*

  1. Inside the VM.

- *Where should the data be kept when the instance is not running?*

  Basic configuration data is saved on the virtual disk images from which the enhanced BPEL engine is started from. Other information that may change (i.e. Zookeeper instances, scripts) could be supplied by the user or the cloud management infrastructure on startup.

- *Who is responsible for the configuration?*

  1. User of the service provides all necessary configuration, but may rely on information from the management infrastructure.

2.3.4 Extensible Coordination Service (*flexible* Zookeeper)

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Access policies by IP address and/or shared secret.
  2. Addresses, ports, and keys of other Zookeeper nodes.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime/at rest?*

  1. Shared secrets and access policies must be transmitted and installed in a secure fashion.
  2. Cloud provider should not be able to intercept/steal secrets from the VM.

- *Where is the data deployed (during runtime of the component)*

  1. Inside the VM.

- *Where should the data be kept when the instance is not running?*

  The configuration data is stored on the disk image from which Zookeeper is supposed to run or set by the user/service provider immediately before starting the instance.

- *Who is responsible for the configuration?*

  1. User of the service configures access policies and secrets.
  2. Static configuration of Zookeeper nodes by user or cloud service provider.

2.3.5 State Machine Replication as a Cloud-of-cloud PaaS

The configuration bellow assumes a solution without trusted components, for being deployed on untrusted clouds.

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Addresses of each of the n replicas.
  2. Each replica requires at least one text-based configuration file.
  3. Each of the n replicas and clients should have a unique private key with them.
  4. Every client and replica needs to have access to the public-keys of all replicas.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime/at rest?*

  1. All private keys should be maintained confidential.

- *Where is the data deployed (during runtime of the component)*

  1. On the hosts where the replicas are running.

- *Where should the data be kept when the instance is not running?*

  Inside of the virtual machine image. Maybe protected in a digital wallet that can only be oppened with a password or key presented by the administrator during system activation.

- *Who is responsible for the configuration?*

  1. Owner of the service deployed on the cloud.

2.4 Components of WP 2.3

2.4.1 Access Control

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Cryptographic keys.
  2. Access right policy (as defined by cloud user).
  3. Infrastructure policy and properties as defined by cloud architects.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest?*

  1. The keys must be securely protected; even cloud internal employees should not be capable of accessing the keys.
  2. The integrity of access rights' policy should be protected.

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*

  Access rights will be enforced at three levels:

  1. Inside the VM, when processed by business logic application.
  2. Any action on VM must be validated. Such validation is part of OpenStack.
  3. Outside the cloud at client workstation after downloading data from the cloud.

- *Where should the data be kept when the instance is not running?*

  The data should be kept somewhere secure in the cloud (e.g. encrypted inside a storage accessible to the VM).

- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*

  This can be split at three parts:

  1. Cloud user will define application access rights.
  2. Cloud user's customer will also define access rights related to his data.
  3. Cloud admin will define infrastructure policy and properties.

  We will build automated services that enforces such access rights at the levels defined above.

- *What are you capable of managing?*

  Access Rights on VM level (to be built part of OpenStack), and on Data level both when accessed inside the cloud and at outside the cloud at client devices).

- *Is there a temporary/permanent uni or bidirectional connection to the managed component?*

  A bidirectional connection with the managed component.

2.4.2 Self-Managed Support Services

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. Cryptographic keys.

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest?*

  1. The keys must be securely protected; even cloud internal employees should not be capable of accessing the keys.

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*

  Inside the VM.

- *Where should the data be kept when the instance is not running?*

  The data should be kept in the VM.

- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*

  Cloud user will define access rights

- *What are you capable of managing?*

  Running processes inside a VM.

- *Is there a temporary/permanent uni or bidirectional connection to the managed component?*

  Bidirectional connection with the managed component.

2.4.3 Ontology-based Reasoner to Check TVD Isolation

- *Which configuration data (including cryptographic keys) is needed by the component to run?*

  1. The core ontology (classes and properties).
  2. The system model (i.e. individuals in the ontology).

- *Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest?*

  1. The core ontology is public.
  2. The system model is private (known by the Cloud Provider/Infrastructure).

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*

  1. The core ontology is built-in in the component.
  2. The system model is dynamically acquired from the infrastructure components (e.g. via queries to `libvirt`).

  Note: we assume that the component is always running.

- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*

  1. We assume that, for the demo, the configuration is fixed and decided a-priori. We do not expect that any entity can change the configuration, i.e. the core ontology. More generally:
     (a) The cloud infrastructure manager may modify (parts of) the core ontology, for instance to specify default policies/assumptions (e.g., he can decide wether by default a L2 link is considered secure or not).

(b) Similarly, the cloud customer admin may request analysis with slightly different models.

- *What are you capable of managing?*

  1. For the demo, check isolation of TVDs. In more detail, we extend the deployment of a new VM by setting up secure channels among physical nodes, such that the communications intra-TVD that go over the physical network result protected. More in general, i.e. other than the demo, we are capable of checking security properties based on ontology reasoning (e.g., verify if two components can securely communicate, list possible threats to an asset . . .).

- *Is there a temporary/permanent uni or bidirectional connection to the managed component?*

  1. No network connections. This component will be integrated in the management component as a library/plugin.

2.4.4 Trusted Objects Manager (TOM)

- *What are you capable of managing?*
  - TrustedServer (managing VMs and TVDs).
  - Secure S3 proxy.
  - Other appliances provided by Sirrix (TrustedVPN, TrustedDesktop, etc.).
  - Other TClouds components, if necessary.
- *Is there a temporary / permanent uni or bidirectional connection to the managed component?*
  - Yes. The default is a permanent bidirectional connection. But depending on the appliance this can be customiized.

2.4.5 Trusted Management Channel

See Sections 2.4.4 and 2.2.2.

2.4.6 Automated Audit System

- *Which configuration data (including cryptographic keys) is needed by the component to run?*
  - Discovery data of cloud infrastructure.
  - Policy specification.
- *Requirements on the confidentiality of the data (e.g. for keys) during runtime / at rest?*
  - Discovery data should not be accessible by any cloud consumer.

- *Where is the data deployed (during runtime of the component), e.g. inside the VM, inside the OS / Hypervisor, in the Cloud Framework (e.g. OpenStack Management)?*
  - Depending on deployment strategy: Either inside VM or in the Cloud Framework.
- *Where should the data be kept when the instance is not running?*
  - Not necessary. Discovery data can be obtained again by performing the discovery, and policy specification is given for each audit.
- *Who is responsible for the configuration: cloud infrastructure manager, cloud customer admin, cloud user?*
  - Cloud infrastructure manager should provide mechanism to obtain discovery data.
  - Cloud customer admin can validate that customers are correctly isolated from each other.
  - Optionally, the cloud user can validate the isolation of his part of the infrastructure.
- *How is the component being deployed / instantiated?*
  - The audit tool can either be deployed inside a VM, which then receives commands from the central management API server, or is closely integrated with the management API server.
- *What are you capable of managing?*
  - We manage the correct isolation of tenants in a cloud infrastructure by performing audits of the infrastructure against an isolation policy.
- *Is there a temporary/permanent uni or bidirectional connection to the managed component?*
  - Temporary bidirectional connection: Audit API call will audit the infrastructure with regard to a given policy specification and will determine policy compliance or violation. A report will be provided in case of policy violation.
- *Further requirements?*
  - None.

Chapter 3

Management of Tailored Cloud Components

Chapter Authors:
Johannes Behl, Rüdiger Kapitza, Klaus Stengel (FAU)

3.1 Introduction

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

First of all, the interfaces for these kinds of services are hardly standardized, which makes it difficult to move applications to different cloud providers (i.e. "Vendor Lock-in"). Additionally, it's very hard to judge the security and reliability implications of using such a service, because the implementation details and information about the hosting environment are usually entirely opaque. The user often has to rely solely on the assertions made by the cloud provider.

On the part of the cloud provider, offering those services require additional infrastructure for their management. As those services are also separately billed on a pay-as-you-use basis, they require close monitoring of each user's actions. Only this kind of tracing allows for fair accounting, because the individual functions offered by a service are used at varying frequencies and put different strain on the hardware components. On top of that, it's also the provider's responsibility to decide, how much of the available physical hardware should be dedicated to the IaaS or the miscellaneous service offers.

Due to the obstacles explained in the previous paragraphs, we want to experiment with a different approach for providing those services. The basic idea is to leverage the existing IaaS platform to run the necessary software components, either by the cloud provider or the customer that wants to use the service. The required components are just provided like any other virtual machine image at the provider. This has the following advantages:

- For the **cloud user** the whole process required to operate the service becomes transparent to the user. All data is guaranteed to be stored in separated virtual machines, thus pro-

vides better isolation from other customers. It becomes possible to transfer the images to other cloud providers and run the same service with the same programming interfaces at different providers without any porting effort.

- From the **provider**'s perspective, there is no longer any need to monitor the behaviour of the customer's applications, because the billing process can simply happen on a per-VM basis. All physical hardware can be managed by the same IaaS platform and can provide the additional services on demand.

In order to make this work, there are some additional measures required to ensure that the components do not cause increased costs by unnecessarily occupying resources on the IaaS level. To meet this goal, we first reduce the constant costs of running such services by removing most overhead in a special tailoring process, which is the topic of a corresponding component of WP 2.1. Furthermore, we also need a way to continously capture the resource requirements of an application in order to estimate and provision the required infrastructure, which is the topic that will be covered here.

The rest of this chapter is organized as follows: We start with a short explanation of how the tailored cloud services we proposed in WP 2.1 operate. In the next section we give a short summary of the required management tasks involved and what the main challenges are. Finally, we propose a possible way to achieve a fully automated solution.

3.2 Tailored Cloud Services

In WP 2.1 a method to strip down simple web service Applications to run on commodity cloud infrastructure without much overhead is proposed. The basic procedure for using such a service involves the following steps:

1. The application is packaged together with a generic runtime environment and distributed in form of an VM image to the cloud provider(s) the service is supposed to be hosted at.

2. When starting the virtual machine, the common runtime system gathers information about the cloud platform it is running on and waits for additional parameters for the tailoring process.

3. When the all required parameters for the service instance are available, the tailoring process begins and produces program code that is optimized for the execution environment and the specified usage parameters.

4. The common runtime is replaced with the service program from the previous step and provides the requested service.

The whole process is designed to eliminate as much unnecessary parts from a web service as possible. It is common practice nowadays to run an entire Java VM, based on a commodity OS, e.g. Linux, on the virtual machines hosted at the cloud provider. This is very wasteful, because it requires about 150 MB of RAM just for the basic execution environment. Instead, we statically link the application with a runtime system specially designed to run on top of virtual machines, and remove any parts of the runtime not necessary for the particular web service. Our tailoring solution is expected to reduce the required overhead to a few megabytes at most. This allows the memory otherwise consumed by unnecessary parts of the runtime system to be used for the

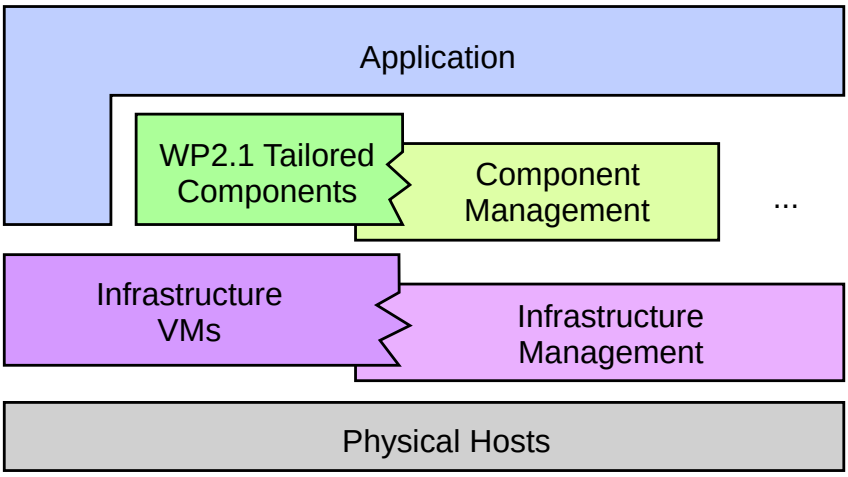


Figure 3.1: Tailored components and IaaS architecture

web service itself. So it becomes possible to use smaller and cheaper VM instances to store the same amount of data compared to the systems currently in use.

Besides the improvements regarding resource consumption, we also expect an improvement in the reliability aspects of the resulting service; Since the tailored service contains less layers of indirection in the software stack and the overall amount of active code in the system is much smaller, there are less possibilities for running into misbehaving code. The tailoring aspect also adds a certain amount of variability to the system, which means that a dedicated attacker who tried to exploit a certain error in the runtime system has less chances to succeed on multiple instances of the service at once.

Even if an instance of a tailored service was compromised, the attacker is still confined to the virtual machine environment. As outlined in the previous chapter, each VM is ideally dedicated to a single customer and therefore is expected to providing better isolation than hosting multiple instances on the same commodity operating system. This is supported by the fact that the interface to the VM environment has much fewer functions than common operating systems and thus is generally easier to secure.

As outlined in the required setup procedure, the tailoring aspects require knowledge of certain parameters about the environment in which the final service should be executed. While the VM execution environment can be examined by the runtime system on its own, all other parameters must be specified externally. This leads to the next section and main topic of this chapter, where the challenges regarding the deployment and administration of such services are analyzed and discussed.

3.3 Administrative Task Challenges

The primary issues that need to be addressed, in order to make these tailored components as easy to use as the services already offered by commodity cloud providers, are of administrative nature. While the existing services can just be used as required and the cloud provider is entirely responsible for the management aspects, they now become transparent to the user.

In fact, the instantiation of the tailored service components may even require more effort by the administrator compared to traditional operating systems, caused by the parameters necessary for the tailoring process. This is generally undesirable in a large scale cloud scenario, where

everything has to be automated as much as possible to stay manageable.

Moreover, we also have to deal with dynamic changes regarding the utilization of VM resources. If we run the component on top of an ordinary IaaS platform, any unused capacities occupied by the VM hosting the service can't be reused by other VMs. This increases costs, either because the user has to pay more than necessary for his VM instances, or because the cloud provider is no longer able to reach optimal utilization of the available hardware resources. While there are known techniques, like memory balooning or CPU hotplugging, which could be used for dynamic reallocation of resources, these aren't widely supported across common cloud providers. Therefore it's currently not possible to rely on these mechanisms, especially in case of multi-cloud scenarios, so we also need alternative ways to provide scalability.

Therefore we need to investigate how to eliminate the administrative overhead by looking at each involved party and provide an algorithm to make the setup and scaling process fully automatic from a user's viewpoint. The layer overview in Figure 3.1 shows the general architecture of an application using our tailored components in the cloud. On the very bottom of the architecture are the physical hosts which represent the necessary hardware to run the virtual infrastructure. The (pink colored) Virtual Layer above the physical hosts essentially consists of two parts: The virtual machines that are provided to the customer of the IaaS-cloud, and the closely linked facilities for managing the infrastructure. Our tailored cloud components (depicted in green) introduced in the previous section run on top of virtual machines, where usually the application layer is located. This means from an architecture point of view, that the tailored components provide an additional layer between the raw virtual machines and the application. However, this layering is not strict, as it is reasonable to assume that the end-user applications are also hosted on top of the same Virtual Layer, albeit in different virtual machine instances from the tailored components. Finally, the management of the taiored components is best placed between the application, the management of the virtual infrastructure and the tailored component. This way, it has access to all relevant actors involved in setting up and providing the service.

3.4 Automated Management

In the previous section we examined the required administrative tasks and the architecture where the corresponding management component will be embedded. It is important to note that the applications wanting to use the service components are the driving force behind all resource requirements. In order to provide a solution for automatically managing the service components, we need to have access to usage statistics from the application layer. This allows our solution to perform near-term predictions and plan the necessary changes at the lower layers. We provide a small library that must be linked to each application and allows the programmer to specify, how much demand for certain resources offered by the service currently exists.

One key aspect to keep this approach simple from an application developer's perspective is, that every resource is expressed in terms of the application. For example, in a database scenario, the required storage capacity is always counted in terms of records in the database instead of megabytes. This leads to a good seperation of concerns, as the additional memory requirements for indexing and backup data often depend on the configuration of the database, which is something the application should not be necessarily aware of. As a consequence, a major part of the management for the tailored components consists of translating the resource requirements between the application and the meaning in terms of raw infrastructure resources.

Describing resource requirements and performance properties isn't an entirely new prob-

lem, as similar techniques are already employed when dealing with machine-readable Service Level Agreements (SLAs), like the WSLA Specification [LKKF03] does. Unfortunately, those concentrate more on the business and monitoring aspects instead of automated deployment [BPA11, EdPG+08]. Although it is possible to trigger certain actions in case the service no longer delivers satisfactory performance, it is unable to accommodate the required tailoring process. Research efforts from autonomic computing initiatives [LML05] are more focused on the technical aspects, but have a more holistic approach, where the global hard- and software interdependencies are modeled [TBDP+06]. This is useful for making high-level decisions and provide strategies for resource allocation, but they still do not provide a clear way to derive parameters for the tailoring of service components. The same argument applies to the self-managed infrastructure described in section 4.

Therefore, our solution is to keep a strictly layered model, deriving the requirements for the tailored services directly from the application layer. Only the translated infrastructure resource requirements are fed into the generic cloud managment to use the resoucre allocation and migration features of the underlying infrastructure. This way, we can re-use the existing techniques to provide the necessary resource allocation and reconfiguration services. Our work concentrates on the automated tailoring and resource dependencies. We only track the application-specific usage and decide how to configure the tailored services accordingly, while leaving the resource allocation strategies to the infrastructure layer.

3.5 Summary

As outlined in the previous sections, the software components currently offered by cloud providers have several restrictions, especially in cloud-of-cloud scenarios, where portiability between different providers is am essential property. The alternative solution of hosting the desired components on common virtual machines can improve the situation, but current commodity software wastes expensive resources and has more administrative overhead. Fortunately, the resource issues can be addressed by tailoring the VM images to contain only the minimal set of components to run the service, as proposed in WP2.1. The administrative overhead can be eliminated by connecting the service closer with the application in order to manage the provisioning of service instances automatically. With some small additional effort invested in the development of applications, we expect to get the same level of automatism as from the native services of the cloud-providers, but without any vendor lock-in effects.

Chapter 4

Migration of Resources Inside and Across Clouds

Chapter Author:
Imad M. Abbadi (OXFD)

4.1 Introduction

We propose the usage of self-managed services to manage the migration of resources inside and across Clouds. Self-managed services are software services which provide Cloud computing environment with automated capabilities, e.g. manage resource's availability, reliability, and resilience inside a Cloud and across Clouds-of-Clouds. Such automation of management is based on many static and dynamic factors including Cloud user properties and Cloud infrastructure properties [Abb11a]. Our main interest is in providing security and privacy by design for availability and resilience self-managed services. We now define these services, details of which can be found in ([Abb11b]).

Resilience — Figure 4.1 provides a conceptual model of *Resilience* function. This function resembles system administrators, who *Deploys* a *Resilient Design* at virtual layer. The *Resilient Design* is provided by System Architect process at two cases: the first is when producing architecture for a new service request, and the second when updating an already exiting architecture. *Resilience* function communicates with other resources (e.g. physical servers' VMM) and/or other management tools (e.g. Virtual Control Centre [Abb11a]) to *Deploy* the *Resilient Design*. The resilience function is also in charge of communicating failures of a resource to other services. This is by *Triggering Cascaded Actions*; e.g. on failure it might trigger the *Availability* function to divert traffic to other available routes. The *Resilience* function should also *Maintain* security and privacy by design, e.g. *Resilience* function should consider the hosting of virtual resources at physical domains which are not geographically located within boundaries restricted by the user properties.

Availability — The *Availability* function is in charge of i) maintaining communication channels of available virtual services with resources at application layer and ii) distributing application layer requests evenly across available redundant virtual resources. Availability is supported by a correctly *Deployed Resilient Design*. The higher resilient a system the higher availability/reliability would be expected. Figure 4.2 provides a conceptual model for application *Availability* service. This Figure provides examples of *Incidents* from *Resilience* and *Changes* from *Scalability* service, which Triggers the *Availability* service. The *Availability* service in turn Performs *Actions* based on the *Incidents* and *Changes*. The *Actions* also Trigger *Cascaded Actions*

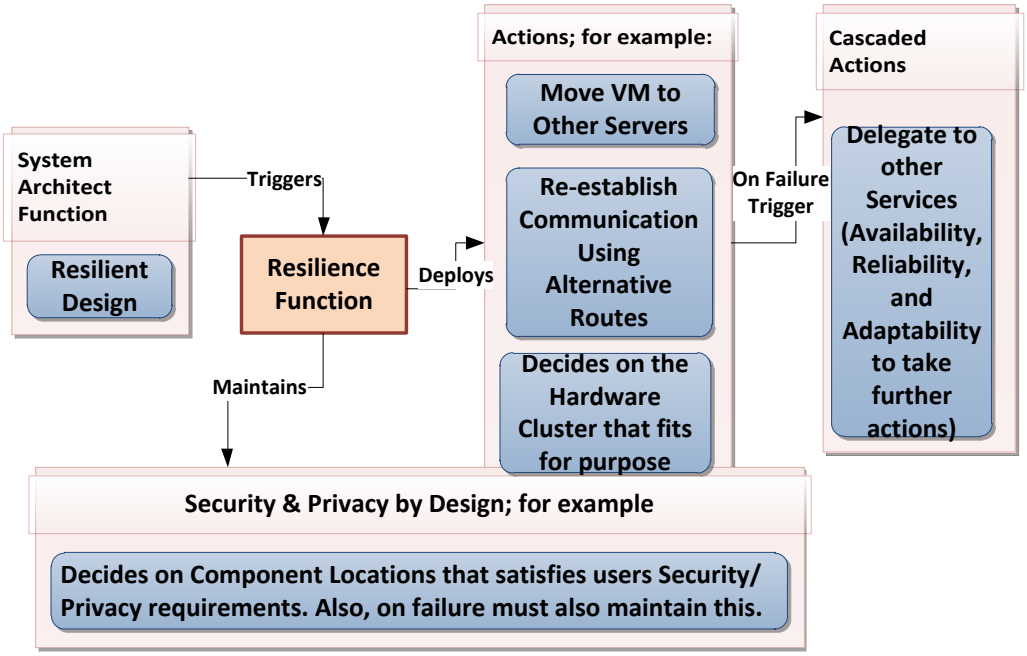


Figure 4.1: Resilience Functions

to other services at both *Application Layer* and *Virtual Layer*. For example, if a channel is marked unusable by the *Resilience* function, the availability process immediately stops diverting traffic to that channel, and re-diverts the channel traffic to other active channels until the *Adaptability* function fixes the problem. Availability function should also consider security and privacy requirement by design. For example, it should maintain secure communication channels when distributing load, verifies the identity of communicating parties, and communicates securely with other services.

In order to provide privacy and security by design for the above management services we need first to understand Cloud infrastructure taxonomy. In the next section we start by providing a brief description of the Cloud taxonomy, we then discuss what aspects of services we focus on for the purpose of the demonstration.

4.2 Cloud Management

In this section we briefly outline Cloud taxonomy and infrastructure management.

4.2.1 Cloud Infrastructure Taxonomy Overview

In this section we outline the taxonomy of the Cloud infrastructure, detailed discussion of which can be found in previous work [Abb11a]. A Cloud infrastructure is analogous to a 3-D cylinder, which can be sliced horizontally and/or vertically (see Figure 4.3). We refer to each slice using the keyword “layer”. A layer represents Cloud’s resources that share common characteristics. Layering concept helps in understanding the relations and interactions amongst Cloud resources. We use the nature of the resource (i.e. physical, virtual, or application) as the key characteristic for horizontal slicing of the Cloud. For vertical slicing, on the other hand, we use

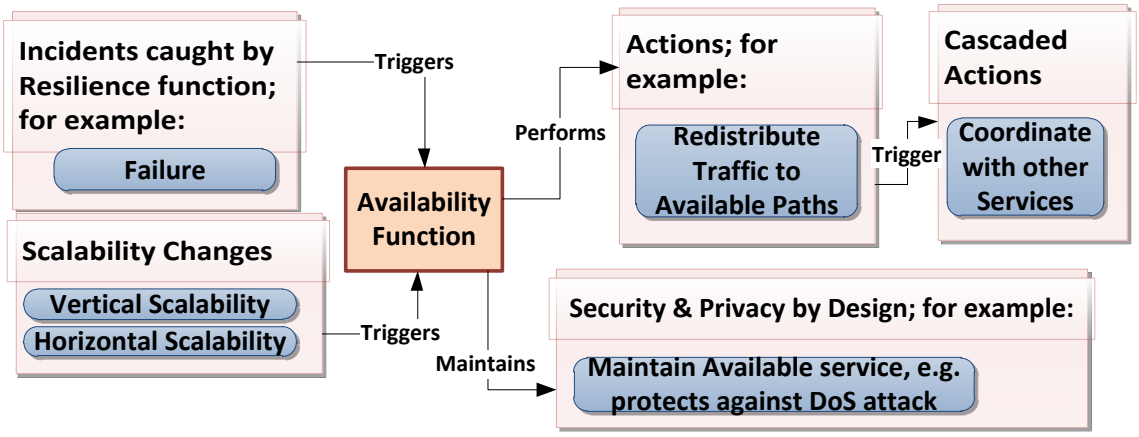


Figure 4.2: Availability Function

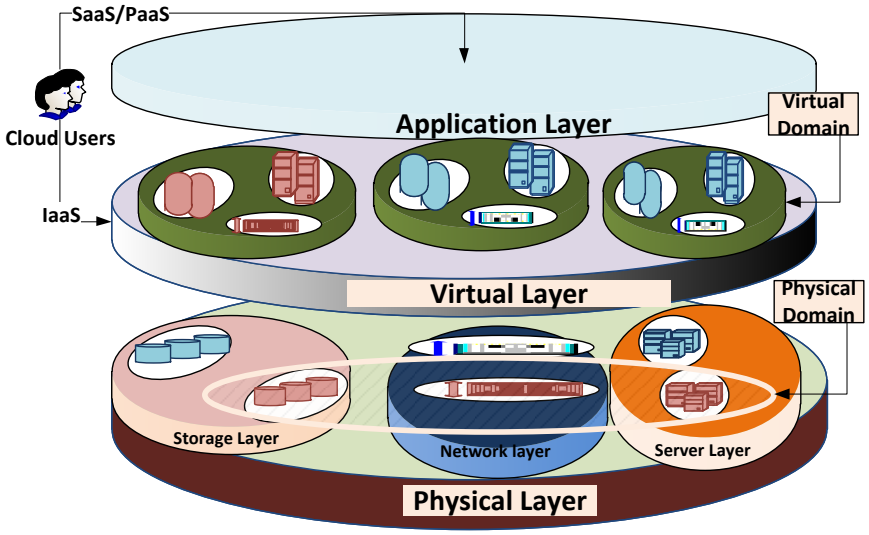


Figure 4.3: Cloud 3-D View

the function of the resource (i.e. server, network, or storage) as the key characteristic for vertical slicing. Based on these key characteristics a Cloud is composed of three horizontal layers (Physical Layer, Virtual Layer, and Application Layer) and three vertical layers (Server Layer, Network Layer, and Storage Layer).

In the context of this report we mainly focus on *Horizontal Layer*. We identify a *Horizontal Layer* to be the parent of physical, virtual or application layers. Each *Horizontal Layer* contains *Domains*; i.e. we have Physical Domains, Virtual Domains and Application Domains. A *Domain* represents related resources which enforce a *Domain* defined policy. *Domains* at physical layer are related to Cloud infrastructure and, therefore, are associated with infrastructure properties and policies. *Domains* at virtual and application layers are Cloud user specific and therefore are associated with Cloud user properties. *Domains* that need to interact amongst themselves within a horizontal layer join a *Collaborating Domain*, which controls the interaction among members using a defined policy.

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

4.2.2 Virtual Control Centre

In this section we outline part of Cloud's virtual resource management, detailed discussion of which can be found in previous work [Abb11a, Abb11c]. Currently there are many tools for managing Cloud's virtual resources, e.g. vCenter [VMw10] and OpenStack [Ope10]. For convenience we call such tools using a common name Virtual Control Center (VCC), which is a Cloud specific device that manages virtual resources and their interactions with physical resources using a set of software agents. Currently available VCC software agents have many security vulnerabilities and only provide limited automated management services.

VCC manages the infrastructure by establishes communication channels with physical servers to manage Cloud's Virtual Machines (VMs). VCC establishes such channels by communicating with Virtual Machine Manager (VMM) running on each server. Such management helps in maintaining the agreed SLA and Quality of Service (QoS) with customers. VCC will play a major role in providing Cloud's automated self-managed services. We now summarize the factors which affect decisions made by VCC services.

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

User Properties — These are as follows.

- **Technical Requirements** — In IaaS the organization enterprise architects team would provide an architecture for the outsourced infrastructure based on application requirements. This includes VMs, storage and network specifications. For

example, enterprise architects could provide the properties of outsourced applications, e.g. DBMS instances that require high availability with no single point of failure, middle-tier web servers that can tolerate failures, and the application nature is highly computational. Realizing this would enable Cloud providers to identify the best resources that can meet user requirements [Abb11a].

- **Service Level Agreement (SLA)** — SLA specifies quality control measures and other legal and operational requirements. For example, these define system availability, reliability, scalability (in upper/lower bound limits), and performance metrics.
- **User-Centric Security and Privacy Requirements** — Examples of these include (i.) users need stringent assurance that their data is not being abused or leaked; (ii.) users need to be assured that Cloud providers properly isolate VMs that run in the same physical platform from each other (i.e. problems of multi-tenant architecture [RTSS09a]); and (iii.) users might need to enforce geographical location restrictions on the processing and storage of their data.

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

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

4.3 Security and Privacy by Design

As we discussed earlier Cloud infrastructure is structured into groups of resources (i.e. Physical Domains). Each is associated with a set of properties (i.e. infrastructure properties) which enable the Physical Domain to address part of User Properties. In other words a user virtual domain is associated with user properties and is hosted at selected physical domain. The physical domain selection should be based on its properties to be capable of addressing user properties. This concept shows that user resources do not move randomly in the Cloud. In this section we discuss an important part in this direction, which is about providing an automated management tool which enable the control of resource movements within a Cloud infrastructure (WP 2.1) and across Cloud-of-Cloud infrastructure (WP 2.2). This will be a key element of the TClouds demonstration later.

(Assumption: We assume that Cloud infrastructure is monitored using CCTV recording, and the physical infrastructure is physically protected against attacks, e.g. moving a server between locations.)

Single Domain Management — Resources within a single physical domain are typically located within physical proximity close to a shared storage. All physical servers have access to a shared pool of storage servers where virtual machine images (VMI) are stored. The physical domain should be associated with a policy defining each VM primary hosting server and backup hosting servers. Within a Physical Domain a machine by default start

on its specifically allocated primary hosting server. If the VM primary hosting server fails (for any reason) to provide the services requested by the VM the VM would then automatically be relocated to any of the backup hosting servers which has enough resources to serve the VM. The allocation of VMs to primary and backup servers should be done automatically based on VM properties and physical servers' properties by the VCC without human intervention.

Multiple Domain Management — Resources within a single domain might need to be relocated (e.g. migration process) to another physical domain for several reasons. For example, i) source physical domain has physical failure which affects its properties, ii) a physical domain is overloaded with resources which raise a need to move some resources to a new domain, or iii) a virtual resource properties change which results in a need to migrate it to new domain that satisfy the new needs.

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

This can be managed by in a similar way to the management of resources migration between physical domains within a single Cloud, as discussed earlier. The main difference user requirements would need to be validated to ensure that the user agrees on that.

Chapter 5

Key Management and Secure Storage

Chapter Authors:
Sven Bugiel, Stefan Nürnberger (TUDA)

In this chapter, we elaborate on the issue of secure and trustworthy key management within cloud infrastructures with respect to untrusted cloud service providers. We describe our approach to mitigate this issue, based on existing Trusted Computing concepts and technology, and to establish secure block storage based on the secure key management.

5.1 Trusted Computing Basics

In our design we make use of standard concepts of Trusted Computing. Readers already familiar with this topic may safely skip this section.

The TPM. The most prominent approach to Trusted Computing technology has been specified by the Trusted Computing Group (TCG, [tcg]). The cornerstone of TCG Trusted Computing is a hardware extension, called *Trusted Platform Module* (TPM, [Tru11]), which acts as a hardware trust anchor and enables the integrity measurement of the platform's software stack, the secure reporting of these measurements to a remote party (*remote attestation*), and secure storage that can be bound to the platform state (*sealing*).

Attestation. The platform state is measured in form of cryptographic hashes (SHA1) of all executable content in the system. The hashes are stored in a secure memory location of the TPM, called *Platform Configuration Register* (PCR), such that the stored values and their ordering are reflected by the PCRs (*extension*). To securely attest the PCR values to a remote party, the TPM specifications introduce a special attestation key type (*Attestation Identity Key, AIK*), which can only be used from within the TPM in order to sign PCR values, but not for externally supplied data. Together with a certificate that verifies that an attestation key belongs to a benign TPM, a remote party is able to verify the authenticity and integrity of the reported platform state and thus judge the platform's trustworthiness. Additionally, the TPM provides common RSA-2048 signing and encryption of externally supplied data.

Secure Execution. One of the latest additions of Trusted Computing in conjunction with modern processors, is the ability to run software in a state isolated from the rest of the system with an extremely minimalistic trusted computing base (TCB). Secure Execution Environments (SEE) can be seen as a separation of two isolated modes of the processor and other hardware. When switching to the secure execution environment, no assumptions have to be made about software

running prior to the execution of the secure environment. The hardware further assures, that external access (e.g. hardware DMA requests) cannot access the code and data running in the SEE. Additionally, before running code in the SEE, the processor extends a PCR so that the exact code that runs in the SEE can be attested to a third party.

Keys. TPM-generated keys form a key hierarchy, with the so-called *Storage Root Key, SRK* at the top. While this root key never leaves the TPM, all other keys can be stored on untrusted storage, because their private key portion is encrypted with their parent key. A specific feature of the TPM is, that both TPM-generated keys and TPM-encrypted data can be bound to a platform state, i.e., a state in which the key is usable or the data can be decrypted, respectively. The latter one is denoted as *sealing*. Moreover, keys and seal data structure contain information about the platform state in which they were created.

5.2 Classical Cryptography in the Cloud

Despite the well-known benefits of cloud computing, concerns about information security arise, as customers currently have to treat the cloud as an unobservable black box which forces them to blindly trust the provider.

Currently, Service Level Agreements (SLA) only shallowly deal with security objectives (e.g. data isolation by access control). Even worse, there is no guarantee that these objectives are indeed fulfilled. When handling sensitive data in the cloud, the client has to trust the cloud provider not to eavesdrop on the data being processed inside a running Virtual Machine Instance (VMI). Usually, this trust is widened to the fact that the provider does not eavesdrop on data incorporated in saved Virtual Machine Templates (VMT) or on external storage and that the provider wipes all the hard disks after they broke down. As a countermeasure, one could use cryptography to accomplish confidentiality. In order to process data, it has to be decrypted at some point. Plaintext data that is processed inside the VMIs is then temporarily stored in the VMI's memory. One possibility to ensure confidentiality nevertheless, would be to perform all operations under encryption (e.g. to use Fully Homomorphic Encryption (FHE) [Gen09]), which however, is currently not yet efficient in practice. Instead, we build a reasonable and practical solution that hides the cryptographic keys from the VMs and the cloud provider. However, as plaintext is processed in the VMI's memory, we assume that nobody but the client can get access to the VMI's memory, i.e. cloud administrators have only remote access with limited privileges. These implications are schematically depicted in Figure 5.1.

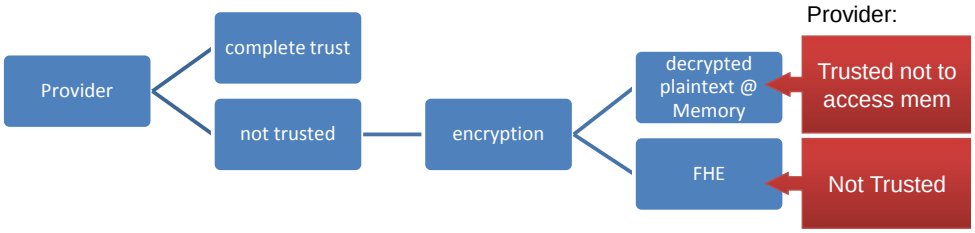


Figure 5.1: Schematic Trust Models and Their Consequences.

5.2.1 Keys in VMs

The reason yet not to store keys in the VM's memory in the first place, is the fact that one of the main benefits of cloud computing is running multiple instances of the same VMT. Controlling the use of the same key in different instances or managing revocations is hard or even impossible. Further, provisioning this key is only possible by placing it in the VMT or by sending it to the VMI over a trusted channel. For establishing a trusted channel in turn, a secret key already in place in the VMT is needed for authentication. Putting the key in a VMT is not an option, as it would imply trusting the external storage where the VMT is saved. Such storage is easier accessible than VMI's memory and faces the risk of lost control once the physical hard drives are replaced. Furthermore, keys stored in a VM impede the concept of VMT sharing and the needed cryptography needs to be implemented and managed by the cloud's client. The different alternatives of storing keys and their implications are depicted in Figure 5.2.

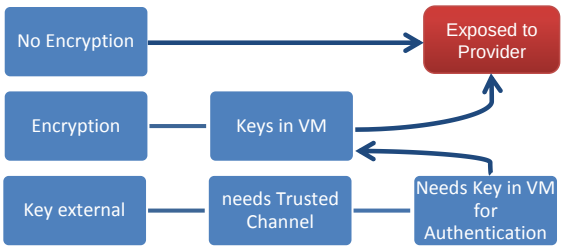


Figure 5.2: Position of Key (Management) and its implication

In contrast, a central key management within the cloud provides the possibility to audit the key usage and provides verifiable security for legacy VMs. Hiding the cryptographic key (especially from the cloud provider) is crucial, as only a central key management guarantees control over key usage. This means key management shall take place in the cloud, while the key itself must never be exposed to anybody but the client. We focus on a practical scenario that limits the needed trust in the cloud provider by leveraging existing Trusted Computing approaches in order to hide cryptographic keys from adversaries, including the cloud provider itself.

5.2.2 Our Approach

We adopt and combine the concepts of Secure Execution Environments with whitebox cryptography in order to build a trust anchor for the key management in the cloud. With the concept of *Binding Keys* borrowed from Trusted Computing, it is possible to provision secrets to the cloud in an encrypted state. The decryption of that secret is only possible by a secure hardware chip, the *Trusted Platform Module* (TPM). The TPM ensures that access to the secret is only granted for the creation of publicly known, user-defined whitebox algorithms. These algorithms ensure secure storage (confidentiality, integrity, authenticity, and freshness) and authentication of running VMIs. The keys they use are however never exposed.

5.3 Cloud Provider Key Issues

Key management has always been an issue since the dawn of cryptography. A lot of key management strategies and solutions have been proposed since then in the literature. Of particu-

lar interest are overall solutions that take the life cycle of a key into account, as proposed by Björkqvist et al. [BCH+10]. An open standard for enterprise key management has been built on top of their design, the *OASIS Key Management Interoperability Protocol* (KMIP, [oas]). However, these solutions are external to the cloud and cannot be easily adopted to provide key management for VMTs. Notwithstanding, they are reasonable solutions when providing keys by external means, as described in subsection 5.2.1.

A lot of cloud storage providers use encryption to secure their customer's data. However, the level of security varies and usually, the provider has access to the data due to a shared key that is known to the provider. Even worse, to the best of our knowledge, there is currently no IaaS provider which supports encryption of VM Images. The following table (Table 5.1) gives an overview of the security mechanisms in place at a few exemplary cloud storage providers.

Table 5.1: Cloud Storage Provider Security Overview

Provider	**Encryption**	
	in Transit	**of Data**
Amazon S3	Yes (HTTPS, signed Requests)	None
DropBox	Yes (HTTPS)	Yes (Shared key AES-256)
Mozy	Yes (HTTPS)	Yes (Shared key 448 bit blowfish)
CrashPlan	Yes (Proprietary)	Yes (User key[1])

DropBox and *Mozy* do encrypt their data, however, with a shared key that is known to the provider [dro, moz]. The security documentation of *CrashPlan* states that the data is encrypted with a randomly generated key which is in turn encrypted with the user's password. However, the same document also states that their administrators can access the data at any time [cra]. *DropBox* and *Mozy* additionally use *deduplication* (e.g. [QD02]) in order to save space. Deduplication, however, can lead to serious privacy issues when applied across accounts [HPSP10]. In order to protect data from the provider, it is of course always pssible to do local encryption before uploading a file. Amazon even provides a Java-based framework that interacts with their S3 storage and encrypts files locally [ama] before uploading them to the cloud.

In order to enhance trust in the cloud provider's infrastructure, a higher level of transparency is desirable [CGJ+09]. However, this transparency is somewhat contrary to the original idea of cloud computing, which promotes the abstraction from the actual cloud implementation as a major benefit of cloud computing.

When not trusting the cloud provider at all, it is possible to operate on encrypted data. Particularly known are the concepts of Fully Homomorphic Encryption [Gen09] and Yao's Garbled Circuits [Yao86]. They are however contrary to cloud computing's performance benefits due to their huge complexity overhead. Additionally, von Dijk et al. [VDJ10] have shown that cryptography is not enough in a multi-tenant environment like a cloud. Additional techniques like access control and a secure virtualization architecture are needed to support the cryptographic methods. These shall support the stronger separation and isolation of different customers from each other and from the potentially untrusted cloud provider. Christodorescu et al. [CSS+09] argue that beside virtualization security, machine introspection and understanding of the inner workings of a VM is necessary as well.

[1] However, their documentation on security states: *"Admins can restore without password, allowing easy local fast restore"* [cra]

5.4 Cloud System Model

We consider the most general cloud service model that provides the ability for a customer to run arbitrary Virtual Machines in the cloud (*Infrastructure as a Service, IaaS*). This cloud consists of a *client* (the cloud customer) and the cloud provider, as depicted in Figure 5.3.

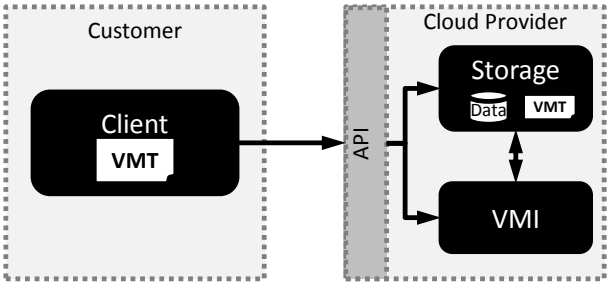


Figure 5.3: The IaaS Cloud Model. The client accesses the cloud provider's infrastructure from his/her premises over the internet.

The client is able to provide arbitrary VMs that run on the cloud provider's infrastructure. These VMs are provided as VMTs by the client. Multiple VMIs of these templates can be run inside the cloud. If these VMIs are designed to incorporate software like a web server, public users – neither part of the customer nor cloud provider – can use this portal and access services provided by the software running in such a VM. This software can in turn use the persistent storage to save data by the portal users (e.g. documents, photos) which is potentially confidential.

Additionally, for our approach to work, we assume that the physical hosts in the cloud provider's infrastructure are equipped with a TPM and HW (e.g., mainboard and CPU) that supports the execution of code within a Secure Execution Environment (cf. 5.1). Since modern off-the-shelf computers provide these features, this is a very reasonable assumption.

5.5 Trust and Adversary Model

Theoretically, an administrator of the cloud could mount many attacks, e.g. eavesdrop memory of a running VMI, as it most likely contains plaintext data. An administrator may furthermore have access to the local storage that functions as a back-end for cloud storage and saved VMTs. Of course, cloud providers want to avoid this by enforcing physical access control, role based access control, and operation surveillance. Furthermore, no single person accumulates all the rights necessary to mount such attacks. Hence, it is reasonable to assume that access to physical memory is (1) logically prohibited, i.e. a memory dump of a VM is not possible, and that (2) physical memory access is not possible, e.g. to mount a *cold boot attack* [HS08, HSH+09]. Otherwise, an adversary could access plaintext data which cannot be prohibited with means other than homomorphic encryption.

Under these reasonable and practical assumptions we propose a solution that protects VMTs and storage from being eavesdropped on (confidentiality), from being modified (integrity and authenticity) and from replay-attacks (freshness). The main focus is hiding the keys from anybody but the client, i.e. even the cloud provider cannot access the keys.

5.6 Design

A sole key management solution does not suffice, because even if the keys were stored securely, their exposure to a running VMI implies loss of control over the key. The key would no longer be concentrated at a single point and its usage could no longer be verified. The key could be eavesdropped on, accidentally saved on storage or be distributed. Therefore, we propose a solution that hides the keys by providing a model that abstracts cryptography from the VMIs and software using it, by providing a legacy plaintext environment while still providing secure storage whenever data leaves a VMI. The necessary keys involved in the cryptography are still protected against malicious cloud providers.

Idea. Our solution enables the client to provision a key in a secure way that can later be used from within the cloud without exposing it to neither the VM nor the cloud provider. This is possible because the key is wrapped – i.e., its use is restricted to one publicly known algorithm which is enforced by a hardware component and its correct execution can in turn be proven to the client. The key never leaves the algorithm, but the algorithm does all cryptographic operations on behalf of the user (either the client, or the hypervisor). Two components support the idea, namely

Setup Component for secure key provisioning from the client to the cloud. The client receives a certificate from the hardware module that assures him that a certain unwrapping mechanism is only possible by an algorithm known to the client. The client can then wrap its key if it trusts this configuration or algorithm respectively. The wrapped key can only be unwrapped by this known algorithm inside the Security Proxy.

Security Proxy works as a trust anchor for key usage later on. The Security Proxy hosts the known algorithm that is the only one that can access the wrapped key.

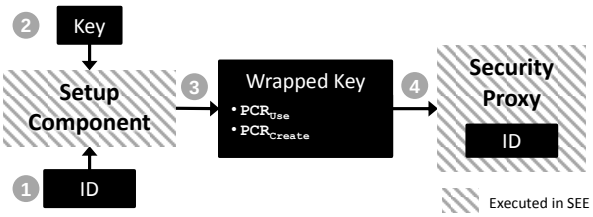


Figure 5.4: Setup and usage of the key with Trusted Computing. Hatched components are executed in an SEE.

We assume that one key can only be unwrapped by exactly one publicly-known algorithm. This is achieved by diversifying a basic algorithm to be binary unique (see implementation details in section 5.7). We deliberately produce different algorithms for each key, in order to guarantee access to a key is only given to exactly one *known* algorithm. This algorithm is hosted by the Security Proxy, so by assigning a unique ID to each Security Proxy, the algorithms get unique as well. This can be achieved by concatenating an integer number to each Security Proxy.

Secure Execution. The Setup Component as well as the Security Proxy execute in a Secure Execution Environment (see section 5.1). This implies that the client can get a certificate of proper execution issued by the hardware component.

In combination with the fact that each Security Proxy is unique, this enables the assignment of a Security Proxy to a VMT, and in turn to its corresponding VMI. The Security Proxies are identical in their functionality but can be distinguished by a unique ID. Each Security Proxy provides (1) confidentiality, (2) integrity, (3) authenticity and (4) freshness of persistent data. It keeps the involved cryptographic keys secret and only functions as an unavoidable black-box that makes secure storage for VMTs and their corresponding VMIs available. The access to external secure storage or to the client is possible by providing a translation layer between plaintext and ciphertext domains. Therefore, the security objectives can be observed and audited. The usage of the Security Proxy is depicted in Figure 5.5.

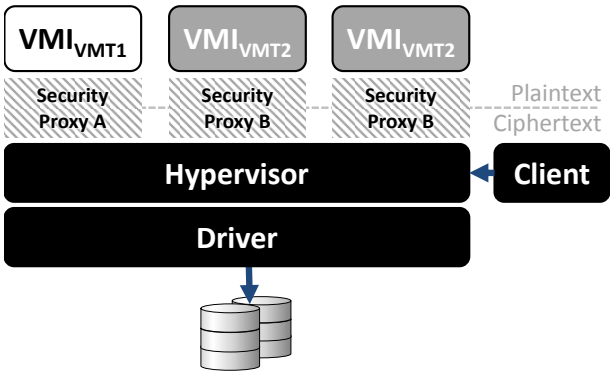


Figure 5.5: Introduction of the *Security Proxy*. VMIs stemming from the same VMT share the same shade of gray.

5.6.1 Components

The *Setup Component* enables secure key provisioning by assigning a unique ID to each VMT the client registers. The key supplied by the client is then wrapped using the platform configuration (PCR) of the Security Proxy with that exact ID. That means, the client encrypts his key using a public key that is certified that its associated private key can only be used by the Security Proxy with the appropriate ID. This is ascertained by the `PCR` value during use by the Security Proxy ('$\texttt{PCR}_{Use}$' in Figure 5.4). Furthermore, the configuration state of the machine while creating the wrapped key must be saved as well ('`PCR Create`' in Figure 5.4), so that the Security Proxy can check the platform configuration at the state of creating the key.

The *Security Proxy* is the only entity having access to the client-supplied key in the cloud, as it was wrapped by the *Setup Component* during provisioning. When the Security Proxy has access to the wrapped key, it ensures that the `PCR` value during creation of the key was a known-good value, hence it was created in a trusted environment. Furthermore, the placement of the Security Proxy is crucial, in order to make it transparent to the infrastructure while at the same time providing a guarantee that its use is mandatory (as shown in Figure 5.5) so that the enforcement of security objectives is inevitable and hence can be audited.

The Security Proxy should not just provide key management capabilities, but access to secure storage and a trusted path to a running VMI. It is required to provide the following functions:

Protect enforces the desired security objectives (confidentiality, integrity, authenticity, freshness) by applying cryptographic means and transforming the plaintext message to ciphertext. Its counterpart is called

SecVerify which transforms data back to plaintext in case the verification of security properties (integrity, authenticity, freshness) was successful.

We additionally want to add *auditability* by using a *proof of execution* to be able to proof and verify the adherence to security objectives. In order to provide the three main functionalities mentioned above, the Security Proxy is internally composed of the following cryptographic primitives:

Authenticated Encryption – To preserve confidentiality and authenticity.

Monotonic Counters – To guarantee the freshness and to detect replay attacks.

Verified Decryption – To reverse the process of encryption and to verify its authenticity.

Counter Comparison – The saved counter state is compared with the current monotonic counter value in order to prevent replay attacks.

COPE – *Certificate of Proper Execution* of internal computations for a third party to provide verifiability.

PROTECT is achieved by using the Security Proxy's primitives *Authenticated Encrpytion* and *Monotonic Counters*. *Authenticated Encryption* transforms the plaintext message to ciphertext while incorporating a Message Authentication Code (*MAC*). The *Monotonic Counter* is increased and saved along with the encrypted message as ciphertext.

PROTECT's inversion, SECVERIFY, is based on *Verified Decryption* and *Comparison*. While *Verified Decryption* reverses the process and transforms the ciphertext back to plaintext, given that the correct key is used, *Verification* is used to check the *MAC* and thus the message's authenticity and integrity. *Comparison* is necessary in order to compare the current counter value to the one[2] saved and encrypted.

All of the mentioned primitives use *COPE* to cryptographically ensure to the client that an operation has been carried out successfully. This is realized by an authenticated log file that is generated inside the Security Proxy and its generation is therefore verifiable as it runs inside the SEE.

5.6.2 Placement.

The placement of the Security Proxy is crucial, as its use shall (1) be transparent, but (2) mandatory at the same time. In order to minimize key distribution and to improve performance, the Security Proxy is local to every physical machine. We identified a placement so that the security objectives can be guaranteed in all IaaS service model use cases: The deployment of a new VMT by the client, its instantiation and secure access to the corresponding VMI. Furthermore, the basic building blocks of this protocol allow for VMT Migration from one to another physical host.

[2]This concept implies that every message and update to that message have their own counter. This can be achieved using Monotonic Counter Trees [SVDO+06]. For the sake of simplicity we abstracted from that fact and refer to them just as *'the counter'*.

The Security Proxy is to be placed in a layer where the virtual machine monitor (VMM) accesses external storage. Furthermore, it must be used when the VMM receives a VMT in order to instantiate it as a VMI. For provisioning of the keys, the cloud infrastructure must allow the client to access one *Setup Component* as the keys are local to each Security Proxy. If a VMI is created on a different host, these keys have to be migrated into another Security Proxy.

5.6.3 Instantiation

After the client supplied a wrapped key to the Setup Component, he is required to provide a VMT that is encrypted with the symmetric key he supplied. When a VMI is being created from that VMT, the hypervisor can decrypt the VMT using the Security Proxy's SECVERIFY function. The Security Proxy with the correct corresponding ID is the only entity getting access to the symmetric client's key that is protected by the wrapped key. This unwrapped key is then used for all its cryptographic primitives and depicts the trust anchor. When such a Security Proxy is created, it needs to be made sure that the one with the correct ID correlating to a particular VMI is created. If the wrong ID is used in the Security Proxy it is however not a problem, as a different resulting un-wrapped key could only decrypt and instantiate a different VMI which in turn can only access its external storage belonging to the VMI anyway.

5.6.4 Revocation

For the deletion/revocation of keys, the process has to be reversed. This means that the wrapped keys have to be destroyed as well their associated Security Proxies. As a deletion is generally hard to prove, our solution uses an SEE, analogously to the *Setup Component*. The code responsible for the deletion of the wrapped key, the Security Proxies and the reset of the monotonic counters can then be proven to the client.

5.6.5 Interaction

The following gives an overview of how the entities (client and hypervisor) interact with the Setup Component and the Security Proxy (Figure 5.6).

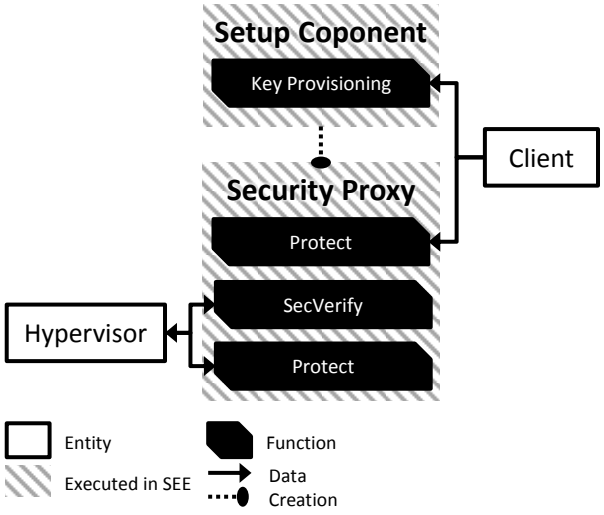


Figure 5.6: Interaction of the client, hypervisor, Setup Component and Security Proxy.

First, the client provisions the key by using the *Key Provisioning* function of the Setup Component. After that, the client is able to the Security Proxy's PROTECT function in order to provision data. The hypervisor in turn is able to access and verify protected data using the SECVERIFY function provided by the Security Proxy. Persistent data is written back, again, using the PROTECT function. As the keys are already provisioned to the Setup Component and in turn available to the Security Proxy, they can be used to proof authenticity in a secure channel, so that it becomes a trusted channel. A detailed description of the protocols is given in the next section.

5.6.6 Sequence Diagram

Sequence diagram for the key provisioning phase (Setup Component).

Summary: The *Client* wants to securely provision a key to the SBS component, so that it can be used in by the SBS component only. In order to do so, the SBS component generates a wrapped key, that is, a *"capsule* which certifies, that a secret can only be used in a pre-defined environment and platform configuration (PCR_{use}). The client can then encrypt (*"encapsulate*) the secret with the public key of this wrapper. The clients key is then sealed to a known platform configuration for later use.

Predefined functions:

SIGN Creates a digital signature using a secret key sk.

ENCRYPT(PK, X) Encrypts the message x under asymmetric public key pk

Asymmetric keys (pk, sk) are implied to have already been generated.

5.6.7 Analysis

The placement of the Security Proxy does not change fundamental parts of the cloud architecture and maintains compatibility with legacy VMs that are not aware of encryption, MACs, etc. These VMs do not even have to be changed when securing the external storage they use. It further prevents cryptographic keys from being exposed to the VM, which ensures that they are not leaked – unintentionally or by distributing VMTs. The standard use cases of a cloud (VMT deployment, VMI access, VMI migration) are not changed in any way from a client's perspective. Therefore, it does not hinder the client in its original intention of using the cloud, but only adds security value.

Security. The client gets a certificate that assures the unwrapping can only be done by a certain algorithm, i.e. the Security Proxy with a specific ID. This proofs to the client that an unwrap operation of the key he provided is always assigned to a specific unique ID. The client can investigate the publicly known algorithm which is allowed to access his key. He can then decide whether to trust the key provisioning and whether to wrap the key, so that the Security Proxy can eventually access it.

As the two phases (Setup and Usage of a key) and their corresponding components (Setup Component and Security Proxy, respectively) are both executing a Secure Execution Environment (see section 5.1), the client has an additional run-time assurance of code that uses its key.

The proposed design of the Security Proxy adheres to the security objectives and provides verifiable key management and auditing functionality. It protects external storage's confidentiality and authenticity by means of encryption and MACs. Moreover, the MACs used to authenticate data only rely on the symmetric key k and therefore no PKI is needed when using

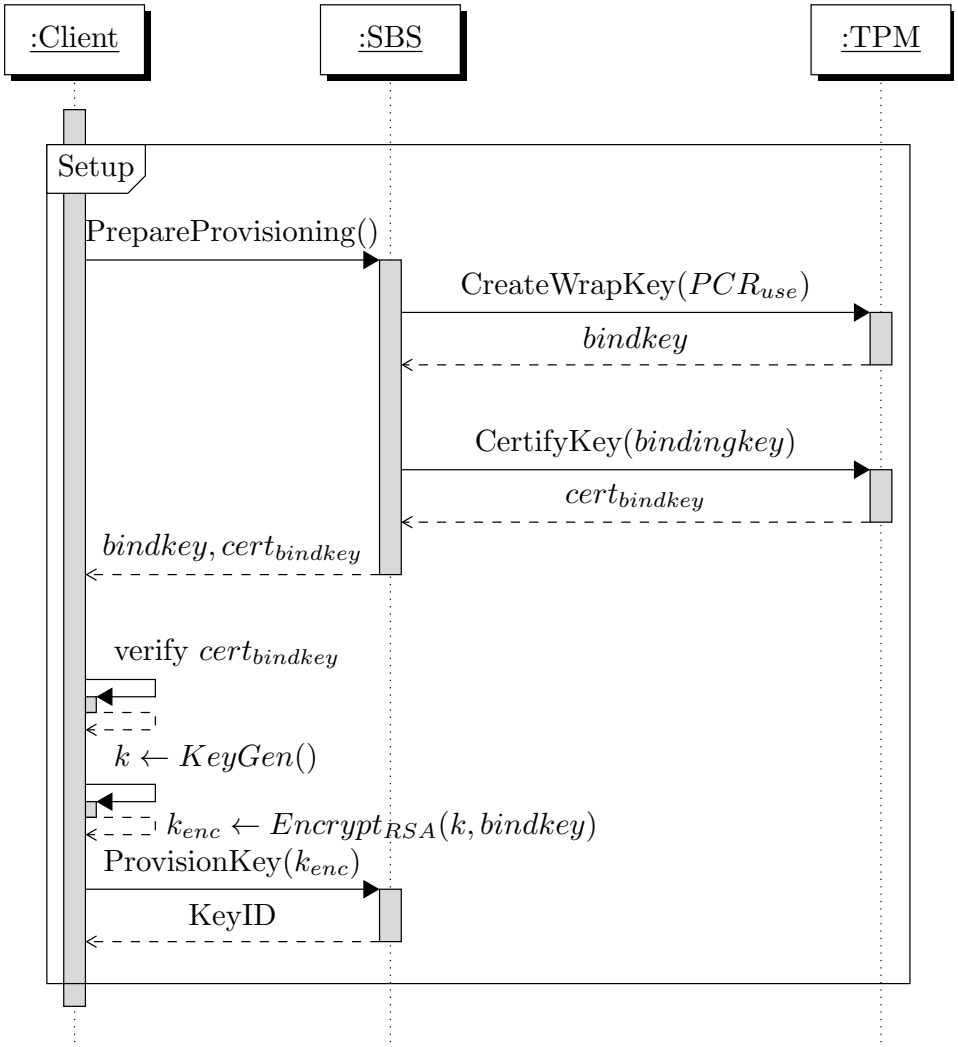


Figure 5.7: Sequence Diagram of the Setup Phase (key provisioning) for the Setup Component

different Security Proxies to verify the authenticity of data – as opposed to digital signatures. A PKI is, however, necessary for the client to check the COPEs. The frequency with which they are checked is up to the client, though. The keys itself are only available to the Security Proxy and isolated from external access, which means the use of the Security Proxy is mandatory and it can consequently verify the internal key usage and provide logs for auditing. We assume that the Security Proxy is created securely (in an SEE, see section 5.7 for implementation). As it does not leak the key, it can only grant access to exactly *one* VMT of which a VMI is created. This VMI has no means to access cryptographically protected storage except the one belonging to this very VMI through accessing the Security Proxy. However, the entity controlling the Security Proxy could change the plaintext sent to and received from the Security Proxy. To ensure that the controlling entity does not behave in a malicious way, its behavior should be verifiable as well. This is shown in the implementation details we provide in the next section.

5.7 Implementation Proposal

To implement our proposed design several building blocks are needed: (1) A hypervisor to mange running VMs and isolate them from each other and from external access. (2) A Secure Execution Environment that provides isolation of secret keys and can verify that access to them is only granted to well-known code. (3) A cloud software that manages the hypervisor and its VMs.

Building Blocks. The first building block is NOVA [SK10], a very modular hypervisor featuring a micro-kernel-like design. It separates the hypervisor into multiple components and introduces a Micro- hypervisor which is mainly responsible for the enforcement of isolation between these components. It provides a small and well defined TCB and can be easily extended with additional security features.

The secure execution environment can be provided by late-launch capable processors like AMD's SVM [Adv05] or Intel's TXT [Int07]. For that purpose a framework like Flicker [MPP+08] seems to be suitable. However, Flicker suffers from rather slow transition times between commodity execution mode and secure execution mode. Thus, it is not suitable for streaming plaintext/ciphertext blocks of data through the Security Proxy with respect to performance. Similar to TrustVisor [MLQ+10], we propose to use the Secure Execution Environment (SEE) as a 'factory' in the sense of a software engineering design pattern. That means, the SEE only creates a Security Proxy which can be verified and is in turn trusted.

For a cloud scenario to allow remote administrator access to each physical machine, a cloud management software is needed that can communicate with the hypervisor in order to instruct it to start/stop the VMs etc. Because of choosing NOVA, we do not even need to increase the TCB for that purpose, as we can leverage the fact that one VM could run a traditional Linux-based operating system to run the cloud management software and connect to the rest of the cloud infrastructure via network. This Linux VM does not have to be trusted or secured as it works a relay and just writes and reads encrypted and authenticated data. We chose OpenStack [Ope10] for that purpose, a suitable open-source cloud manager that is compatible with a variety of hypervisors and easily extensible.

Architecture. The generated Security Proxys have to be created for every VM (see Figure 5.8), as they are designed to only incorporate one key at a time. This Security Proxy runs below the VMM as an isolated user mode process of the hypervisor. Due to the fact that the

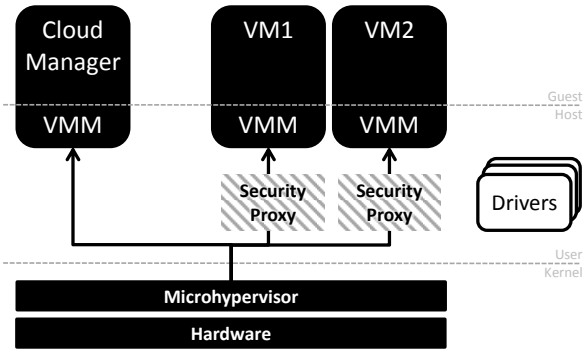


Figure 5.8: Proposed Architecture with NOVA

Security Proxy does not rely on access to hardware and is event-driven, we place it in NOVA's user space. The advantage is that we do not incorporate the TCB of an operating system and VMM which allows easier verification of the Security Proxy.

Management. Our proposal only requires to add key provisioning capabilities to contemporary cloud APIs. To start and stop VMs, the client accesses the cloud management VM which in turn invokes the Security Proxy. Access to untrusted storage is routed through the Security Proxy which also ensures that keys are never exposed. The secure channel is also established with the help of the Security Proxy. As every VMM has only access to its corresponding VM, a trusted channel can be established between the client and an instantiated VM via the Security Proxy (see Figure 5.9).

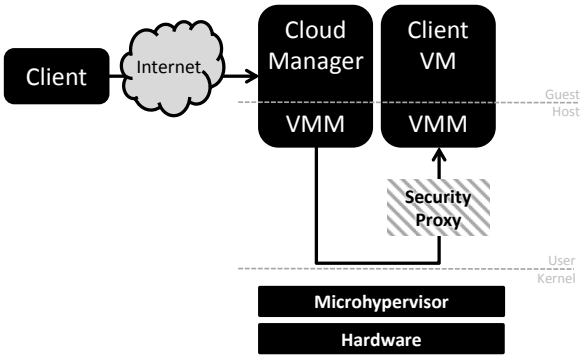


Figure 5.9: Trusted Path between client and its VMI.

The administration of the cloud, or the VMs respectively is done from within the *cloud manager* and allows a local or remote administrator to start and stop VMs. As it runs in a separate compartment it is isolated from the other VMs and especially the hypervisor and cannot directly manipulate or eavesdrop data.

Internals. In order to allow VMs access to the data, the VMM of each guest calls the Security Proxy to de- and encrypt a block and to provide the block level abstraction layer, expected by standard operating systems.

Security. In order to proof the integrity of the hypervisor and Security Proxy to the client we make use of a DRTM which is implemented by tboot [tbo] for Intel late-launch-capable [Int07] architectures and OSLO [Kau07] for AMD powered [Adv05] machines.

Restriction. There is a restriction concerning the combination of hardware virtualization and late launch features. As stated in the standard lecture for late launch capable processors by Intel (*"Dynamics of a Trusted Platform"*, [Gra09]), the hardware virtualization features must be enabled, but turned off, when using the late launch (Intel TXT) capabilities. Moreover, the measurement of the hypervisor during boot (Dynamic Root of Trust for Measurement, DRTM) prohibits the use of TXT features, as TXT has already been used during boot and the processor is currently executing in the secure execution mode. At least according to the documentation, AMD's SVM does not suffer from those limitations.

5.8 Conclusion

In this chapter, we elaborated on the issues of deploying user credentials, like cryptographic keys, within the cloud providers infrastructure with respect to a not fully trusted cloud provider. We argued that under the assumption that the cloud provider applies techniques to prevent physical attacks, Trusted Computing concepts and security extensions of modern off-the-shelf hardware can be leveraged to mitigate the remaining threat of credential disclosure. We proposed an architecture that makes use of these concepts and technology and showed how it can be used to implement secure block storage for virtual machines.

Chapter 6

Key Management for Trusted Infrastructures

Chapter Authors:
Michael Gröne, Norbert Schirmer (SRX)

6.1 Introduction

In this section we give a high level overview of the key management within the Trusted Infrastructures (cf. Deliverable D2.1.1 Chapter 13) which is used to build a trusted cloud infrastructure. Some preliminary Trusted Computing[tcg] basics were introduced in Section 5.1.

6.2 Architecture Overview

In the Trusted Infrastructures (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 6.1).

All hardware components, appliances as well as the TOM, are equipped with a Trusted Platform Module (TPM) [Tru11]. When a component is started the TPM is employed for secure booting to ensure the integrity of the hardware and software (in particular of the security kernel) of the component. Moreover, the local hard drives of a component are encrypted by a key that is stored within the TPM. Via sealing, the component can only decrypt the local hard drives if the TPM has crosschecked the integrity of the component. Hence only an integer security kernel can access the decrypted data. Note that on an appliance there is no need for manual administration and hence no almighty root account is needed. All administrative tasks are controlled by the TOM. This mitigates the risks of malicious insiders like cloud administrators within the premise of the cloud provider.

The TOM is in charge to deploy configuration data (including key material and security policies) to the appliances. Security services within the security kernel then handle the configuration and ensure that the security policies are properly enforced by the component. Via the Trusted Management Channel (TMC) the TOM ensures the integrity of an appliance using remote attestation before transmitting any data. The communication between TOM and the appliances is secured by a trusted channel (cf. D2.1.1, D.2.4.1).

Conceptually, appliances are stateless with respect to all configuration data.[1] On every

[1] Exceptions are caches for limited offline periods.

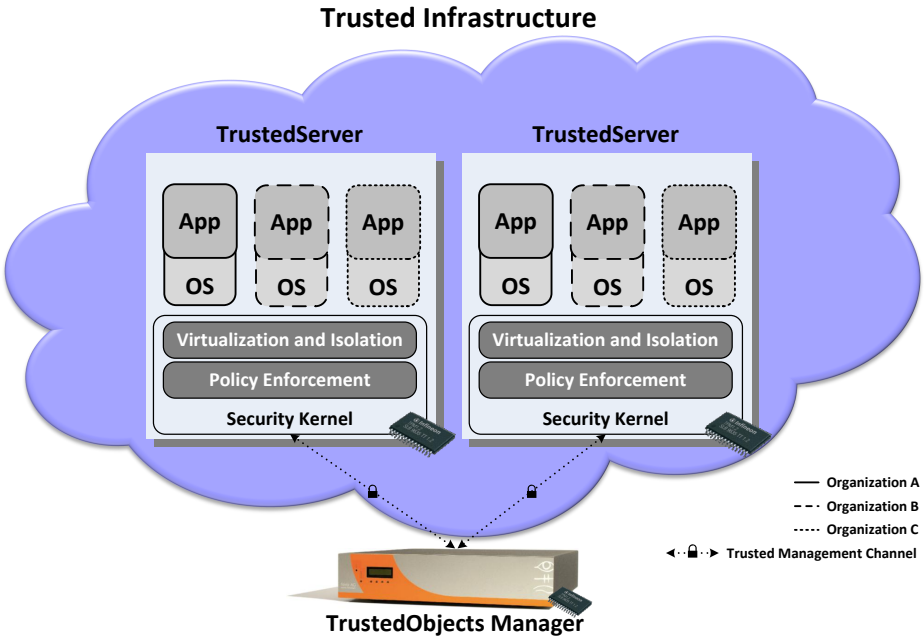


Figure 6.1: Schematic Trusted Infrastructure - TrustedServers managed by TrustedObjects Manager.

boot they retrieve the current configuration from TOM and also maintain a permanent channel to TOM to retrieve configuration updates (e.g., changes in the security policy) while they are running.

6.3 Key Management for Trusted Virtual Domains

Central to Trusted Infrastructures is the concept of a Trusted Virtual Domain (TVD) [CLM+10, CDE+10], which allows to deploy isolated virtual infrastructures upon shared computing and networking resources. By default, different TVDs are isolated from each other. Communication is restricted to compartments within the same TVD and data at rest is encrypted by a TVD specific key. Remote communication between components of the same TVD over an untrusted network are secured via encryption (secure channel). Only appliances of the same TVD, which have access to the same TVD key, are able to decrypt the data. An appliance can be equivalent to compartment, if there is only one compartment per appliance, or in case of the TrustedServer a compartment is a single virtual machine instance and its operating system and applications. Therefore, an appliance such as the TrustedServer can simultaneously run various compartments attached to different TVDs (cf. Figure 6.2).

The TVD isolation builds on a public key infrastructure (PKI) managed by the TOM. For each TVD, two public/private key pairs are generated by the TPM of the TOM. The private keys never leave the TPM (or other special purpose cryptographic hardware) of the TOM. Only the public keys are transmitted to appliances.

- PKI for TVD signature keys. The TOM acts as the root CA for the TVD. When a appliance attempts to enter a TVD, it creates a public / private key pair and the appliance signs

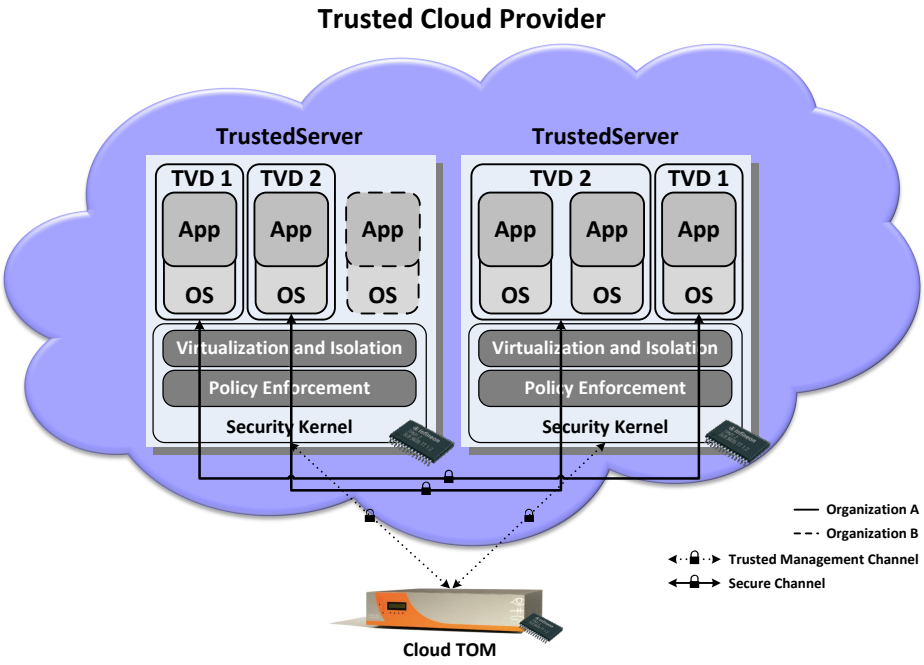


Figure 6.2: TrustedServers and TVDs, managed by TrustedObjects Manager.

the public key. This signed public keys can be used by the appliance as authentication tokens to establish a secure communication channel between different compartments of the same TVD. to prove membership of a TVD.

- TVD encryption keys. An compartment can encrypt data with the TVD key but only the TOM can decrypt this data. Typical data that is encrypted by the TVD key are symmetric keys created by the appliance that are used to encrypt payload data. The encrypted symmetric key can be sent along with the encrypted payload data to other members of the same TVD. The receiver asks the TOM to decrypt the symmetric key, and then decrypts the payload data. Keeping the TVD decryption key local to the TOM and not spreading it to all the appliances enables an easy revocation of TVD decryption capabilities.

6.4 Resilience and fault tolerance

In the architecture described above, the TOM is a critical component as it manages all the security policies and the key material. If the TOM fails the complete infrastructure it manages will also fail. Hence it is important to be able to engineer a fault tolerant, resilient TOM. Currently the TOM is a single point of failure. However the key management and key distribution was designed to allow for replicated TOMs supporting cold, warm, or even hot standby [Jal94]. The crucial design decisions here are the stateless appliances and the separation of ordinary configuration information and the core key material (private keys never leave the TPM of the TOM) within the TOM. Ordinary configuration information can be shared between TOMs and is sufficient to allow a new TOM to reconfigure the whole infrastructure. Basically, when a TOM fails, another TOM can take over and distribute a new configuration and a new PKI to the

appliances.[2] For the TVD signature keys this scheme works fine. The new TOM distributes its public key to the appliances and thereby revokes the old ones. For the TVD encryption keys the situation is more involved, as the private key of the old TOM never leaves its TPM any other TOM is unable to decrypt data which was encrypted for the old TOM. Here the idea is to use broadcast encryption schemes [FN94], to enable all TOM replicas to decrypt the data.

6.5 From Private to Public Cloud

In the scenario described so far one trusted management component, the TOM, is in charge to manage all the infrastructure. In a cloud scenario this includes the TrustedServers of the cloud infrastructure as well as all managed appliances of the customer. Such a setup matches the requirements of a private cloud but do not fit well into the idea of a public cloud. It should not be assumed that all on-premise appliances (e.g., mobile devices, VoIP telephones, desktop and laptop PCs and on-premise servers) of the customer should be managed by the cloud management component (cf. Figure 6.4).

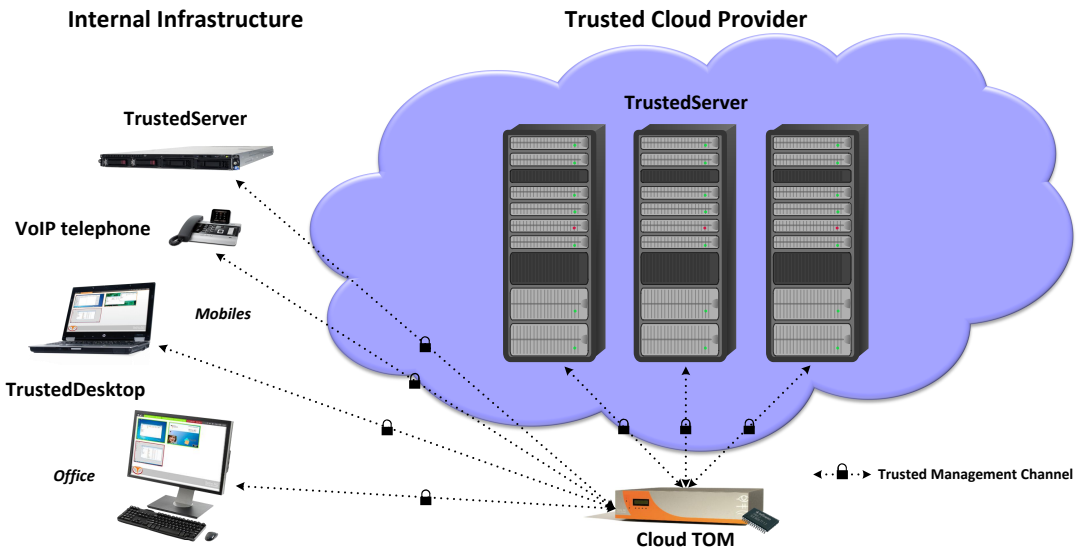


Figure 6.3: Internal Infrastructure appliances managed by cloud TOM.

This would imply that all the infrastructure of any customer is managed by a cloud providers management component. This is not an ideal setup. Note that this is not a issue of the trust model as we asses the trust in the cloud provider's TOM via Trusted Computing technologies. But an organization should be able to manage its own infrastructure by themselves and it should be possible for an organization to use the services of more than one cloud provider at the same time.

We propose to extend the scenario with federation of TOMs. Every customer / organization has its own trusted management component which is in charge to manage the organizations on-premise infrastructure (e.g. TrustedDesktops). Moreover, the cloud provider has a TOM. Both

[2] Note that during the setup phase of an appliance the set of TOMs that are accepted as managers are bound to the appliance. Therefore an appliance can not be hijacked by a foreign TOM. Moreover, an appliance uses remote attestation to ensure the integrity of the TOM.

are connected via the TMC. A protocol between the organizations TOM and the cloud providers TOM allows the organization to extend its infrastructure to the cloud (cf. Figure 6.4).

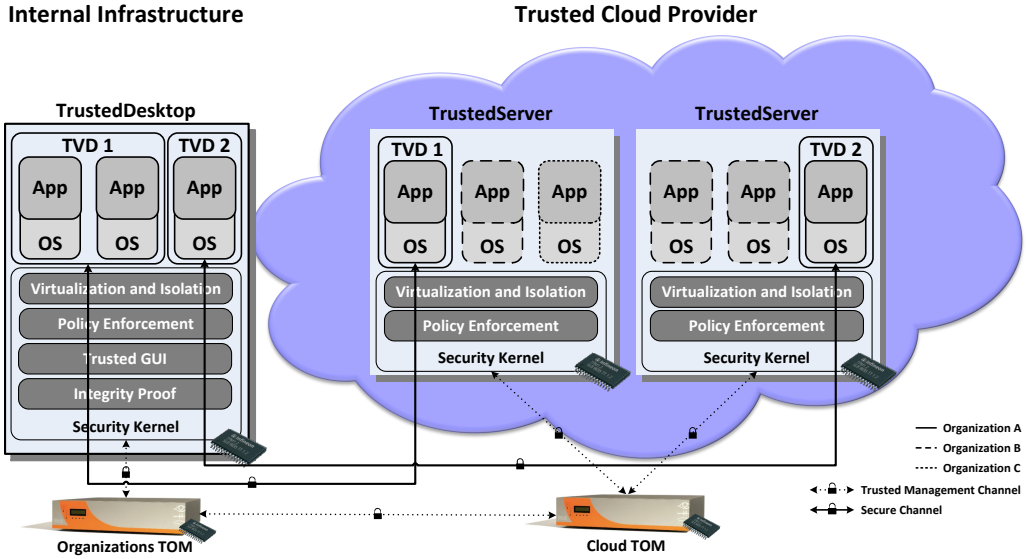


Figure 6.4: Internal Infrastructure (organizations TOM) combined with Cloud Infrastructure (cloud TOM).

So let us revisit the key management issues within this scenario of a federation of TOMs. We start with the signature key for a TVD. When an organization wants to start a VM in a TVD within the cloud, the organizations TOM asks the cloud TOM to start a compartment within a TVD for which it has the root CA, CA_O. The cloud TOM will then create a new root CA for that TVD, CA_C. From now on not only CA_O is accepted as root CA for the TVD but the set $\{CA_O, CA_C\}$. This is propagated by the organization's TOM as well as the cloud TOM to all relevant appliances. On an more abstract view this operation can be seen as a *join* of TVDs. Two separate TVDs, identified by CA_O and CA_C are joined to a combined TVD $\{CA_O, CA_C\}$. In the cloud scenario this join is not symmetric but more a master-slave relationship, with the organizations TOM acting as master and the cloud TOM acting as a slave. The security policy of the organization is pushed to the cloud TOM and then further to the servers within the cloud.

With the TVDs encryption key the extension towards a federation of TOMs is not as straightforward as with the signature key of the TVD. As described above TVD encryption is handled by an asymmetric key where the private key is stored within a single TOM, or via broadcast encryption within a set of TOMs. In a scenario where the cloud TOMs may dynamically change (as an organization buys resources from different cloud providers) this rather static setup does not fit well into the picture. Here we can go for at least two options:

- As the organization TOM acts as a master, the cloud TOM can always forward all decryption requests to the organizations TOM. The drawback of this approach is the tight coupling of the different TOMs which can no longer manage their appliances alone but always need the to refer to the master TOM.

- Alternatively we can relax our restriction of the private decryption key never leaving the TOM and also distribute it to the cloud TOM. This alternative seems to better suit the requirements of a distributed, federated infrastructure. Every TOM is capable to manage

all of its appliances on its own without having to consult a master TOM. Moreover, this also simplifies the task of a scalable, resilient TOM within a single infrastructure, as we do not need more advanced encryption schemes like broadcast encryption.

6.6 Conclusion

This chapter elaborated on the management aspects, especially the management component(s) for Trusted Virtual Domains in the context of a trusted cloud infrastructure. Trusted Computing technology is employed to guarantee the integrity of the trusted computing base which is distributed among various components and stakeholders. The architecture is designed such that the management components reflect the responsibilities of the different stakeholders. The cloud customer is responsible for managing the security policy of his organization, while the cloud provider manages the cloud infrastructure. The management protocol ensures that the organization's security policy is correctly enforced within the cloud components of the provider.

Chapter 7

Ontology-based Reasoning for Cloud Infrastructures

Chapter Authors:
Daniele Canavese, Emanuele Cesena, Gianluca Ramunno, Jacopo Silvestro, Paolo Smiraglia, Davide Vernizzi (POL)

7.1 Introduction

Cloud computing is grounded on a massive use of virtual appliances. To automatize the management of a virtualized system, it is necessary to describe it in a coherent and centralized way [BSB+10].

A simple yet powerful approach for modeling a knowledge domain (e.g., a cloud environment) is given by ontologies. In philosophy, ontology is the study of the nature of being, existence, or reality in general and of its basic categories and their relationships. Similarly, in information technology, an *ontology* is an explicit specification of a conceptualization. Its main objective is the formal representation of an application domain (i.e., our portion of reality), according to our knowledge of the domain itself. An ontology defines a set of representational primitives used to describe a domain of knowledge. These primitives are typically:

- *Instances*, *objects* or *individuals*. These may include concrete objects such as a person or a car, as well as abstract objects such a word or a number.
- *Concepts* or *classes*. They are collections of objects such as people, cars or numbers.
- *Relationships* or *properties*. They describe the way classes and individuals can be related to each other such as the `hasColor` or `isASubclassOf` properties.

Ontologies offer a great expressive power, giving the possibility to describe in a compact and more natural way a large variety of application domains with the benefit of a formal representation. Furthermore, an ontological model can be analyzed by a *reasoner* or *inferential engine*, a software component able to deduce additional properties and classifications using logic approaches. For example, a reasoner can infer that `A` can communicate with `C` (via `B`), although in the original ontology only the `A-B` and `B-C` connections are present. In addition, reasoners can also discover *inconsistencies*, that is impossible or forbidden situations. In an IT environment, consistency checking can be very useful to asses a configuration's validity or to find the policy conflicts.

Starting from a description of the infrastructure and a logical model relating the description to security goals, threats and available countermeasures, it is possible to automatically derive security properties of the system. This can leverage the effort of both cloud administrators, that need to discover vulnerabilities in their infrastructure, or auditors that need to assess the compliance with certain security properties.

In this scenario, ontologies offer a suitable tool for both describing the system and building the logical model. Moreover, they support automatic reasoning, that can be used to infer new properties – specifically, we are interested in security properties – following a bottom-up approach [KLK05, HSD07, FE09]. This is usually done by defining classes with more and more levels of abstraction, and linking them with consistent relations. For instance, we can derive a logical channel between two services from the existence of a network path between the two hosts running the services (with no filtering devices in between). Then we can state that the logical channel can be secured if, e.g., both the hosts have the capability to establish TLS channels.

Coming back to the general problem of modeling a system, one of the main issues of automatic approaches is how to actually gather precise information that correctly describes the system under investigation. The widespread use of virtualization in cloud infrastructures from one side increases the complexity of the description as it allows to dynamically modify the network topology, and add or remove virtual machines on-the-fly. On the other side, tools such as VMWare products or open source counterparts as Libvirt [Red11] are available and can be used to manage such a complexity [BSB+10]. Therefore, they can also be used to automatically gather real-time information about the system.

In this chapter, we propose a coherent ontology to describe a cloud infrastructure system and annotate security properties. More in detail, the system description can be provided at different layers: the *physical layer* containing the hardware devices and the physical networks; the *virtual layer* which includes virtual machines and virtual networks; the *software layer* that contains the software running on a physical or virtual machine, from the operating systems to the applications providing services; the *service layer* which describes services running in the system, their behavior and interaction; and finally, the *security layer* that defines security threats, requirements and mechanisms to annotate physical or virtual objects and services.

For each layer, we have selected standard (or at least publicly available) languages or ontologies to facilitate the integration of our description with other works in literature, thus allowing a wider range of analysis to be performed on a common knowledge base.

Our contribution is two-fold. On one side we provide the "glue" between ontologies at the different layers, so that the information at each layer can be used by others. This contribution turns into the definition of a *unified ontology*. On the other side, we define a building block for the virtual layer, i.e. a completely new ontology inspired by the Libvirt object model and XML language. The new ontology to describe virtual systems, which has been developed in collaboration with the Posecco project (EU contract IST-257129, `www.posecco.eu`), consists of 272 classes, 74 object properties and 102 data properties. The most important class is `VirtualDomain`, which is related to other 18 classes and is referenced in 34% of the properties. A more detailed description of the ontology is available in [SCCS11], while it is also publicly available for download at `http://security.polito.it/ontology`.

The remainder of this chapter is organized as follows: In Section 7.2 we discuss the related works, particularly in the field of the ontology-driven security analysis. and in Section 7.3 we introduce the languages on which we rely for the definition of our (overall) unified ontology. In Section 7.4 we introduce the new building block, i.e. the ontology for the virtualization domain. The unified ontology is described in Section 7.5, including a detailed example that demonstrates

the validity of our approach.

7.2 Related Work

In [YBS08], Youseff et al. proposed an ontology for the entire cloud computing environment, mentioning virtualization in their *Software Kernel* layer. However the focus of their work is to provide a description at a high level of abstraction, and thus it lacks details. Dastjerdi et al. [DTB10] proposed an ontology of the virtual environment based on the *Open Virtualization Format* (OVF) [DMT10]. The focus of their work is very similar to our ontology for the virtual domain, however their paper contains few details about the actual ontology that it is not publicly available.

In [DKF+03], Denker et al. proposed an ontology for describing the security aspects of web services; their work focused on security mechanisms such as encryption, signatures and authentication systems but did not include the concepts of threats and attacks such as the ontology described by Kim et al. in [KLK05]. Kim and his team revisited Denker's work, reorganized its structure in a more consistent fashion and added support for OWL-S. However, both representations totally lacked a systematic classification of asset elements.

A comprehensive approach was made by Herzog et al. in [HSD07]. Starting from [KLK05], they created an ontology capable of modeling security mechanisms, threats, vulnerabilities and also assets and their attempt is the starting point of our work. In [FE09], Fenz et al. expanded the Herzog ontology in an orthogonal direction, adding new concepts derived from a number of security recommendation standards such as the German IT Grundschutz Manual [BSI05] and the standard ISO 27001 [ISO05].

The automatic inferential capability of ontological architectures can be used to better understand the potential pitfalls of a system, as described by Steele in [Ste08]. He introduced an ontology for vulnerability assessment and he showed that this approach can be used to discover what a web service user can actually access using complex inferential chains, emphasizing the limits of the manual inspection.

Security attack modeling is also an interesting application field for the ontologies, as demonstrated in [VH06]. In this work, Vorobiev et al. proposed a common knowledge base for distributed firewalls and IDSes, modeling a set of web service attacks such as the DoS and Mitnick attacks. Using such models, a distributed architecture can be hardened thanks to the shared information about the current system status.

Furthermore, a noteworthy work of comparison and review of ontological techniques in the security field is available in [BLVG+08]. In this paper, Blanco et al. compared and analyzed about thirty works, confirming this as a very active research branch.

7.3 Existing Building Blocks

In this section we review the existing languages that we have used as a basis to build our overall ontology.

In Section 7.3.1 we briefly discuss the languages which we used to build and enhance our ontology, while Sections 7.3.2 and 7.3.3 introduce a number of XML based standards that inspired us in the creation of the hierarchical content of the ontology.

7.3.1 Ontology Languages

OWL [PSHH04] is a standard language to describe ontologies developed by the World Wide Web Consortium (W3C).

Before the introduction of OWL, the default language used to represent knowledge was RDF (Resource Description Framework). RDF contains the constructs needed to express the meaning of terms and concepts in a way that can be easily processed by a computer. However, RDF has some limitations like the constraints on cardinality and attributes. To overcome these limitations, the W3C developed a new language called OWL. The new features introduced by this language are the possibility to express some characteristics like transitivity and inverse, or the possibility to declare the disjunction or the combination of classes by means of boolean operators (union, complement, intersection).

OWL has been released in three different versions, with increasing expressive capabilities (and complexity): OWL-Lite, OWL-DL, OWL-Full.

OWL-Lite is the syntactically simplest version, that can be used to define class hierarchies and constraints but, for instance, it supports cardinality constains only permitting cardinality values of 0 or 1. OWL-DL, whose name denotes correspondence with description logics, is the intermediate version, with increased expressive power and two fundamental properties: computational completeness (all the propositions are computable) and decidability (all the computations end in a finite amount of time). OWL-DL includes all OWL language constructs with restrictions such as type separation (a class can not also be an individual or property, a property can not also be an individual or class). Finally OWL-Full has the maximum expressivity, at the price of no warranty about computational completeness and decidability. For example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right.

To enhance the expressiveness of OWL, the Semantic Web Rule Language (SWRL) [HPSB+04] was designed. SWRL provides a high-level abstract syntax which extends the OWL syntax, while OWL objects are still used for the semantic. Using SWRL it is easier to make complex definitions, such as chains of properties.

SPARQL [PS08] is a W3C standard language, similar to SQL, used to perform interrogations on OWL models, resorting to the semantic relations represented by this language. The query can use operators such as `DISTINCT`, `ORDER BY`, `LIMIT` that are also present in SQL with the same meaning. The results of the query can be in the same format as the knowledge base, or encoded in XML.

7.3.2 P-SDL (POSITIF System Description Language)

P-SDL [BLP+07] was developed in the context of the POSITIF project (EU contract IST-2002-002314). It is an XML-based language that comprises the definition of a vocabulary of elements and attributes. It can be used to formally describe an IT system, its logical and physical topology, functionalities, network configuration, and security capabilities and mechanisms of each node.

Let us consider the high level UML model shown in Figure 7.1. The main class is `Element` and all the network components, including the `Network` itself, are defined as its specializations (exploiting inheritance). `NetworkElement`, a subclass of `Element`, represents all the (physical or virtual) elements which compose a network. A `NetworkElement` has at least one communication point with some other entities: a `Link` represents the physical connection between two network nodes (through their own interfaces). An `Interface` is a physical

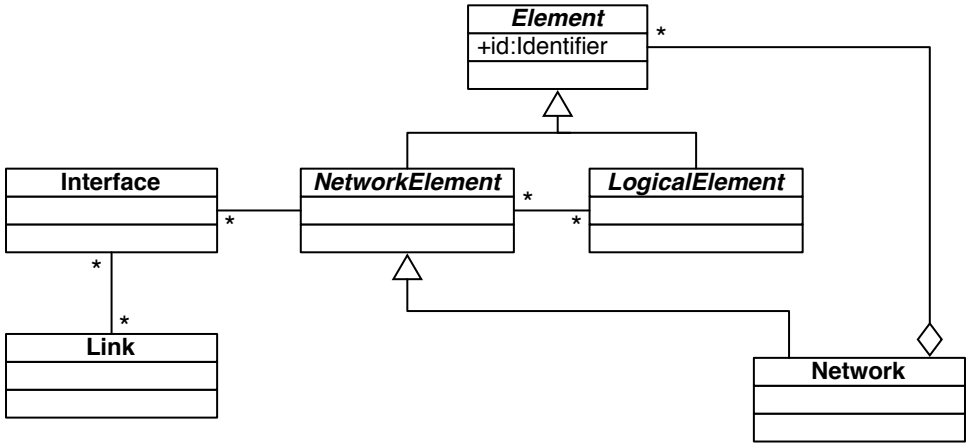


Figure 7.1: P-SDL language UML model.

point of access and communication between network elements. Many network addresses can be associated to a single interface. This is the basic entry point to any connected element.

`LogicalElement` is an abstract class used to represent features and abilities that a network node may have, including:

- Services: represent the ability to provide any given service, like a web or FTP server.
- Capability: represent the ability to enforce a security policy, through a security protocol or an application gateway, e.g. the capability to establish secure channels through TLS.
- Other logical elements that can not be classified inside these two main classes, for instance the presence of an operating system or hypervisor, a software element, or an application protocol.

7.3.3 WS-CDL

The Web Services Choreography Description Language (WS-CDL) [KBR+05] is an XML-based language that describes peer-to-peer collaborations of participants by defining their observable behavior from a global, external point of view.

The primary goal of WS-CDL is to provide a declarative language that defines from a global viewpoint the common and complementary observable behavior of services, the information exchanges that occur and the jointly agreed ordering rules that need to be satisfied. Other goals include reusability and modularity of the specification and composability of already existing descriptions into a new choreography.

The WS-CDL specification is aimed at being able to precisely describe collaborations between any type of participant regardless of the supporting platform or programming model used by the implementation of the hosting environment and it is therefore useful not only in the context of Web Services, but can be used more in general to describe any choreography between services.

7.4 An Ontology for the Virtualization Domain

In this section, we propose a novel ontology for describing virtualized environments and performing analysis in this field, with the added benefit of using solely standard languages and

standard OWL-DL reasoners. We note that this ontology is independent from the unified ontology discussed later, thus the virtual ontology is developed to be self-consistent.

7.4.1 Application Field and Scope of the Ontology

Before starting the description of our ontology we introduce the terminology that will be used through the remaining sections. The *hypervisor*, or *virtual machine monitor*, is the core component of a virtualization system; its role is to logically multiplex a number of physical resources. A hypervisor is a software running on a physical system called a *host*. A host can contain zero or more *domains*, or *virtual machines*, that are computing machines which comprehend several virtualized components made available through a hypervisor. Within each domain, an operating system, the *guest OS*, runs; different virtual machines may run different guest OSes simultaneously on top of the same hypervisor.

Our ontology-driven approach is motivated by the need for a coherent description of both a virtual environment and the underlying physical system. It allows the extraction of additional information, such as the topology of the existing virtual networks. Our ontology also enables a range of analyses, including the discovery of relationships between the physical hardware and virtualized elements (such as, for instance, when looking for the physical machines that provide the disk volumes to a domain). Knowing these relationships is particularly relevant for management purposes, to guarantee a proper allocation of the physical resources as well as to assess the effect of physical world on the virtual one (cf., e.g., [BSB+10]). We will come back to these issues and provide relevant examples in Section 7.4.5.

To achieve these goals we have organized our ontology into three main *layers*. Figure 7.2 presents a bird's eye view of the ontology using the UML class diagrams notation.

Starting from the Libvirt XML format, we have identified several classes which map all the objects that can be virtualized by a hypervisor. This brought us to the creation of the *virtual layer*, that contains elements such as the domains, together with all the virtual devices that are contained into the virtual machines.

One of the major limitations of the Libvirt XML format is the lack of a representation for the physical hardware. To bridge this gap, we have built the *physical layer* using the same base structure as its counterpart, the virtual layer. Their similarities allow us to represent both virtual and physical objects in a simple and modular way. It is important to notice that this layer is only intended to provide the minimum set of classes necessary for reasoning on the virtual realm, and it is not a comprehensive ontology of the physical hardware.

During the ontology creation process we have frequently encountered several "concepts", such as bus addresses or CPU features, that do not fit exactly into the virtual or physical worlds. For the sake of modularity, all these concepts were classified into an ad-hoc layer, the *logical layer*. Its content is rather heterogeneous and will be discussed further.

7.4.2 Logical Layer

Since both the virtual and the physical entities make an extensive use of logical notions, their description is provided first. The *logical layer* contains concepts which are used to describe several additional characteristics of physical and virtual objects together with elements which are used to establish relationships between the virtual and physical layers. Furthermore, it includes all the software related classes.

This layer consists of several classes, but the most distinctive ones are actions, mechanisms, features, identifiers and logical bridges.

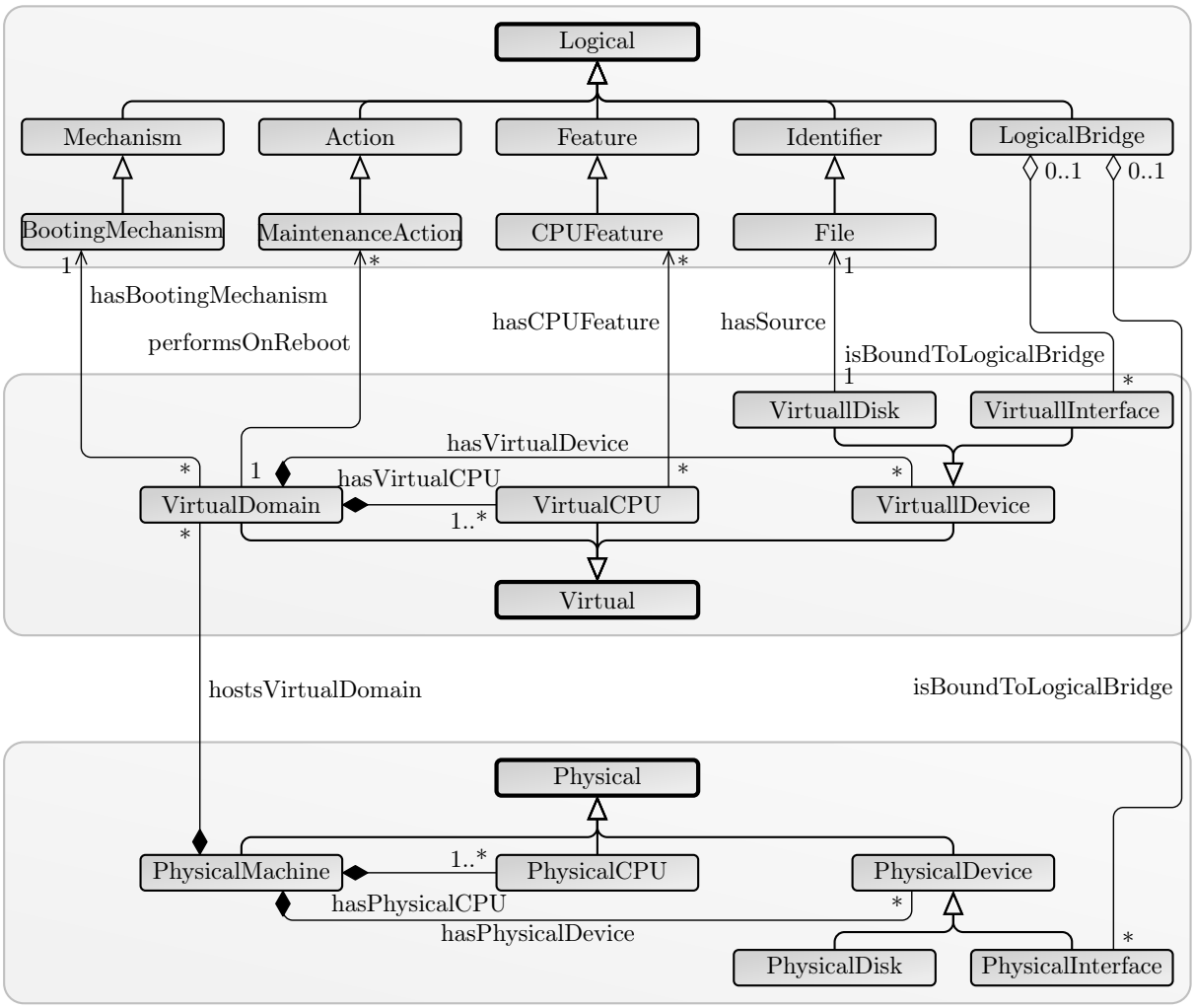


Figure 7.2: The ontology UML class diagram.

The *actions*, represented by the `Action` class hierarchy, are entities used to define what the system must perform when a particular event occurs. Actions effectively model event-driven reaction concepts. For example, the class `MaintenanceAction`, depicted in Figure 7.2, is used to instruct a domain what to do when a system reboot or shutdown is requested. This class contains several individuals, each one representing a single action, such as the `Restart` and the `Destroy` objects, that respectively represent a simple virtual machine reboot or a full domain shutdown.

The *mechanisms*, modeled by the `Mechanism` class hierarchy, specify how a particular goal should be accomplished. Both actions and mechanisms describe how to perform a job, but the first are event-driven, whereas the latter are event-independent. For example, the `BootingMechanism` class defines how to start a guest OS. It contains several individuals that are used to specify how the boot sequence must be accomplished, e.g., launching the domain boot loader or letting the hypervisor perform a direct kernel launch.

The *features*, mapped to the `Feature` class hierarchy, are used to model the characteristics of a physical or virtual object. More specifically, features are used to declare what a physical component supports and what a virtual entity requires. For example, the `CPUFeature` class describes both the physical and the virtual CPU features, allowing the precise determination of what a physical processor offers and, on the other hand, what a virtual processor needs. This

class includes several individuals such as the `SSEFeature` and the `TSCFeature` objects that respectively represent the Streaming SIMD Extensions and the TimeStamp Counter features.

The *identifiers*, depicted by the `Identifier` class hierarchy, are named tokens that uniquely designate an object. Since the IT world extensively makes use of a wide range of such concepts, this class hierarchy is remarkably broad, counting 36 descendant classes. Figure 7.2 displays, as an exemplification, the `FilePath` class that models a file pathname, a concept used for several purposes, such as to specify an image file for a virtual disk. Another particularly useful and flexible identifier is the *network host*, represented by the `HostId` class, which is used to specify a generic machine using a DNS name, a MAC address, or more frequently an IP address, modeled by the `IPAddress` class. Identifiers contain additional notions such as directory pathnames and bus addresses (PCI, USB, CCID, ...).

The logical layer also consists of the `LogicalBridge` class, which represents the concept of *logical bridge*, a core entity that is used to create virtual networks and to allow a domain to communicate with the outside world. Bridges are used to join together both physical and virtual NICs, connecting the two layers. Several virtual and physical objects, particularly the network interfaces, can use the `isBoundToLogicalBridge` object property to explicitly state their connection with a specific logical bridge. This class is crucial for a great variety of network analysis, as we show in Section 7.5.6.

Most of the virtual and physical layer classes are strongly dependent on the logical layer through several object properties such as the `hasAddress` property that specifies the address owned by an entity, e.g., the IP address of a physical or virtual network interface. Furthermore, several virtual devices can have the `hasSource` and the `hasTarget` properties, which respectively define the data-source (e.g., a path to an image file) and the target address, that is the location where a guest OS will find this data (e.g., the `hda` disk).

7.4.3 Virtual Layer

The *virtual layer* consists of all the components that are virtualized by a generic hypervisor, i.e., the virtualized hardware and the domain concept. It is mapped to the `Virtual` class hierarchy and a subset of its classes is shown in Figure 7.2.

One of the fundamental concepts is the notion of *domain*, represented by the `VirtualDomain` class. Since this class plays such a key role in our ontology, a great number of data and object properties can be attached to it. These attributes are used to link virtual machines to other virtual notions or to the physical world, relying on the objects provided by the logical layer.

Every domain has a set of *virtual CPUs*, mapped to the `VirtualCPU` class hierarchy. Virtual CPUs can be described in detail, for instance using the property `hasCPUFeature` that uses the `CPUFeature` class individuals to define the requirements needed by a particular virtual processor.

Similarly to virtual CPUs, *virtual devices* are modeled by the `VirtualDevice` class hierarchy which represents all the virtualized hardware that can be attached to a domain with the exception of the processors. Virtual devices are basically virtual peripherals such as disks, virtual sound and video cards, and generally everything that can be connected to a virtual bus (PCI, USB, CCID and so on). The virtual device structure is vast and heterogeneous since every piece of hardware is extremely specialized. Figure 7.2 displays the `VirtualInterface` class which models all the virtual network interfaces that a domain can use for communicating. The `VirtualInterface` class has several descendants that specialize the network interfaces, e.g., the `VirtualBridgeInterface` class represents a virtual network interface connected to a logical bridge. Figure 7.2 depicts also the `VirtualDisk` class which models

the virtual disks, i.e., hard disks, CD-ROM and floppy drives. This class has a special property called `reachesSource` that relates a device to its real data source; in Section 7.4.5 we demonstrate how to automatically infer this property in a remote storage scenario.

Another representative entity is the `VirtualPool` class, which describes the storage pools. A *storage pool* is a set of volumes, e.g., disk images, that can be used to implement several advanced storage techniques such as remote disks or backing stores.

Since `VirtualDomain` is the core class of our ontology, we briefly introduce several object properties that are used to connect it to other concepts. The `hasVirtualDevice` and `hasVirtualCPU` object properties are used to specify that a particular domain contains a set of virtualized CPUs and devices. These properties also have a number of sub-properties like the `hasVirtualInterface` and `hasVirtualDisk` properties, that respectively specify the virtual interfaces and disks owned by a domain. Moreover, virtual machines are related to the logical layer entities, for example, as stated in Section 7.4.2, the `hasBootingMechanism` property indicates how a domain should boot its operating system. Furthermore, for each virtual machine we can specify what to do when a reboot is requested; the `MaintenanceAction` that will be performed is defined by the `performsOnReboot` object property. Additionally, the `hasHypervisorType` property is used to refine the domain type, by defining the compatibility between a virtual machine and a specific hypervisor.

7.4.4 Physical Layer

The *physical layer* consists of all the tangible objects, i.e., the real hardware. Libvirt itself does not provide a description model of the real world, since this is outside its scope, but we felt that creating an ontology solely describing the virtual realm would not be very useful. In fact the virtualized hardware is strongly related to the real hardware and decoupling these two layers will severely limit the usability of our ontology for performing analyses. Therefore we created a hierarchical structure for the physical world, similar to the one described in Section 7.4.3 for the virtual layer. Its root class is named `Physical` and its internal architecture closely resembles the virtual layer one. Considering these similarities, we briefly discuss a selection of the most distinctive subclasses.

The `PhysicalMachine` class represents the *physical machines*, i.e., the real computers hosting zero or more domains. This class is the physical counterpart of the `VirtualDomain` class and, in a similar way, several object and data properties can be assigned to it in order to describe its configuration in detail, ranging from the owned hardware to the hosted virtual machines.

The `PhysicalCPU` class models the *physical CPUs*, i.e., the real processors of a physical machine. The supported features of the physical processor can be specified using the `hasCPUFeature` object property that points to the desired `CPUFeature` class individuals.

The `PhysicalDevice` class hierarchy maps all the *physical devices* apart from the processors. Figure 7.2 depicts the `PhysicalDisk` and `PhysicalInterface` classes which respectively represent the physical disks (hard disks, CD-ROM and floppy drives) and physical NICs. The latter possesses an object property named `isPhysicallyLinked` that is used to represent a physical connection, e.g., a cable, between two physical network interfaces.

A virtual machine can contain several virtual CPUs and devices. A physical one can be described in a similar manner but with a set of physical processors and devices. This can be done by specifying the object properties `hasPhysicalCPU` and `hasPhysicalDevice` which in turn have several sub-properties such as `hasPhysicalInterface` and `hasPhysicalDisk`, that are used to define the network interfaces and disks owned by a physical machine. Further-

more, the physical machines have the `hostsVirtualDomain` object property which represents its hosted domains. This property is particularly important because it links the physical layer to the virtual world. In addition, the `runsHypervisor` object property can be used to specify which hypervisor is running on a host.

7.4.5 Refinement of the Core Ontology

So far, we have discussed the structure of our ontology, the type of concepts it contains and their relations. Here we show how this ontology can be used to simplify the management of a virtual infrastructure and how the reasoning facilities provided by an OWL-DL reasoner can be exploited to this end.

In the ontology, freely available at `http://security.polito.it/ontology`, we have included all the classes defined in this section as children of the `ExampleViews` class. All the following listings use the Manchester OWL syntax [HPS09].

First, we can add new concepts to the ontology, to obtain a finer classification of the individuals and infer additional properties.

For instance, we defined the concept `MultiConnectedDomain` (Listing 7.1) that gathers all the domains which have more than one network interface and the `DomainWithRemoteDisk` (Listing 7.2) which describes all the domains that are connected to a remote disk, i.e. a disk shared by a remote host through a virtual pool.

```
VirtualDomain and (hasVirtualDevice min 2 VirtualInterface)
```

Listing 7.1: `MultiConnectedDomain` definition.

```
VirtualDomain and (hasVirtualDevice some
    (VirtualDisk and (hasSource some
        (FilePath and ( inverse (hasTarget) some VirtualPool))))))
```

Listing 7.2: `DomainWithRemoteDisk` definition.

Moreover, we added a number of object properties and enriched the definition of several existing ones, by adding axioms to increase their expressiveness. We added the `isLinked` relation as a super-property of `isBoundToLogicalBridge`, and defined it as symmetric and transitive. In this way we assert that if an interface is bound to a logical bridge then this interface is linked to that bridge and vice-versa and that all the network interfaces bound to the same bridge are linked to each other (since the `isLinked` property is transitive).

Finally by adding a property chain (shown in Listing 7.3), we show how it is possible to infer the connection among domains starting from their description, which includes their virtual network interfaces, and the description of the logical bridges of a physical machine.

```
hasVirtualInterface o isLinked o  inverse (hasVirtualInterface) ->
   isConnectedToDomain
```

Listing 7.3: `isConnectedToDomain` property chain.

Given this property chain, if there are two different domains each having a virtual network interface linked among them, then the two domains are connected. Subsequently, as for `isLinked`, we can state that `isConnectedToDomain` is transitive and symmetrical. In this case we assume that each virtual machine works as a network bridge and forwards the traffic

coming from each interface to all the others. Verifying such a property would require analyzing the internal network configuration of each domain and since this configuration is not handled by Libvirt, this is out of the scope of this work. Moreover we decided to be conservative from a security point of view, therefore we consider this kind of connection possible.

We now show how it is possible to infer additional information about the system, by adding simple logical rules expressed using the SWRL language [HPSB+04]. This method permits several analyses on the system without requiring the development of ad-hoc reasoning tools, thus ensuring great flexibility and expandability.

First we show how complex concepts can be defined to identify misconfigurations and anomalies. With the concept of `MachineHostingUnsupportedDomains`, asserted using the rule in Listing 7.4 we can identify a physical machine hosting a domain of a type while running an hypervisor of a different type, e.g., a XEN domain and a KVM hypervisor. Since such a domain would not be supported by the hypervisor, this situation represents an anomaly.

```
PhysicalMachine(?m), hostsVirtualDomain(?m,?dom),
   hasHypervisorType(?dom,?htype), runsHypervisor(?m, ?hyp),
   DifferentFrom (?hyp,?htype) -> MachineHostingUnsupportedDomain(?m)
```

Listing 7.4: SWRL rule 3, `MachineHostingUnsupportedDomains` definition.

Previously, we defined the `DomainWithRemoteDisk` concept, to gather all the domains connected to a remote storage device. When considering network connected storage it can be interesting to verify if the domain using it is able to access the network address to which the storage is attached. For this reason we defined the following rule that works in the following

```
hostsVirtualDomain(?mac, ?dom), hasVirtualDisk(?dom, ?vdisk),
   hasSource(?vdisk, ?file),  hasTarget(?pool, ?file),
   VirtualPool(?pool), hasSource(?pool, ?nas), hasAddress(?nas, ?ip)
 hasAddress(?pint2, ?ip), hasPhysicalInterface(?mac, ?pint1),
    isPhysicallyLinked(?pint1, ?pint2) -> reachesSource(?vdisk,
    ?pint2)
```

Listing 7.5: SWRL rule 4, `reachesSource` definition

way. We consider a physical machine *mac* that hosts the virtual domain *dom* which is using a remote disk *vdisk*. *vdisk* has as source the file *file* which is the target of a virtual pool *pool*. We check if the machine *mac* is connected, through a network connection, to the physical machine *nas* which is the source of *pool*. Here we made some simplifications, since reaching the network interface of the machine to which the disk is attached is not sufficient to state that the disk is working properly, but it is a necessary condition. A more accurate analysis would require checking the internal configuration of the physical machine.

7.5 A Unified Ontology for Verification

In this section we introduce our unified ontology that is suitable for modeling an IT system, particularly a cloud infrastructure, and its security properties.

Since security is a complex matter we will discuss it at different levels, splitting our security architecture into five main areas, each one with a different granularity, which allow us to describe a system in more or less detail. First we describe the physical and virtual layers of the

model which is suitable for representing a wide range of both real and virtual hardware. Then we discuss the software and service levels that can be used to model the feature offered by a server, last its security description level.

7.5.1 Physical Layer

The physical layer has been designed to describe the network containing the nodes which run the IT resources used to implement the services. It is the ontological counterpart of the P-SDL language described in Section 7.3.2.

7.5.2 Virtual Layer

To describe the virtual layer we included in our ontology the one described in Section 7.4, using the import feature offered by OWL language. In particular we use the virtual and logical elements to describe this layer. We note that there is a partial overlapping with the ontology derived from P-SDL, both for the physical elements and for virtual domains - besides a rich description capability for the physical layer, P-SDL provides a primitive support to describe virtual machines as well). We used the class equivalence of OWL to relate equivalent classes when applicable, thereby obtaining a coherent and more expressive description.

7.5.3 Software Layer

The software layer is used to connect the virtual to the physical and the service levels. In fact it can relate either a virtual domain to a physical machine that hosts it or a software to a service it provides. It is composed by a handful of classes (`Software`, `OperatingSystem`, `Hypervisor`), related to each other by means of object properties such as `runSoftware`, `hostsDomain` and `providesService`. Similarly as for the physical layer, this layer gets input from a P-SDL model.

7.5.4 Service Layer

The service layer describes the choreography between services with the aim to create a bridge between a WS-CDL description and OWL-S. The `Service` class is linked to the OWL-S classes by class equivalence or subsumption. For our security analysis, we are also interested in the interactions among service components. For this reason we modeled the `ServiceChannel` class, which can also be used to attach security annotations to each communication channel. We currently provide only a minimal ontology that relates WS-CDL to OWL-S, but we plan to go into further details to be able to gain from the expressiveness of both languages (for a comparative analysis between WS-CDL and OWL-S we refer to [CDMV09]).

7.5.5 Security Layer

To describe security related concepts, such as security goals and security mechanisms, we used the ontology described in [HSD07] which describes assets, threats, vulnerabilities and countermeasures. This ontology provides a general taxonomy for the security concepts, with the aim of becoming the reference ontology in this field. Additionally, security concepts are related using object properties. Resorting to logical reasoning it is possible to query the ontology to answer questions like: Which security goal is protected on stored data using encryption? Which attack

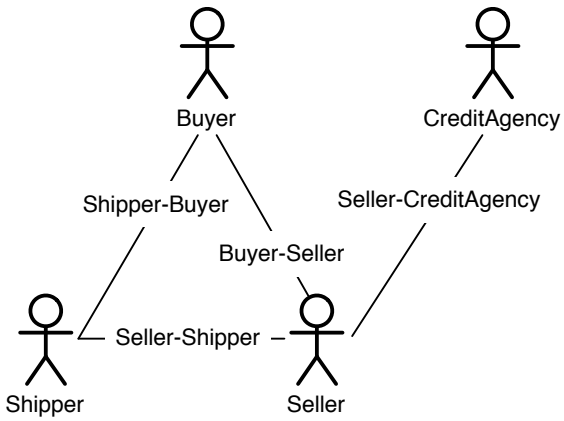


Figure 7.3: Choreography of the WS-CDL primer.

threaten data confidentiality? Security related concepts can be connected to the other layers using object properties such as `hasSecurityGoal` or `usesCountermeasure`.

7.5.6 A Detailed Example

To demonstrate the validity of our approach we present an example service running on a physical network where some security mechanisms are in place. We then consider confidentiality as a security property to be guaranteed among the service components and present the result of the analysis.

Service Layer

As a reference example, we consider the WS-CDL Primer [RTF06]. This choreography describes a good purchasing scenario, whose roles and relationships are depicted in Figure 7.3. For a detailed description of the choreography and a complete WS-CDL listing we refer to [RTF06].

The roles in the choreography (Buyer, Seller, CreditAgency and Shipper) are implemented by real services which are composed by a set of software components. We present here the complete list of components implementing each role:

- the Buyer (e.g., a private citizen) communicates through a web-browser;
- the Seller (e.g., a company) uses a web-server as a front-end and a database server which contains a catalog of the products to be sold. In addition to this it has a backup server which stores a copy of the database;
- the CreditAgency uses 3 different servers to manage the transactions, these servers provide the same functionalities and are mirrored for dependability purposes;
- the Shipper has its own server from which it manages shipping orders.

Network and Virtual Layers

The network where the example choreography is instantiated is shown in Figure 7.4. This represents a (simplified) hosting provider with a network distributed in two locations, connected through the Internet. In addition to this a personal computer is connected to the Internet to represent a private citizen accessing the service from her home.

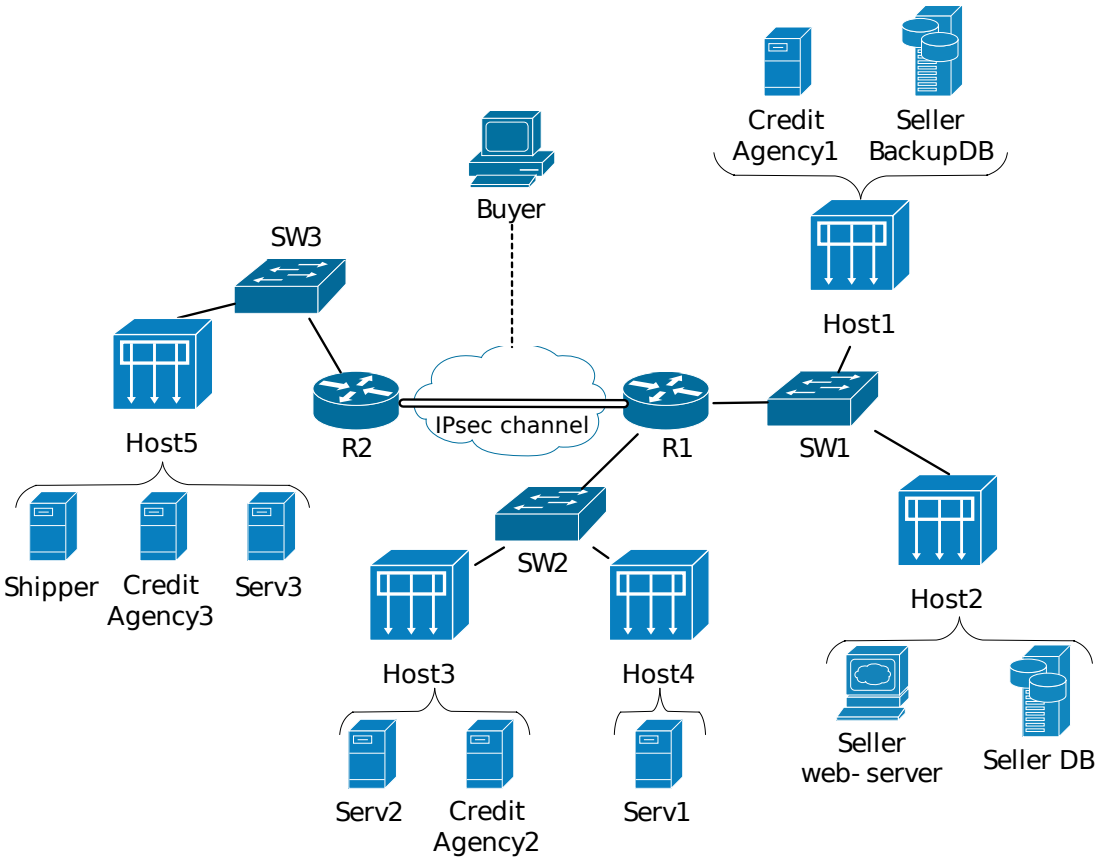


Figure 7.4: Network level view of the primer.

In more detail, the provider network is partitioned into three layer-2 networks identified by the switches SW1, SW2 and SW3. SW1 and SW2 are connected via a router R1, while SW3 is connected to a router R2. An IPsec tunnel is available for communications between R1 and R2 (we assume they are connected through the Internet). Several hosts are attached to the switches, and each host is able to run virtual machines.

The software components described in Section 7.5.6 are allocated in this way:

- the Buyer web browser runs on the personal computer;
- the Seller web-server and the Seller DB run in two different virtual machines on Host2; the Seller BackupDB runs in a virtual machine on Host1.
- the CreditAgency servers run in different virtual machines, respectively on Host1, Host3 and Host5.
- the Shipper server runs in a virtual machine on Host5.

In addition, other services (Serv1, Serv2, Serv3) run in virtual machines on the same hosts, to emphasize the multi-tenant architecture typical, e.g., of a cloud computing environment.

Security Analysis

In [HSD07], the authors show, among other examples, the `CountermeasureByConfidentialityOfData` class, that classifies all the countermeasures protecting confidentiality of data. With our ontology, it is possible to make a step further, and to retrieve the *instances* of these

countermeasures, i.e. the actual countermeasures available in the system. In the example presented before, the same query presented in [HSD07] returns the IPsec channel between routers R1 and R2.

As another example, we can extract from the model all the assets where, e.g., eavesdropping may happen. Through the security ontology the reasoner infers the `DataInTransit` class and, exploiting the other layers in the whole ontology, it deduces all the channels between services and all the network segments underlying each service channel. For instance, the service channel between Seller web-server and CreditAgency1 (as well as the network segments Host2–SW1 and SW1–Host1) is subject to eavesdropping. We note that, by construction, the channels between two virtual machines in the same host (e.g., between Seller web-server and Seller DB) are not liable for eavesdropping, as the hypervisor protects such virtual channels. Similar considerations apply to query the model for all the assets that may be protected by a specific countermeasure, e.g., IPsec.

We conclude with a remark on the allocation of the Seller. As the Seller is allocated onto 3 servers, but there is no explicit description of the behavior and the interactions among them, we perform a worst-case analysis and assume all communications may happen. Therefore, we shall have 3 distinct communication channels respectively from Seller web-server to Seller DB, from Seller DB to Seller BackupDB and from Seller web-server to Seller BackupDB. The latter is probably useless, as the web-server probably will not directly communicate with the backup DB. However this can not be deduced with the level of details provided by this model.

7.6 Conclusions and Final Remarks

This chapter discussed the role of ontologies in a virtualized environment, focusing on a security related point of view. The report has presented a general overview of what ontologies are and their advantages. Then details about an ontology for describing a virtualized environment were illustrated, providing also a number of examples which allow several analyses of an IT infrastructure. Then, a more complete ontology was presented, allowing not only to describe virtual systems, but also physical ones and their security features. Furthermore, a real example scenario was provided which showed how ontologies can be successfully used to perform security analyses of a virtual network.

The previous sections showed how the expressive power of the ontologies can be applied in the application domain of virtualized systems, allowing to describe complex virtual infrastructures using a more natural, easier and powerful formal language system compared to traditional database languages such as SQL. They also proved that the reasoning facilities of this technology can be used to easily perform several analyses of a system, in particular security analyses, and consistency checking of configurations in an automatic way, thus simplifying the administrators' job.

Chapter 8

Automated Information Flow Analysis of Virtualized Infrastructures

Chapter Authors:
Sören Bleikertz, Christian Cachin, Thomas Groß, Matthias Schunter (IBM)

8.1 Introduction

Large-scale virtualized infrastructures and cloud deployments are a common and still growing phenomenon. The goals of server virtualization include high utilization of today's hardware, fast deployment of new virtual machines and load balancing through migration of existing virtual machines. Virtualized infrastructures provide standardized computing, virtual networking, and virtual storage resources. Correspondingly, infrastructure clouds provide simple machine creation and migration mechanisms as well as seemingly unlimited scalability, while the costs incurred are only proportional to the resources actually used.

The growth of IT infrastructures and the ease of machine creation have lead to substantial numbers of servers being created (server sprawl). Furthermore, then led to large and complex configurations that arise by rank growth and evolution rather than by advance planning and design. Indeed, the configuration complexity often exceeds the analysis and management capabilities of human administrators. We depict an example for a mid-size infrastructure in Figure 8.1. This, by itself, calls for automated security analysis of virtualized infrastructures. The high complexity of an analysis is amplified when considering security properties such as isolation, because then the analysis of individual resources must be complemented with an analysis of their composition.

In addition, virtualization providers often aim at establishing multi-tenancy, that is, the capability to host workloads from different subscribers on the same infrastructure. Also, they provide an open environment, in which arbitrary subscribers can register without trust between them being justified. Therefore, we need to assume that workloads as well as VMs are under the control of an adversary, and that an adversary will use overt and covert channels in its reach.

Industry partially approaches isolation with automated management and deployment systems constraining the users' actions. However, these mechanisms can fail, may lack enforcement, or can be circumvented by human intervention.

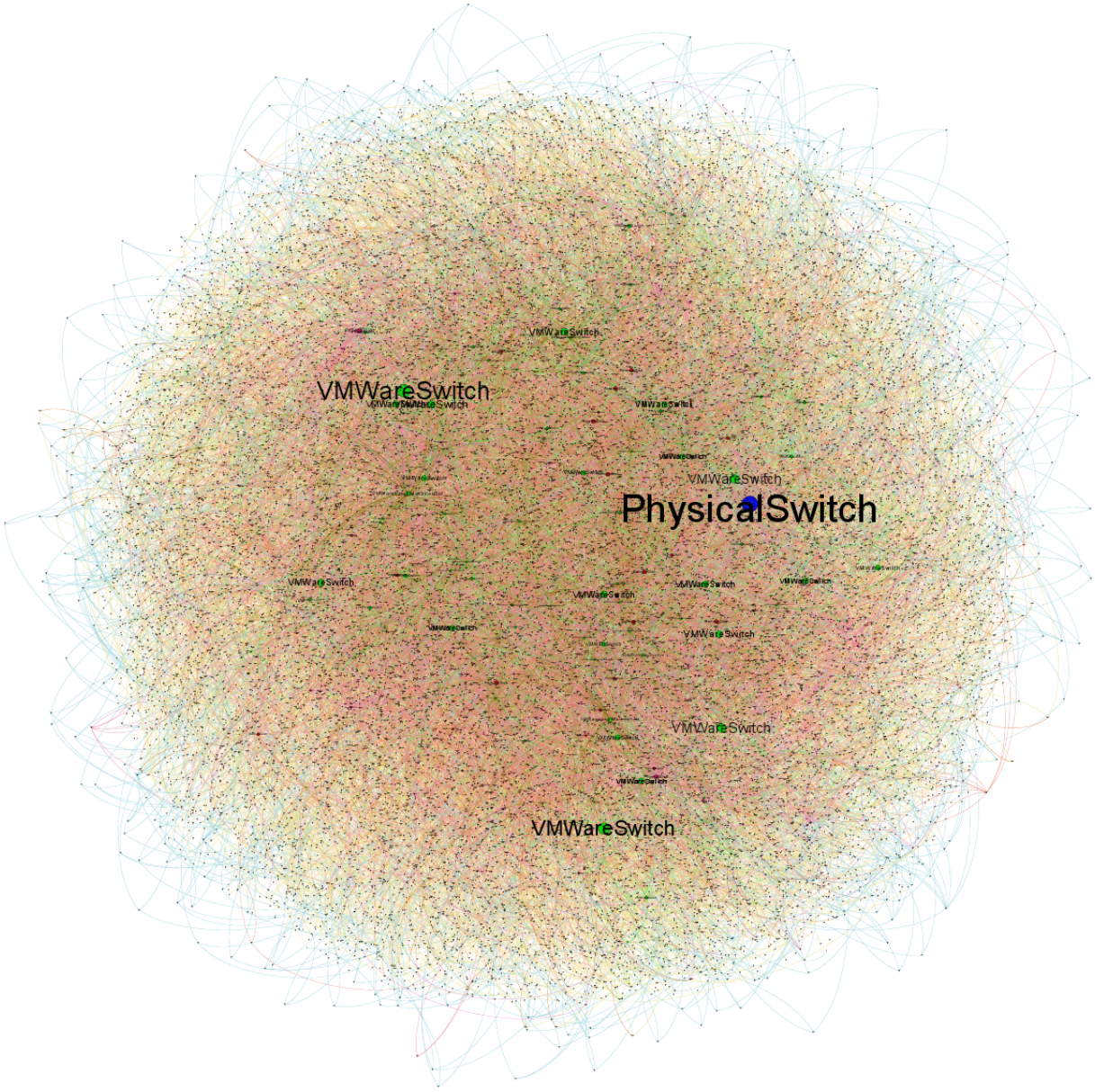


Figure 8.1: Illustration of the overwhelming complexity of a mid-size infrastructure with 1,300 VMs.

8.1.1 Contributions

The aim of this work is to automate information-flow analysis for large-scale heterogeneous virtualized infrastructures. We aim at reducing the analysis complexity for human administrators to the specification of a few well-designed trust assumptions and leave the extrapolation of these assumptions and analysis of information flow behavior to the tools.

We propose an information flow analysis tool for virtualized infrastructures. The tool is capable of discovering and unifying the actual configurations of different virtualization systems (Xen, VMware, KVM, and IBM's PowerVM) and running a static information flow analysis based on explicitly specified trust rules. Our analysis tool models virtualized infrastructures faithfully, independent of their vendor, and is efficient in terms of absence of false negatives as well as adjustable false positive rates.

Our approach transforms the discovered configuration input into a graph representing all resources, such as virtual machines, hypervisors, physical machines, storage and network resources. The analysis machinery takes a set of graph traversal rules as additional input, which models the information flow and trust assumptions on resource types and auxiliary predicates. It checks for information flow by computing a transitive closure on an information flow graph coloring with the traversal rules as policy. From that, the tool diagnoses isolation breaches and provides refinement for a root causes analysis. The challenge of information flow analysis for virtualized infrastructures lays in the faithful and complete unified modeling of actual configurations, a layered analysis that maintains completeness and correctness through all stages, and a suitable refinement to infer the root causes for isolation breaches.

Our method applies strict over-abstraction to minimize false negatives. This means that we only assume absence of flows for components that are known to isolate. This enables us to reduce the analysis correctness to the correctness of the traversal rules. As this method accepts an increase in the false positive rate, we allow administrators to fine-tune the trust assumptions with additional traversal rules and constraint predicates to obtain a suitable overall detection rate.

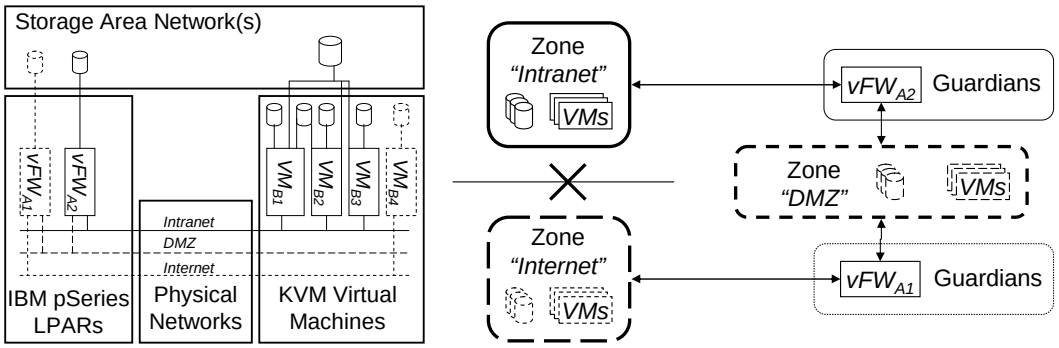
We report on a case study for a mid-sized infrastructure of a financial institution production environment in Section 8.7.

8.1.2 Applications

Our technique is applicable to the isolation analysis of complex configurations of large virtualized datacenters. Such datacenters include different types of server hardware, implementations of virtual machine monitors, as well as physical and virtual networking and storage resources.

Let us consider a simplified version of such a configuration in Figure 8.2(a). This simplified version includes the following hardware: A IBM pSeries server, an x86 server, a virtual networking infrastructure providing VLANs, and a Storage-Area Network (SAN) providing virtual storage volumes. The virtual resources (networks, storage, machines, and virtual firewalls) are depicted inside these hardware resources. Keep in mind that sizeable real-world configurations contain thousands of virtual machines and hundreds of thousands of connections.

Figure 8.2(b) depicts a desired isolation topology for this example: we have three example virtual security zones "Intranet", "DMZ", and "Internet". Furthermore, we permit communication between Intranet and DMZ that is mediated by a trusted guardian, such as a virtual firewall vFW_{A2}. Similarly, firewall vFW_{A1} moderates and restricts the communication between the DMZ and Internet zones, respectively. The isolation analysis must check that there do not exist components that connect two zones or are shared by two zones, while not being trusted to



(a) Architecture Overview (b) Isolation Zones

Figure 8.2: An example setup of a virtualized datacenter with an isolation policy for three virtual security zones.

sufficiently mediate direct and covert information flows.

Note that we focus on validating the virtualized infrastructure's configuration. Once we have guaranteed that no undesired information flow exists except through the specified guardians, we would need to employ techniques from firewall filtering analysis, e.g. [MK05, MWZ00, Woo01], to ensure that the guardians have been configured correctly.

8.2 Related Work

Virtual systems introduce several new security challenges [GR05]. Two important drivers that inspired our work are the increase of scale and the transient nature of configurations that render continuous validation more important.

The first area of related work is security of virtual machine monitors. This knowledge is needed to underpin the user's individual decisions whether to trust a given component. Analysis of well-known attacks such as jailbreaks [Woj08] allows one to detect vulnerable configurations. This includes information leakage vulnerabilities of today's infrastructure clouds that allow covert or overt communication between multiple tenants that should be isolated. Examples include co-hosting validation [RTSS09b] and cache-based side channels [Aci07, Per05].

To our knowledge, there do not exist any research contributions on the static high-level information flow analysis in virtualized infrastructures. Still, we draw inspiration from information flow analysis such as research in separation [KF91, Rus82], channel control [Rus81], and non-interference [GM82, Gra91, Man01, Rus92]. We discuss these influences on our own definition on structural information control in Section 8.3. More often than not, we find research in this space focused on the information flow between high and low variables and not on the information flow in larger topologies.

A second area of related work is *reachability analysis* in networks and the related configuration analysis of firewalls. Our isolation analysis draws from known work on reachability analysis. Analyzing firewalls and complex network infrastructures allows one to decide to what extent two known networks are connected. In particular, Al-Shaer *et al.* [ASMEAE08, KL09, XZM+05] analyze entire network infrastructures including packet filters, transformers, and routers. In [BSP+10], it was shown how reachability analysis can be applied to infrastructure clouds. *Firewall configuration analysis* allows the understanding and validation of firewall rules [MK05, MWZ00, Woo01]. While this work focuses on the TCP/IP level, our goal is to ensure "physical" isolation by ensuring that VLANs and virtual networks are disjoint. This ap-

proach is similar to the approach proposed in [KSS+09]. If media-layer networks are connected while isolation is implemented on the TCP/IP level, we see potential to further extend our work by using these concepts for TCP/IP isolation analysis that is then fed into our analysis concept.

This area of research is also important for modeling the behavior of imperfect guardians. Whereas this paper assumes that guardians always make correct decisions and stop dangerous information flow, reachability and firewall configuration analyses allow one to model imperfections as explicit traversal rules. Guardians with packet inspection and stateful analysis may even discover illegal information flow hidden in legal flows.

Whereas earlier security analyses considered stand-alone elements of a virtualized infrastructure, a tool-supported information flow analysis of a full virtualized environment is still missing, not to speak of complex heterogeneous and large-scale virtualized infrastructures with a diversity of underlying platforms. The research areas of information flow and reachability analysis underpin our efforts, yet so far, they have not produced a mechanized approach for this problem statement.

8.3 A Model for Isolation Analysis

8.3.1 Flow Types

In the quest for a suitable requirements definition, we now review information flow types [SM03, Lam73, RTSS09b, GM82, Gra91, Rus92, Man01, HY86, Rus82, KF91, Jac90].

We consider *overt* and *covert* channels. *Covert channels* [Lam73] are not intended for information transfer at all, yet seem to be a common phenomenon in virtualized infrastructures. Requiring the absence of all covert channels from hypervisors, physical hosts and resources will render many resulting system impractical. Therefore, we allow administrators to specify a certain amount of covert channel information flow as tolerable.

An *overt channel* is intended for communication; a principal can read or write on that channel within the limits of some access control policy.

Lampson [Lam73] introduced the term *covert channel* as a channel not intended for information transfer at all. Consider a malware in VM Alice which attempts to transfer information to another instance of the malware in VM Bob, both hosted on the same hypervisor. The malware on VM Alice can, for instance, monopolize a resource[1] to transmit a bit observed by the malware on VM Bob in performance or throughput decrease.

We perceive covert channels to be a common phenomenon in virtualized infrastructures. Requiring the absence of all covert channels from hypervisors, physical hosts and resources, will render many resulting system impractical. Therefore, we allow administrators to specify a certain amount of covert channel information flow as tolerable.

Requirement Definition

We informally stated our security goal as *isolation* between zones, which sounds similar to *non-interference* [GM82, Gra91]. This requirement enforces that actions in one zone do not have any effect on subsequent behavior or outputs in another zone.

The transitivity of non-interference renders it, however, unsuitable to model our setting, in which information flow via guardians may be permitted, whereas the corresponding direct flow

[1] Examples include reserving a bus, launching expensive computations, flooding a cache, sending many network packets.

is disallowed. Agreeing to the arguments of Rushby [Rus92] and Mantel [Man01], we would need *intransitive non-interference* [HY86] to start with. Furthermore, the existing definitions are based on traces of steps and, thus, inherently dynamic[2], whereas we aim at a high-level static information flow analysis (its topology and communication links). Therefore, we preclude a step/trace-based non-interference analysis.

Another candidate is the analysis for *separation*, e.g. [Rus82, KF91]: one removes all guardians from the system and verifies that the remaining parts are perfectly separated; however this approach was criticized by Jacob [Jac90].

The concept of *channel control* [Rus81] sounds interesting, as it captures our requirement to specify *exceptions* to the general zoning requirements. For instance, two zones should not communicate with each other *unless* a guardian mediates and filters the communication. In our case, however, we are not studying single channels, but a complex topology of channels.

Information Control

We note (1) that we need intransitivity and (2) that *channel control* [Rus81] captures our requirement to specify *exceptions* to the general zoning requirements. Thus, we introduce a property we call *structural information control* that essentially lifts channel control to topology:

Definition 1 (Structural Information Control) *A static system topology provides* structural information control *with respect to a set of information flow assumptions on system nodes if there does not exist an inter-zone information flow unless mediated by a dedicated guardian.*

Observe that we aim at the detection of isolation breaches (information flow traces), which renders our approach loosely similar to model checking, and not at the verification of absence of information flow, which would be similar to theorem proving.

8.3.2 Modeling Isolation

Modeling Configurations

Our static information flow analysis is graph-based. Each element of a virtualization configuration is represented by (at least) one vertex (VMs, VM hosts, virtual storage, virtual network). Connections between elements are represented by edges in the graph and model *potential* information flow. Note that our approach requires completeness of the edges: While not all edges may later actually constitute information flows, we require that all relations that allow information flow are actually modeled as an edge.

The vertices of the graph are typed: our model distinguishes VM nodes, VM host nodes, storage and network nodes, etc.

Definition 2 (Graph Model) *Let $\mathbb{T} \subset \Sigma^+$ a set of vertex types and $\mathbb{P} \subset \Sigma^+$ a set of vertice properties. A virtualization graph model contains a set of typed vertices $V \subset \mathbb{V} := (\Sigma^+ \times \mathbb{T} \times \mathbb{P})$ and a set of edges $E \subseteq (V \times V)$. A vertice v is a triple of label, type and properties set $(l, t, p) \in \mathbb{V}$. An edge e is a pair of start and end vertice $(v_i, v_j) \in (V \times V)$. A set of edges E is called* valid *with respect to a set of vertices V', if $E \subseteq (V' \times V')$. A graph $(\bar{V}, \bar{E})$ is called a* valid subgraph *of graph (V, E), if $\bar{V} \subseteq V$ and $\bar{E} \subseteq E$ is valid with respect to $\bar{V}$. An edge set $\vec{E} \subseteq E$ is called a* path *if the edges and their respective vertices form a connected valid sub-graph of (V, E).*

[2] To that end, Haigh and Young [HY86] have shown that it is necessary to analyze the complete trace of actions subsequent to a given action to validate that the action is allowed to interfere with another zone.

We represent complex structures of the virtualization infrastructure by sub-graphs of multiple vertices. For instance, we construct guardians such as firewalls with complex information flow rules by a firewall vertex connected to multiple port vertices.

Information is output at one or more *information source* nodes, propagates according to *traversal rules* along the nodes and edges of the graph, and is consumed at an *information sink*. We treat information sources as independent and information as untyped and unqualified.

Definition 3 (Information Sources and Sinks) *For a set of vertices V, we define a set of information sources $\hat{V} \subseteq V$ and a set of information sinks $\check{V} \subseteq V$. A vertice $\hat{v} \in \hat{V}$ is called* information source*, a vertice $\check{v} \in \check{V}$* information sink.

Modeling Information Flow Assumptions

A *traversal rule* models an assumption on information flow from one vertex type to another vertex type. For instance, a traversal rule will specify that if a VM host is connected to a storage provider, this edge constitutes a direct information flow and is to be traversed. Also, a traversal rule may specify that if two VMs are connected to the same VM host, this implies the risk of covert channel communication and, therefore, constitutes an information flow.

Definition 4 (Traversal Rules) *For the set of vertice types $\mathbb{T} \subset \Sigma^+$ and a set of vertice properties $\mathbb{P} \subset \Sigma^+$, the* traversal rules *are a propositional function of source type, destination type, source properties, and destination properties over a type relation R and a predicate P:*

$$f_{\mathbb{T},\mathbb{P}} : \quad (\mathbb{T} \times \mathbb{T} \times \mathbb{P} \times \mathbb{P}) \to \{\mathsf{stop}, \mathsf{follow}\} :$$

$$f_{\mathbb{T},\mathbb{P}}(t_i, t_j, p_i, p_j) := \begin{cases} (t_i, t_j) \in R \wedge P(p_i, p_j) & \mathsf{follow} \\ (t_i, t_j) \notin R \vee \neg P(p_i, p_j) & \mathsf{stop} \end{cases}$$

We call traversal rules simple*, if P is always true.*

Definition 5 (Completeness) *For the set of vertice types $\mathbb{T} \subset \Sigma^+$ and a set of vertice properties $\mathbb{P} \subset \Sigma^+$, traversal rules $f_{\mathbb{T},\mathbb{P}}$ are called* complete *if R and P associated to $f_{\mathbb{T},\mathbb{P}}$ are complete. We call a default rule a* completion *of incomplete traversal rules $f_{\mathbb{T},\mathbb{P}}$, if it maps all undetermined cases to either* stop *or* follow*. We call non-default rules* explicit*.*

Whereas completeness is a property of a set of traversal rules, we define *coverage* as in how far a set of traversal rules determines the analysis of a graph deterministically without invoking the default rule.

Definition 6 (Coverage) *For the set of vertice types $\mathbb{T} \subset \Sigma^+$ and a set of vertice properties $\mathbb{P} \subset \Sigma^+$, consider a virtualization graph (V, E) as in Def. 2 and the subset of edges $E' \subseteq E$ that are matched by explicit traversal rules $f_{\mathbb{T},\mathbb{P}}$. We call the quotient of number of explicitly matched edges to total number of edges* coverage*: $c = |E'| \,/\, |E|$*

Observe that a complete coverage, that is, $c = 100\%$ is important for achieving a low false-positive rate.

The traversal rules specify general assumptions on information flow in virtualized environments and, thereby, embodies a part of the overall trust assumptions. The specification of traversal rules is therefore orthogonal to the isolation policy of a system. Whereas our system comes with a root set of traversal rules as base line trust assumptions, we allow users to specify multiple sets of *user-defined traversal rules* and thereby *user-defined trust assumptions*.

Similar to the tainted variable method for static information flow analysis, we employ the metaphor of color propagation. We associate colors to information sources $\hat{v} \in \hat{V}$ and to vertices that have received information flow from a certain source by the evaluation of traversal rules $f_{\mathbb{T},\mathbb{P}}$. The total information flow of a system is the *transitive closure* of the graph traversal governed by the traversal rules $f_{\mathbb{T},\mathbb{P}}$. This means, that the information flow from any source to any sink can be efficiently statically analyzed by a reachability analysis between source and sink.

We define graph coloring recursively.

Definition 7 (Graph Coloring) *Let traversal rules $f_{\mathbb{T},\mathbb{P}}$, a graph (V, E) and an information source $\hat{v} \in \hat{V} \subseteq V$ with color c be given. Then, $\hat{v}$ is* colored *with c by definition. A vertice $v \in V$ is* colored *with c, if there exists an edge $e = (\cdot, v) \in E$, which is* colored *with c. An edge $e = (v_s, v_d) \in E$ with $v_s = (\cdot, t_s, p_s)$ and $v_d = (\cdot, t_d, p_d)$ is* colored *with c iff (i) v_s is* colored *with c and (ii) $f_{\mathbb{T},\mathbb{P}}(t_s, t_d, p_s, p_d) =$* follow.

8.4 Isolation Analysis of Virtual Infrastructures

We apply the foundations from the preceding section to virtualized infrastructures. Our approach (see Figure 8.3) consists of four steps organized into two phases: 1) building a graph model from platform-specific configuration information and 2) analyzing the resulting model. The graph model is formally defined in Def. 2.

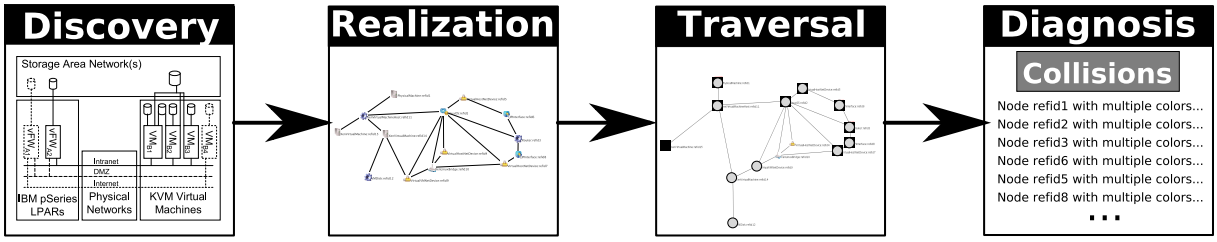


Figure 8.3: Overview over the analysis flow.

The first phase of building a graph model is realized using a discovery step that extracts configuration information from heterogeneous virtualized systems, and a translation step that unifies the configuration aspects in one graph model. For the subsequent analysis, we apply the graph coloring algorithm defined in Def. 7 parametrized by a set of traversal rules and a zone definition. The assessment of the resulted colored graph model enables a diagnosis of the virtualized infrastructure with respect to isolation breaches.

8.4.1 Discovery

The goal of the discovery phase is to retrieve sufficient information about the configuration of the target virtualized infrastructure. To this end, platform-specific data is obtained through APIs such as VMware VI, XenAPI, or libVirt, and then aggregated in one discovery XML file. The target virtualized infrastructure, for which we will discover its configuration, is specified either as a set of individual physical machines and their IP addresses, or as one management host that is responsible for the infrastructure (in the case of VMware's vCenter or IBM pSeries's HMC). Additionally, associated API or login credentials need to be specified.

For each physical or management host given in the infrastructure specification, we will employ a set of discover probes that are able to gather different aspects of the configuration. We realized multiple hypervisor-specific probes for Xen, VMware, IBM's PowerVM, and LibVirt. Furthermore, if the management VM is running Linux, we also employ probes for obtaining Linux-specific configuration information. Currently, we do not discover the configuration of the physical network infrastructure. However, the framework easily be extended beyond the existing probes or use configuration data from a third-party source.

8.4.2 Transformation into a Graph Model

We translate the discovered platform-specific configuration into a unified graph representation of the virtualization infrastructure, the *realization model*. The realization model is an instance of the graph model defined in Def. 2. It expresses the low-level configuration of the various virtualization systems and includes the physical machine, virtual machine, storage, and network details as vertices. We generate the realization model by a translation of the platform-specific discovery data. This is done by so-called *mapping rules* that obtain platform-specific configuration data and output elements of our cross-platform realization model. Our tool then stitches these fragments from different probes into a unified model that embodies the fabric of the entire virtualization infrastructure and configuration.

For all realization model types in $\mathbb{T}$ (cf. Def. 8), we have a mapping rule that maps hypervisor-specific configuration entries to the unified type and, therefore, establishes a node in the realization model graph. We obtain a complete iteration of elements of these types as graph nodes. The mapping rules also establish the edges in the realization model.

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

8.4.3 Coloring through Graph Traversal

The graph traversal phase obtains a realization model and a set of information source vertices with their designated colors as input. According to Def. 7, the graph coloring outputs a colored realization model, where a color is added to a node if permitted by an appropriate traversal rule. We use the following three types of traversal rules (see Def. 4 and the definition of traversal rules below) that are stored in a ordered list. We apply a first-matching algorithm to select the appropriate traversal rule for a given pair of vertices.

Flow rules model the knowledge that information can flow from one type of node to another if an edge exists. For example, a VM can send information onto a connected network. These rules model the "follow" of Def. 4. *Isolation rules* model the knowledge that certain edges between trusted nodes do not allow information flow. For example, a trusted firewall is known to isolate, i.e., information does not flow from the network into the firewall. These rules model the "stop" of Def. 4. *Default rule* means that ideally, either isolation or else flow rules should exist for all pairs of types and all conditions, that is, we want to achieve complete coverage according to Def. 6: For any edge and any two types, the *explicit traversal rules* should determine whether this combination allows or disallows flow. In practice, the administrator may lack knowledge for certain types. As a consequence, we included a default rule as *completion*. Here, we establish a *default flow rule*: whenever two types are not covered by an isolation or flow rule, then we

default to "follow". To be on the safe side, i.e., reducing false negatives, we assume that flow is possible along this unknown type of edges.

Given this set of rules, we then traverse the realization model by applying the set of traversal rules and color the graph according to information flows from a given source. The traversal starts from the information sources and computes the transitive closure over the traversal rule application to the graph.

8.4.4 The Traversal Rules

The graph coloring algorithm requires a set of traversal rules that model information flows, isolation properties, and trust assumptions. We will propose a set of rules and explain their purposes, and defer the correctness argument and a detailed discussion to Section 8.5.2.

Definition 8 (Traversal Rule) *Let F be a set of follow types $\{\mathsf{stop}, \mathsf{follow}\}$, $\mathbb{T}' \subset \mathbb{T}$ be a set of realization model types {Port, NetworkSwitch, PhysicalSwitch, ManagementOS, PhysicalDevice, VirtualMachine, VirtualMachineHost, StorageController, PhysicalDisk, FileSystem, File, any}, and D be a set of flow directions $\{\Rightarrow, \Leftarrow, \Leftrightarrow\}$, where $\Rightarrow$ and $\Leftarrow$ denote a unidirectional, and $\Leftrightarrow$ a bi-directional flow. A traversal rule is a tuple (f, t_0, t_1, d, P, g) with $f \in F$, $t_0, t_1 \in \mathbb{T}'$, $d \in D$, P is a predicate over properties and colors of the realization model, and g is a color modification function. During graph coloring (see Def. 7), g can transform the color c of a colored vertex $\hat{v}$ while coloring a new vertex v, i.e., $c(v) = g(c(\hat{v}))$.*

The traversal rules specified in Table 8.1 are a ordered list of rules (as defined in Def. 8). In case the condition is left empty, a $true$ predicate is assumed, and in case the color modification is empty, g is the identity function.

Definition 9 (Matching Rule) *Given a traversal rule (f, t_0, t_1, d, P, g) as defined in Def. 8 and a source and destination vertex from the graph traversal: v_s and v_d respectively. The rule matches iff i) $(t_0 = t(v_s) \vee t_0 = any) \wedge (t_1 = t(v_d) \vee t_1 = any)$ where $t(v)$ denotes the type of a given vertex v, ii) $d \in \{\Rightarrow, \Leftrightarrow\}$, iii) $P = true$.*

The first-matching algorithm iterates over the ordered list of traversal rules (Table 8.1) and applies the matching rule defined in Def. 9. If the matching evaluates to true, the iteration stops and the matched rule is returned. The matching of the traversal rules induces a function representation of the traversal rules as defined in Def. 4.

Our trust assumptions are specified in the rules namely, Rule 1, Rule 2, Rule 3, and Rule 4. These model that VLANs are isolated on physical switches, that the privilege VM and the physical machine are trusted and do not leak information, and that we exclude cross-VM covert channels (see Section 8.5.2).

Rule 5 simply stops an information flow if a network port is disabled. Rule 6 and Rule 7 model the VLAN en- and decapsulation of network traffic. Traffic with a VLAN tag is modeled as a new color *vlan* with the VLAN tag appended, which is created in case of encapsulation and removed in case of decapsulation. In the case of VMware, the VLAN tag for a VM is modeled as a non-zero *defaultVLAN* property of the port. Rule 8 specifies that if a port is marked as trunked, which is required in the case of VMware to allow traffic from the VMs to the physical network interface, the VLAN traffic is also allowed to flow. Otherwise, if the *vlan* color tag mismatches the port's VLAN tag, we isolate and stop the information flow (see Rule 9). This also applies to Rule 10, which is the default isolation rule for VLAN traffic, if one of the previous rules did not match.

Table 8.1: Traversal Rules

#	Type	Flow	Condition +Color Modification
		Trust Rules	
1	stop	$PhysicalSwitch \Rightarrow Port$	Has any *vlan* color
2	stop	$ManagementOS \Leftrightarrow any$	
3	stop	$PhysicalMachine \Leftrightarrow PhysicalDevice$	
4	stop	$VirtualMachine \Leftrightarrow VirtualMachineHost$	
		Network Switches	
5	stop	$Port \Leftrightarrow NetworkSwitch$	Port is disabled
		VLAN	
6	follow	$Port \Rightarrow NetworkSwitch$	Port has VLAN tagging with tag $VLAN + Create *vlan-$VLAN*
7	follow	$NetworkSwitch \Rightarrow Port$	Port's VLAN tag matches color's one +Remove *vlan-$VLAN*
8	follow	$NetworkSwitch \Rightarrow Port$	Port is trunked
9	stop	$NetworkSwitch \Rightarrow Port$	Port's VLAN tag mismatches color's one
10	stop	$NetworkSwitch \Rightarrow Port$	Has any *vlan* color
		Storage	
11	stop	$StorageController \Rightarrow PhysicalDisk$	
12	stop	$FileSystem \Rightarrow File$	
		Default	
13	follow	$any \Leftrightarrow any$	

On the storage side, we model the behavior of the storage controller not to leak information from one disk to another with Rule 11. Furthermore, the filesystem will not leak information from one file to another (Rule 12).

The default rule Rule 13 allows any information flow that was not handled by a previous rule due to the first-matching algorithm.

We make three observations about the traversal rules: *First*, administrators can modify existing and specify further traversal rules, for instance, to relax trust assumptions or to model known behavior of specific components. *Second*, traversal rules serve as generic interface to include analysis results of other information flow tools into the topology analysis (e.g., firewall information flow analysis). *Third*, the behavior of explicit guardians (see Def. 1) is introduced by traversal rules specific to these nodes. For instance, the guardians in the exemplary Figure 8.2, Section 8.1.2, would receive a stop-rule.

8.4.5 Detecting Undesired Information Flows

The goal of the detection phase is to produce meaningful outputs for system administrators. For detecting undesired information flows, we color a set of information sources that mark types of critical information that must not leak. The idea of the color spill method is to introduce nodes called "sinks" (see Def. 3). Each sink is colored with a subset of the colors corresponding to the information that it is allowed to receive. In practice, the administrator provides a list of clusters or zones that shall be isolated, and we add/mark sources and sinks according to the isolation policy with respect to these zones. In our example from Figure 8.2, Section 8.1.2, we would mark nodes from the zones "Intranet" ($VM_{B1}, VM_{B2}, VM_{B3}$) and "Internet" ($VM_{B4}$) as sources and the guardians of the opposite zones (vFW_{A1}, vFW_{A2}) as sinks, to determine

isolation breaches in both directions. After the transitive closure of the traversal rules, we check whether any additional colors "spilled" into a given sink. If a sink gets connected to an additional color, then we have found a potential isolation breach. You could imagine the dedicated color sinks as a honeypot, waiting for colors from other zones to spill over.

Observe that the detection of a color spill only indicates the existence of a breach and between which zones (source-sink pairs) it has occurred. The color flow can be visualized and of some use for administrators to fix the problem. In addition, we study different refinement methods for root-cause analysis, in order to pinpoint critical edges responsible for the information flow in a industry case study (Section 8.7).

8.5 Security of Information Flow Analysis

Definition 10 (System Assumptions) *We assume correctness of discovery/translation and isolation behavior as defined in Section 8.4.4.*

- Completeness of Discovery*: We assume that the configuration output of virtualized infrastructures contains all elements that might solicit information flow (cf. Section 8.4.1)*
- Correct Translation Modules*: We assume that the discovery modules analyzing concrete systems are capable to correctly identify configuration elements that translate to vertices and edges in the realization model (cf. Section 8.4.2).*
- Hypervisor Separation*: We assume that the hypervisor sufficiently prevents cross-VM information flow through covert channel down to a tolerable level (cf. Rule 4, Section 8.4.4).*[3]
- VLAN Separation by Physical Switches*: We assume that physical switches isolate different VLANs from each other (cf. Rule 1, Section 8.4.4).*

8.5.1 Reduction to Correctness of the Traversal Rules

The graph coloring establishes the following events through a transitive closure of traversal rules $f_{\mathbb{T},\mathbb{P}}$ over a graph (V, E) with sources $\hat{V}$ and sinks $\breve{V}$.

Definition 11 (Events) Wlog., *we model admissible colors of an information sink $\breve{v} \in \breve{V}$ as colors associated with $\breve{v}$. Then we have:*

- *We call an event* B *an* isolation breach *if a information sink $\breve{v} \in \breve{V}$ is colored with an inadmissible color of a information source $\hat{v} \in \hat{V}$ such that $\hat{v} \neq \breve{v}$.*
- *We call an event* A *an* alarm *if an isolation audit reports an information flow between a distinct information source $\hat{v} \in \hat{V}$ and information sink $\breve{v} \in \breve{V}$.*
- *We call the set of events* $\neg\mathsf{A} \wedge \mathsf{B}$ *a* false negative.
- *We call the set of events* $\mathsf{A} \wedge \neg\mathsf{B}$ *a* false positive.

Corollary 1 (Structural Information Control) *Under the assumptions from Def. 10 and correct traversal rules $f_{\mathbb{T},\mathbb{P}}$, from the absence of false negatives in an isolation analysis ($\neg\mathsf{A} \wedge \mathsf{B} = \emptyset$) follows that a breach of structural information control is indicated by an alarm event* A.

[3] This assumption is modeled by the hypervisor traversal rules and can be explicitly specified by administrators.

Because we modeled the mediation by dedicated guardians explicitly by traversal rules and inter-zone information flow by B events, this is by construction.

Note that the goal of the analysis system is to detect isolation breaches, that is, breaches of structural information control. We cannot prove an absence of information flow, i.e., verify structural information control, but only detect attack states. We optimize the detection thereof by minimizing the false negative rate through reduction to correct traversal rules (making sure we miss as few breaches as possible) and maximizing the Bayesian detection rate through mitigation of false positives (finding the actual needles in the haystack).

Theorem 1 (Reduction to Traversal Rules) *The correct isolation modeling by traversal rules $f_{\mathbb{T},\mathbb{P}}$ implies absence of false negatives.*

We prove Theorem 1 by contradiction and induction over the length n back-trace graph traversal. The proof by itself is straight-forward as graph coloring (Def. 7) is a recursive definition.

Proof 1 *Let sets of types $\mathbb{T}$ and properties $\mathbb{P}$, a valid graph (V, E) and information sources $\hat{V}$ and sinks $\breve{V}$ be given.*

Suppose *a false negative event $N \in (\neg\mathsf{A} \wedge \mathsf{B}) \neq \emptyset$. By definition, there exists a breach, that is a sink $\breve{v} \in \breve{V}$ for which holds that it is colored by a source $\hat{v} \in \hat{V}, \hat{v} \neq \breve{v}$.*

Initialize a set $\vec{E} = \emptyset$.

Induction start *$n = 1$: the sink $\breve{v}$ is colored because of the breach event* B.

Assume the induction statement true for *$n - 1$: A colored vertice v_{n-1} could only have been colored if*

(a) v_{n-1} is source $\hat{v}$ with the corresponding color (then we are done and output $\vec{E}$) or

(b) there exists an edge $e = (v_n, v_{n-1})$ with $v_n = (\cdot, t_n, p_n)$ and $v_{n-1} = (\cdot, t_{n-1}, p_{n-1})$ for which holds: v_n is colored and the traversal rules $f_{\mathbb{T},\mathbb{P}}(t_n, t_{n-1}, p_n, p_{n-1})$ evaluate to follow. *Accumulate $\vec{E} := \vec{E} \cup \{e\}$.*

If the induction succeeds, then $\vec{E}$ is a construction of a path between source $\hat{v}$ and sink $\breve{v}$, thus an alarm event, A. *We obtain a contradiction against $N \in \neg\mathsf{A}$.*

Consequently, for any case in which no sink-source path can be constructed, there exists an edge e for which the traversal rules $f_{\mathbb{T},\mathbb{P}}$ evaluate to stop. *This reduces the false negative to the correctness for the traversal rules.*

8.5.2 Correctness of the given Traversal Rules

The correctness of the traversal rules from Table 8.1, Section 8.4.4 remains to be shown, where we need to analyze on two levels: (1) correctness of individual rules and (2) correctness of their composition.

Individual Rules

We examine the traversal rules in detail in this section. We first highlight the most important points and then illustrate three specific issues.

- *Network*: We model correct implementation of physical switches (Rule 5), VLAN en- and decapsulation and lift the properties of cryptographic secure channels (e.g., [CK02, MV09]) to VLAN tags (Rules 1, and 8, to 10).

- *Physical Machine, Hypervisor, VM Stack*: We claim secure isolation by management OS and physical machine (Rules 2 and 3) as well as cross-VM isolation (Rule 4). The former rules are elementary for virtualization security, the latter rule is arguable as it models the hypervisor's multi-tenancy capability and needs to be reconsidered depending on the actual environment (cf. [RTSS09b, Aci07] and discussion below).

- *Storage*: We model secure separation by physical disks as well as by the file system (Rules 11 and 12), where the latter rule is systematically enforced by virtualization vendors (e.g., [VMw06]) and can be checked automatically [YTEM06].

First, let us analyze the rules for network switches and VLAN traffic. Rule 5 assume a correct implementation of an isolation by network switches for switched-off ports. Rules 6 and 7 establish the VLAN en- and decapsulation by network switches and are interesting for the security analysis. The rules assign a VLAN-specific color to information flow for in-ports with VLAN tagging and only allow information traversal at out-ports with matching VLAN tags. This models the VLANs' traffic separation by encryption lifted to VLAN tags as well as a cross-session key separation assumption, standard for secure channels: messages encrypted under one VLAN tag cannot interfere with messages encrypt under other VLAN tags and can only be decrypted under the same VLAN tag. We can derive these properties from research on secure channels and their parallel composition (in cryptography for instance Canetti and Krawczyk's work on UC secure channels [CK02]; in formal methods for instance Mödersheim and Viganò's formalization of secure pseudonymous channels [MV09].) Rule 9 stops information flow at ports with non-matching VLAN tags accordingly. Rule 8 has information flow follow through for trunked VLAN ports. Otherwise, we assume that the network and physical switches securely configure and implement VLAN traffic isolation for flows from switch to port (Rules 10 and 1). We conclude that these assumptions are natural and model correct network behavior.

Second, let us consider the stack of physical machine, hypervisor and VMs. Rules 2 and 3 make the assumptions that a management OS and physical host provide secure isolation and that all information flow is accounted for explicitly. These assumptions are necessary for virtualization security, as information leakage from these components can subvert the entire system's security, and model standard trust assumptions. Rule 4 is interesting as it assumes that hypervisors sufficiently separate VMs against each other, that is, that information flow through cross-VM covert channels can be neglected. Research results exist that highlight cross-VM covert channels, for instance [RTSS09b, Aci07]. Therefore, this trust assumption on the hypervisor's multi-tenancy capability must be subject to thorough debate.[4] Whereas the isolation assumptions on physical machine and management OS are natural and well founded, we conclude that the modeling of covert channels is a key trust decision for the hypervisor model.

Third, let us consider the information model for storage. Rule 11 models that the storage controllers are capable of separating information flow to physical disks, whereas Rule 12 establishes that the file system prevents cross file information flow through its access control enforcement. The former property is systematically enforced by virtualization vendors, such as VMware [VMw06] that do not allow reconfiguration of storage back-ends for VMs. The latter property found attention in research and can be checked with tool support [YTEM06]. We conclude that both assumptions are natural to make.

[4]For high-security environments, we recommend to set this rule to follow and therefore only relying on physical separation, yet dismissing hypervisor multi-tenancy.

Correct Traversal Rule Composition

The composition establishes the following robustness principles:

- *Explicit Knowledge Model*: The explicit traversal rules model all and only known facts about information flow and isolation. Thus, traversal rules focus on preventing false negatives introduced by invalid assumptions.
- *Strict Over-abstraction*: When in doubt, the traversal rules must be a conservative estimate towards information flow, that is, model a super-set of potential information flow. By that, traversal rules will never introduce false negatives at the cost of additional false positives.
- *Default-Traversal Behavior*: The default rules establishing completion on the traversal rules must all be *default-follow rules*, that is, evaluate undetermined cases to 1 and log such results. Thus, the completion will only introduce false positives but never false negatives.

We conclude that the traversal rule robustness principles are all lined up to fence off false negatives, yet at the cost of false positives. Whereas this trade-off benefits a conservative security analysis, it impacts its effectiveness, as becomes manifest in its *overall detection rate*.

8.5.3 Overall Detection Rate

The overall detection rate of an analysis system establishes a relation between alarm and breach events, A and B, as defined in Def. 11. We analyze the effectiveness of the analysis system, in particular with respect to rejection of *false positives*, whose influence through the base-rate fallacy rate was established by Axelsson [Axe00] in the area of intrusion detection systems.

Definition 12 (Detection Rates) *The* detection rate *is* $\mathsf{P}[\mathsf{A}|\mathsf{B}]$*, alarm contingent on breach, obtainable by testing the analysis system against scenarios known to constitute a* B *event. The* false alarm rate *is* $\mathsf{P}[\mathsf{A}|\neg\mathsf{B}]$*, the false positive rate, obtainable analogously. The* false negative rate *is* $\mathsf{P}[\neg\mathsf{A}|\mathsf{B}] = 1 - \mathsf{P}[\mathsf{A}|\mathsf{B}]$*. The* Bayesian detection rate *is* $\mathsf{P}[\mathsf{B}|\mathsf{A}]$*, that is the rate with which an alarm event implies an actual breach event. The* all-is-well rate *is* $\mathsf{P}[\neg\mathsf{B}|\neg\mathsf{A}]$*, that is the rate with which the absence of an alarm implies that all is well.*

Our goal is to maximize the Bayesian detection rate and the all-is-well rate, which we determine with Bayes Theorem:

$$\mathsf{P}[\mathsf{B}|\mathsf{A}] = \frac{\mathsf{P}[\mathsf{B}] \cdot \mathsf{P}[\mathsf{A}|\mathsf{B}]}{\mathsf{P}[\mathsf{B}] \cdot \mathsf{P}[\mathsf{A}|\mathsf{B}] + \mathsf{P}[\neg\mathsf{B}] \cdot \mathsf{P}[\mathsf{A}|\neg\mathsf{B}]}$$

If we assume that the rate of breaches is low compared to the rate of non-breaches, $\mathsf{P}[\mathsf{B}] \ll \mathsf{P}[\neg\mathsf{B}]$, we see that the false positive rate $\mathsf{P}[\mathsf{A}|\neg\mathsf{B}]$ dominates the denominator of the Bayesian detection rate. This analysis asks for caution. Even though the focus on absence of false negatives implies a conservative security analysis, the presence of false positives can diminish the effectiveness of the analysis system easily.

Conjecture 1 *The correctness of the traversal rules determines the absence of false negative events. The coverage of explicit correct traversal rules determines the absence of false positive events.*

Although the absence of false negatives is important for the system's security, effectiveness requires the absence of false positives, as well. This is to ensure that administrators are able to find the actual breaches in the set of all alarm events. Therefore, administrators need to fine-tune the traversal rules to maximize coverage and, thus, the Bayesian detection rate.

8.5.4 Discussion

The transitive closure over the graph coloring securely lifts the isolation analysis to an analysis of the traversal rules $f_{\mathbb{T},\mathbb{P}}$. Therefore, the correctness of the traversal rules becomes a make-or-break criterion for the analysis method and impacts the detection rate.

We observe a *complexity reduction*: the simple traversal rules have a complexity of their relation $R \subseteq \mathbb{T} \times \mathbb{T}$. In practice, $|\mathbb{T}| \ll |V|$ as well as $|\mathbb{T}|^2 \ll |E| \leq |V|^2$, with the number of properties set for $f_{\mathbb{T},\mathbb{P}}$ being small. Therefore, the complexity of analyzing the traversal rules $f_{\mathbb{T},\mathbb{P}}$ is much smaller than the complexity of isolation analysis. This allows administrators to explicitly model and thoroughly and efficiently check their knowledge and trust assumptions about information flow and isolation.

Because our traversal rules base on the principle of *over-abstraction*, that is, resort to default-traversal in undetermined cases, the method excludes false negatives, at the cost of additional false positives. The method is therefore always on the conservative side, even though we are well aware that the false positive rate impacts the overall detection rate [Axe00]. We provide the general analysis framework and offer user-defined traversal rules to fine-tune the analysis method to reduce false positives and maximize the Bayesian detection rate. Also, we experiment with refinement methods for a subsequent root-cause analysis to pinpoint critical information flow edges.

In principle, our tool is in a similar situation as the first intrusion detection systems. There do not exist standardized data sets to quantify and calibrate false positive and false negative rates. We approach this situation by obtaining real-world data from third parties and are currently testing the analysis method in sizable real-world customer deployments, such as the case study discussed below, to establish the detection rates.

8.6 Implementation

We have implemented a prototype of our automated information flow analysis in Java that consists of roughly twenty thousand lines of code. Furthermore, we have additional scripts written in Python that perform post-processing for visualization purposes and refinement for root-cause analysis. The prototype consists of two main programs, that is, the discovery, and a processing and analysis program. The result of the discovery is written into an XML file and is used as the input for the analysis.

8.6.1 Discovery

The functionality of the discovery and its different probes were already outlined in Section 8.4.1. There exist different ways to implement a discovery probe. A probe can establish a secure console (SSH) connection to the virtualized host or the management console where commands are executed and the output is processed. Typically, the output is either XML, which is stored in the discovery XML file directly, or the output has to be parsed and transformed into XML.

As alternative to the secure console, a probe can connect to a hypervisor-specific API, such as a web service, that provides information about the infrastructure configuration.

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

8.6.2 Processing

The *processing* program consists of the transformation of the discovery XML into the realization model, the graph coloring, and the analysis of the colored realization graph.

The realization model is a class model that is used for generating Java class files. During the transformation of the XML into the realization model, instances of these classes are created, their attributes set, and linked to instances of other classes according to the mapping rules (cf. Section 8.4.2). Again, we illustrate this process for VMware. Each mapping rule embodies knowledge of VMware's ontology of virtualized resources to configuration names, for instance, that VMware calls storage configuration entries storageDevice. The edges are encoded implicitly by XML hierarchy (for instance, that a VM is part of a physical host) as well as explicitly by Managed Object References (MOR). The mapping rules establish edges in the realization model for all hierarchy links and for all MOR links between configuration entries for realization model types.

The traversal rules used for the graph coloring (cf. Section 8.4.3 and Section 8.4.4) are specified in XML. Intermediate results, such as the paths of the graph coloring, can be captured and used for further processing, i.e., visualization. We implemented Python scripts that generate input graphs for the *Gephi visualization framework* [5], such as illustrated in Figure 8.1.

8.6.3 OpenStack Integration

The integration of our automated information flow analysis approach with the TClouds' designated cloud computing platform *OpenStack* requires the following two extensions.

- OpenStack has to provide an interface for accessing the configuration of the virtualized systems in the cloud.
- The analysis framework needs to be extended to translate OpenStack specific information into the common realization model.

Discovery. We introduce a new API command *discover* in the OpenStack management component. This command is used by SAVE to obtain the configuration information of all virtualized systems in the cloud. The extended management component contacts all compute nodes and uses the dump functionality of *libvirt* on each node to gather the configuration data. This includes information about the running virtual machines, attach block devices or iSCSI storage volumes.

[5] http://gephi.org/

Processing. In order to apply the information flow analysis, the OpenStack specific configuration data has to be translated into the common realization model. The translation module for OpenStack iterates over all the compute node hosts and uses the existing translator for *libvirt*. For iSCSI storage volumes, we obtain the storage provider from the volume's name, in order to establish the relationship between storage and compute nodes.

8.7 Case Study

We launched a case study with a global financial institution for a performance evaluation and for further validation of detection rates and behavior in large-scale heterogeneous environments. The analyzed virtualized infrastructure is based on VMware and consists of roughly 1,300 VMs, the corresponding realization model graph of 25,000 nodes and 30,000 edges. The production system has strong requirements on isolation between clusters of different security levels, such as high-security clusters, normal operational clusters, backup clusters and test clusters. In addition, we can work with a comprehensive inventory of virtualized resources that serves as specification of an ideal state (machine placement, zone designation and VLAN configuration) and as basis for alarm validation.

We examine preliminary lessons learned, where we first consider the operation of the tool itself. The phases discovery, transformation to realization model and graph coloring executed successfully. The visualization of all results presented a challenge as a 25,000-node/30,000-edge graph overburdened the built-in visualization of the tool.

From a performance perspective, the discovery of the infrastructure using the VMware probe in combination with vCenter requires about *seven minutes*, and results in a discovery XML file with a size of *61MB*. The discovery was performed in a production environment, where network congestion and other tools using the same vCenter can have a negative effect on the discovery performance. The overall analysis of the infrastructure using the discovery XML file requires *53 seconds*, where *46 seconds* are spent on the graph coloring. This demonstrates a reasonable performance for the discovery and analysis of a mid-sized infrastructure, such as the one in our case study.

From a security perspective, the tool indeed found several realistic isolation breaches, which we highlighted by adding virtual edges between breached clusters. All isolation breaches constituted potential information flows. By that we could show actual breaches between high-security, normal operational and test clusters. We have furthermore shown that the documentation of the permitted flows was incomplete: One breach that the system identified violated the initial policy given by the customer and was fixed by augmenting the policy.

Root-cause analysis answers the question which edges and nodes are ultimately responsible for the breach. We found that color spill after a traversal to a new cluster may hamper the subsequent root-cause analysis. We therefore introduced multiple automated refinement mechanisms after the graph-coloring phase to support the elimination of classes of potential root causes. *First*, we benefited greatly from a process of elimination, that is, to exclude, for instance, that information has flown over storage edges. *Second*, it was helpful to allow partial coloring, in particular to stop color propagation after detecting a breach to another cluster. *Third*, we introduced a reverse flow tree that captured which path information flow took as prelude to a breach. Figure 8.4 depicts an example of such a color tree: the tree is a subgraph highlighting a cross-cluster information flow path. *Fourth*, we further refined this tree by extracting critical edges, such as passed VLANs, to pinpoint routes of information flow.

In conclusion, we added a refinement phase driven by reusable Python scripts. We obtained

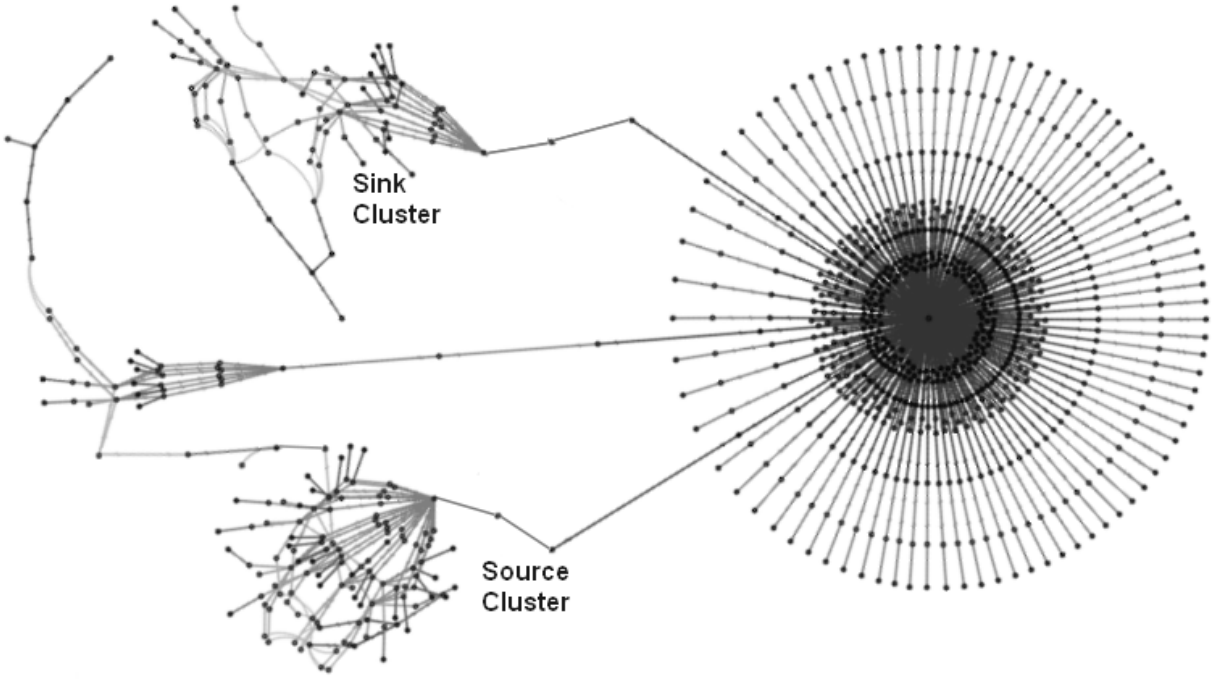


Figure 8.4: Root-cause analysis of a source cluster with information flow to a sink cluster. The tree refinement derives only the sub-graphs relevant for an isolation breach. The "flower" is a large-scale switch.

multiple realistic alarms and could trace their root causes. The graph export to Gephi enabled the efficient visualization of root causes and information flows for human validation.

8.8 Conclusion

We demonstrated an analysis system that discovers the configuration of complex heterogeneous virtualized infrastructures and performs a static information flow analysis. Our approach is based on a unified graph model that represents the configuration of the virtualized infrastructure and a graph coloring algorithm that uses a set of traversal rules to specify trust assumptions and information flow properties in virtualized systems. Based on the colored graph model, the system is able to diagnose isolation breaches, which would violate the customer isolation requirements in multi-tenant datacenters. We showed in our security analysis that we can reduce the correctness and detection rate to the correctness and coverage of the graph traversal rules. Based on existing research and systems knowledge, we submit that the present traversal rules are natural and correct.

Bibliography

[Abb11a] Imad M. Abbadi. Clouds' Infrastructure Taxonomy, Properties, and Management Services. In *CloudComp '11: Proceeding of the International workshop on Cloud Computing: Architecture, Algorithms and Applications (to appear)*, LNCS. Springer-Verlag, July 2011.

[Abb11b] Imad M. Abbadi. Middleware Services at Cloud Virtual Layer. In *DSOC 2011: Proceedings of the 2nd International Workshop on Dependable Service-Oriented and Cloud computing (to appear)*. IEEE Computer Society, August 2011.

[Abb11c] Imad M. Abbadi. Toward Trustworthy Clouds' Internet Scale Critical Infrastructure. In *ISPEC '11: in proceedings of the 7th Information Security Practice and Experience Conference*, volume 6672 of *LNCS*, pages 73–84. Springer-Verlag, Berlin, June 2011.

[Aci07] Onur Aciiçmez. Yet another microarchitectural attack: exploiting i-cache. In *CSAW '07: Proceedings of the 2007 ACM Workshop on Computer Security Architecture*, pages 11–18. ACM, 2007.

[Adv05] Advanced Micro Devices, Inc. *AMD64 Virtualization Codenamed "Pacifica" Technology – Secure Virtual Machine Architecture Reference Manual*, 2005.

[ALRL04] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[ama] Amazon Article: Client-Side Data Encryption with the AWS SDK for Java and Amazon S3. http://aws.amazon.com/articles/2850096021478074.

[ASMEAE08] Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. Global Verification and Analysis of Network Access Control Configuration. Technical report, DePaul University, 2008.

[Axe00] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Trans. Inf. Syst. Secur.*, 3(3):186–205, 2000.

[BCH+10] M. Björkqvist, C. Cachin, R. Haas, X.Y. Hu, A. Kurmus, R. Pawlitzek, and M. Vukolić. Design and implementation of a key-lifecycle management system. *Financial Cryptography and Data Security*, pages 160–174, 2010.

[BLP+07] Cataldo Basile, Antonio Lioy, Gregorio Martinez Perez, Felix J. Garcia Clemente, and Antonio F. Gomez Skarmeta. POSITIF: A Policy-Based Security Management System. In *POLICY '07 Proceedings of the Eighth IEEE International Workshop on Policies for Distributed Systems and Networks*, page 280, Bologna, Italy, 2007.

[BLVG+08] Carlos Blanco, Joaquin Lasheras, Rafael Valencia-García, Eduardo Fernández-Medina, Ambrosio Toval, and Mario Piattini. A Systematic Review and Comparison of Security Ontologies. In *ARES '08 Proceedings of the 2008 Third International Conference on Availability, Reliability and Security*, pages 813–820, Barcelona, Spain, 2008.

[BPA11] N. Bonvin, T.G. Papaioannou, and K. Aberer. Autonomic SLA-driven provisioning for cloud applications. In *11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 434–443, May 2011.

[BSB+10] Matthias Bolte, Michael Sievers, Georg Birkenheuer, Oliver Niehörster, and André Brinkmann. Non-intrusive virtualization management using libvirt. In *DATE '10 Proceedings of the Conference on Design, Automation and Test in Europe*, pages 574–579, Dresden, Germany, 2010.

[BSI05] BSI. IT-Grundschutz Catalogues. Technical report, Bundesamt für Sicherheit in der Informationstechnik, 2005.

[BSP+10] Sören Bleikertz, Matthias Schunter, Christian W. Probst, Dimitrios Pendarakis, and Konrad Eriksson. Security audits of multi-tier virtual infrastructures in public infrastructure clouds. In *Proceedings of the 2010 ACM Workshop on Cloud Computing Security*, CCSW '10, pages 93–102, New York, NY, USA, 2010. ACM.

[CDE+10] L. Catuogno, A. Dmitrienko, K. Eriksson, D. Kuhlmann, G. Ramunno, A.R. Sadeghi, S. Schulz, M. Schunter, M. Winandy, and J. Zhan. Trusted Virtual Domains–Design, Implementation and Lessons Learned. *Trusted Systems*, pages 156–179, 2010.

[CDMV09] María Emilia Cambronero, Gregorio Díaz, Enrique Martínez, and Valentín Valero. A Comparative Study between WSCI, WS-CDL, and OWL-S. In *ICEBE '09: IEEE International Conference on e-Business Engineering*, pages 377–382, Macau, China, 2009.

[CGJ+09] R. Chow, P. Golle, M. Jakobsson, E. Shi, J. Staddon, R. Masuoka, and J. Molina. Controlling data in the cloud: outsourcing computation without outsourcing control. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 85–90. ACM, 2009.

[CK02] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels (extended abstract). In *Advances in Cryptology: EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351. Springer, 2002. Extended version in IACR Cryptology ePrint Archive 2002/059, http://eprint.iacr.org/.

[CLM+10] Luigi Catuogno, Hans Löhr, Mark Manulis, Ahmad-Reza Sadeghi, Christian Stüble, and Marcel Winandy. Trusted Virtual Domains: Color Your Network. *Datenschutz und Datensicherheit (DuD) 5/2010*, pages 289–294, 2010.

[cra] CrashPlan: Archive Encryption Key Security. http://support.crashplan.com/doku.php/articles/encryption_key.

[CSS+09] M. Christodorescu, R. Sailer, D.L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102, 2009.

[DKF+03] Grit Denker, Lalana Kagal, Tim Finin, Massimo Paolucci, and Katia Sycara. Security for DAML Web Services: Annotation and Matchmaking. In *ISWC '03 Proceedings of the Second International Semantic Web Conference*, pages 335–350, Sanibel Island, USA, 2003.

[DMT10] DMTF. Open Virtualization Format Specification. Technical report, DMTF, 2010.

[dro] Dropbox: Secure Storage. http://www.dropbox.com/security.

[DTB10] Amir Vahid Dastjerdi, Sayed Gholam Hassan Tabatabaei, and Rajkumar Buyya. An Effective Architecture for Automated Appliance Management System Applying Ontology-Based Cloud Discovery. In *CCGRID '10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 104–112, Melbourne, Australia, 2010.

[EdPG+08] J. Ejarque, M. de Palol, I. Goiri, F. Julia, J. Guitart, R.M. Badia, and J. Torres. SLA-driven semantically-enhanced dynamic resource allocator for virtualized service providers. In *IEEE 4th International Conference on eScience*, pages 8–15, Dec. 2008.

[FE09] Stefan Fenz and Andreas Ekelhart. Formalizing information security knowledge. In *ASIACCS '09 Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, pages 183–194, Sydney, Australia, 2009.

[FN94] Amos Fiat and Moni Naor. Broadcast encryption. In *Proceedings of the 13th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '93, pages 480–491, London, UK, 1994. Springer-Verlag.

[Gen09] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the 41st annual ACM symposium on Theory of Computing*, pages 169–178. ACM, 2009.

[GM82] Joseph A. Goguen and José Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982.

[GR05] Tal Garfinkel and Mendel Rosenblum. When Virtual is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 20–20, Berkeley, CA, USA, 2005. USENIX Association.

[Gra91] James W. Gray, III. Toward a mathematical foundation for information flow security. In *IEEE Symposium on Security and Privacy*, pages 21–35. IEEE, 1991.

[Gra09] D. Grawrock. *Dynamics of a Trusted Platform: A building block approach*. Intel Press, 2009.

[HPS09] Matthew Horridge and Peter F. Patel-Schneider. OWL 2 Web Ontology Language Manchester Syntax. Technical report, World Wide Web Consortium, 2009.

[HPSB+04] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: A Semantic Web Rule Language Combining OWL and RuleML. Technical report, World Wide Web Consortium, 2004.

[HPSP10] D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services, the case of deduplication in cloud storage. *IEEE Security and Privacy Magazine, special issue of Cloud Security*, 2010.

[HS08] N. Heninger and H. Shacham. Improved RSA Private Key Reconstruction for Cold Boot Attacks. *Cryptology ePrint Archive, Report 2008/510*, 2008.

[HSD07] Almut Herzog, Nahid Shahmehri, and Claudiu Duma. An Ontology of Information Security. In *Techniques and Applications for Advanced Information Privacy and Security: Emerging Organizational, Ethical, and Human Issues*, chapter XVIII, pages 278–301. IGI Global, 2007.

[HSH+09] J.A. Halderman, S.D. Schoen, N. Heninger, W. Clarkson, W. Paul, J.A. Calandrino, A.J. Feldman, J. Appelbaum, and E.W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.

[HY86] J. Thomas Haigh and William D. Young. Extending the non-interference version of MLS for SAT. In *IEEE Symposium on Security and Privacy*, pages 60–60. IEEE, 1986.

[Int07] Intel Corporation. *Intel® Trusted Execution Technology – Preliminary Architecture Specification*, 2007.

[ISO05] ISO/IEC. ISO/IESC 27001:2005 Information technology – Security techniques – Information security management systems – Requirements. Technical report, International Organization for Standardization, 2005.

[Jac90] J. Jacob. Separability and the detection of hidden channels. *Inf. Process. Lett.*, 34:27–29, February 1990.

[Jal94] Pankaj Jalote. *Fault tolerance in distributed systems*. Prentice Hall, 1994.

[Kau07] B. Kauer. OSLO: Improving the security of Trusted Computing. In *Proceedings of 16th USENIX security symposium on usenix security symposium*, pages 1–9. USENIX Association, 2007.

[KBR+05] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web Services Choreography Description Language Version 1.0. Technical report, World Wide Web Consortium, 2005.

[KF91] N. L. Kelem and R. J. Feiertag. A Separation Model for Virtual Machine Monitors. In *IEEE Symposium on Security and Privacy*, pages 78–86. IEEE, 1991.

[KL09] Amir R. Khakpour and Alex Liu. Quarnet: A Tool for Quantifying Static Network Reachability. Technical Report MSU-CSE-09-2, Department of Computer Science, Michigan State University, East Lansing, Michigan, January 2009.

[KLK05] Anya Kim, Jim Luo, and Myong H. Kang. Security Ontology for Annotating Resources. In *OTM '05 Proceedings of the Confederated International Conferences 2005*, pages 1483–1499, Agia Napa, Cyprus, 2005.

[KSS+09] Sunil D. Krothapalli, Xin Sun, Yu-Wei E. Sung, Suan Aik Yeo, and Sanjay G. Rao. A toolkit for automating and visualizing VLAN configuration. In *SafeConfig '09: Proceedings of the 2nd ACM Workshop on Assurable and Usable Security Configuration*, pages 63–70. ACM, 2009.

[Lam73] Butler W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

[LKKF03] H. Ludwig, A. Keller, R. P. King, and R. Franck. Web service level agreement (WSLA) language specification. 2003.

[LML05] Paul Lin, Alexander MacArthur, and John Leaney. Defining autonomic computing: A software engineering perspective. In *Proceedings of the 2005 Australian conference on Software Engineering*, pages 88–97, Washington, DC, USA, 2005. IEEE Computer Society.

[Man01] Heiko Mantel. Information flow control and applications - bridging a gap. In José Nuno Oliveira and Pamela Zave, editors, *FME*, volume 2021 of *Lecture Notes in Computer Science*, pages 153–172. Springer, 2001.

[MK05] Robert Marmorstein and Phil Kearns. A Tool for Automated iptables Firewall Analysis. In *ATEC '05: Proceedings of the USENIX Annual Technical Conference*, pages 44–44, Berkeley, CA, USA, 2005. USENIX Association.

[MLQ+10] J.M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *2010 IEEE Symposium on Security and Privacy*, pages 143–158. IEEE, 2010.

[moz] mozy documentation: How do I know that my data is safe and secure? http://docs.mozy.com/docs/en/user-home-win/faq/concepts/commissure_data_security_c.html.

[MPP+08] J.M. McCune, B.J. Parno, A. Perrig, M.K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 315–328. ACM, 2008.

[MV09] Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In Michael Backes and Peng Ning, editors, *ESORICS*, volume 5789 of *Lecture Notes in Computer Science*, pages 337–354. Springer, 2009.

[MWZ00] Alain Mayer, Avishai Wool, and Elisha Ziskind. Fang: A Firewall Analysis Engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE.

[oas] OASIS Key Management Interoperability Protocol (KMIP). http://www.oasis-open.org/committees/kmip/.

[Ope10] OpenSource. OpenStack, 2010. http://www.openstack.org/.

[Per05] Colin Percival. Cache missing for fun and profit. http://www.daemonology.net/papers/htt.pdf, May 2005.

[PS08] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. Technical report, World Wide Web Consortium, 2008.

[PSHH04] Peter F. Patel-Schneider, Patrick Hayes, and Ian Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Technical report, World Wide Web Consortium, 2004.

[QD02] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of the Conference on USENIX File and Storage Technologies*, pages 89–101, 2002.

[Red11] Red Hat. libvirt: The virtualization API, 2011.

[RTF06] Steve Ross-Talbot and Tony Fletcher. Web services choreography description language: Primer version 1.0. Technical report, World Wide Web Consortium, 2006.

[RTSS09a] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 199–212, New York, NY, USA, 2009. ACM.

[RTSS09b] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and Communications Security*, pages 199–212, New York, NY, USA, 2009. ACM.

[Rus81] John Rushby. Design and verification of secure systems. In *Proceedings of the eighth ACM Symposium on Operating Systems Principles*, SOSP '81, pages 12–21, New York, NY, USA, 1981. ACM.

[Rus82] John Rushby. Proof of separability a verification technique for a class of security kernels. In Mariangiola Dezani-Ciancaglini and Ugo Montanari, editors, *International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 1982.

[Rus92] John Rushby. Noninterference, transitivity, and channel-control security policies. Technical report, SRI International, Dec 1992.

[SCCS11] Jacopo Silvestro, Daniele Canavese, Emanuele Cesena, and Paolo Smiraglia. A unified ontology for the virtualization domain. In *DOA-SVI'11: Proceedings of the 1st International Symposium on Secure Virtual Infrastructures*. Springer, October 2011. To appear.

[SK10] U. Steinberg and B. Kauer. NOVA: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European conference on Computer systems*, pages 209–222. ACM, 2010.

[SM03] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[Ste08] Aaron Steele. Ontological Vulnerability Assessment. In *WISE '08 Proceedings of the 2008 international workshops on Web Information Systems Engineering*, pages 24–35, Auckland, New Zealand, 2008.

[SVDO+06] L.F.G. Sarmenta, M. Van Dijk, C.W. O'Donnell, J. Rhodes, and S. Devadas. Virtual monotonic counters and count-limited objects using a tpm without a trusted os. In *Proceedings of the first ACM workshop on Scalable trusted computing*, pages 27–42. ACM, 2006.

[TBDP+06] Christophe Taton, Sara Bouchenak, Noel De Palma, Daniel Hagimont, and Sylvain Sicard. Self-sizing of clustered databases. In *Proceedings of the 2006 International Symposium on on World of Wireless, Mobile and Multimedia Networks*, WOWMOM '06, pages 506–512, Washington, DC, USA, 2006. IEEE Computer Society.

[tbo] Trusted Boot (tboot). http://sourceforge.net/projects/tboot/.

[tcg] Trusted Computing Group. http://www.trustedcomputinggroup.org.

[Tru11] Trusted Computing Group (TCG). *TPM Main Specification, Version 1.2, Revision 116*, March 2011.

[VDJ10] M. Van Dijk and A. Juels. On the Impossibility of Cryptography Alone for Privacy-Preserving Cloud Computing. *IACR ePrint*, 305, 2010.

[VH06] Artem Vorobiev and Jun Han. Security Attack Ontology for Web Services. In *SKG '06 Proceedings of the Second International Conference on Semantics, Knowledge, and Grid*, pages 42–47, Guilin, China, 2006.

[VMw06] VMware. Providing LUN Security. Available at http://www.vmware.com/pdf/esx_lun_security.pdf, March 2006.

[VMw10] VMware. VMware vCenter Server, 2010. http://www.vmware.com/products/vcenter-server/.

[Woj08] Rafal Wojtczuk. Adventures with a certain Xen vulnerability (in the PVFB backend). http://invisiblethingslab.com/pub/xenfb-adventures-10.pdf, October 2008.

[Woo01] Avishai Wool. Architecting the Lumeta Firewall Analyzer. In *SSYM'01: Proceedings of the 10th conference on USENIX Security Symposium*, pages 7–7, Berkeley, CA, USA, 2001. USENIX Association.

[XZM+05] G.G. Xie, Jibin Zhan, D.A. Maltz, Hui Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of IP networks. In *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 2170 – 2183 vol. 3. IEEE, 13-17 2005.

[Yao86] A.C.C. Yao. How to generate and exchange secrets. In — *27th Annual Symposium on Foundations of Computer Science*, pages 162–167. IEEE, 1986.

[YBS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Towards a Unified Ontology of Cloud Computing. In *GCE '08 Grid Computing Environments Workshop*, pages 1–10, Austin, USA, 2008.

[YTEM06] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24:393–423, November 2006.