NTNU
Norwegian University of
Science and Technology

Automated Security Analysis of Infrastructure Clouds

Sören Bleikertz

Master in Security and Mobile Computing
Submission date: June 2010
Supervisor: Stig Frode Mjølsnes, ITEM

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The importance of virtualized infrastructures and cloud computing is currently increasing rapidly. Virtual infrastructures allow servers, networks, and storage to be virtualized and shared between different users. Cloud computing generalizes and automates this approach such that users of a data center can request virtually any number of machines, networks, and storage while provisioning and scaling is fast and managed transparently by the provider.

The increasing complexity and multitenancy of such virtualized infrastructures can cause severe security problems due to possible misconfigurations, e.g. two different users have access to the same storage, and the abstraction of cloud computing hinders the verification of policy compliance. An automated mechanism is required to handle these scenarios and IBM built a prototype for retrieving the configuration of virtual systems and performing certain security analysis on them.

The goal of this master thesis is to a) extend the prototype to public infrastructure clouds, e.g. Amazon WebServices, and to b) improve the overall security analysis of the discovered hybrid cloud configuration.

a) We are interested to investigate what possible security implications and problems public infrastructure clouds could have and how we can detect them with our analysis framework. The prototype will be extended to handle discovery and isolation analysis of public infrastructure clouds.

b) The analysis improvements could be one or more of the following ones: Identifying the root cause of a security problem in a fine-grained way; Comparison of configurations, e.g. desired configuration and discovered one; Severity rating of security problems and security levels for configurations. We want to investigate if we can successfully detect misconfigurations and do policy compliance checks for private, public and hybrid cloud configurations, and if we can support users and providers of cloud computing in their security management.

Assignment given: 04. January 2010
Supervisor: Stig Frode Mjølsnes, ITEM

MASTER'S THESIS

Automated Security Analysis of Infrastructure Clouds

Sören Bleikertz

Department of Informatics and Mathematical Modelling
Technical University of Denmark

Department of Telematics
Norwegian University of Science and Technology

Supervisors:

Prof. Christian W. PROBST
Technical University of Denmark

Prof. Stig F. MJØLSNES
Norwegian University of Science and Technology

Dr. Matthias SCHUNTER
IBM Research - Zurich

June 2010

Abstract

Cloud computing has gained remarkable popularity in the recent years by a wide spectrum of consumers, ranging from small start-ups to governments. However, its benefits in terms of flexibility, scalability, and low upfront investments, are shadowed by security challenges which inhibit its adoption. In particular, these highly flexible but complex cloud computing environments are prone to misconfigurations leading to security incidents, e.g., erroneous exposure of services due to faulty network security configurations. In this thesis we present a novel approach in the security assessment of multi-tier architectures deployed on infrastructure clouds such as Amazon EC2. In order to perform this assessment for the currently deployed configuration, we automated the process of extracting the configuration using the Amazon API and translating it into a generic data model for later analysis. In the assessment we focused on the reachability and vulnerability of services in the virtual infrastructure, and presented a way for the visualization and automated analysis based on reachability and attack graphs. We proposed a query and policy language for the analysis which can be used to obtain insights into the configuration and to specify desired and undesired configurations. We have implemented the security assessment in a prototype and evaluated it for practical and theoretical scenarios. Furthermore, a framework is presented which allows the evaluation of configuration changes in the agile and dynamic cloud environments with regard to properties like vulnerabilities or expected availability. In case of a vulnerability perspective, this evaluation can be used to monitor the security levels of the configuration over its lifetime and to indicate degradations.

Acknowledgments

I would like to thank my supervisors Matthias Schunter, Christian Probst, and Stig Mjølsnes for allowing me the freedom to pursue my own ideas and directions, although also giving me valuable feedback and comments. Furthermore, I am grateful to the whole Security group of IBM Research Zurich, managed by Andreas Wespi, for supporting this thesis and hosting me for the third time in a row. I am also very thankful to my colleagues who reviewed and provided feedback on this thesis: Animesh Trivedi, Rüdiger Kapitza, Ulrich Dangel, and Gregory Zaverucha. Finally, I would like to thank my family for supporting me throughout my studies.

Sören Bleikertz

Zürich, Switzerland
June 2010

Contents

1 Introduction **1**
- 1.1 Motivation & Problem Statement 2
- 1.2 Scope 2
- 1.3 Methodology 3
- 1.4 Our Contributions 3
- 1.5 Thesis Outline 4

2 Background **5**
- 2.1 Cloud Computing 5
- 2.2 Virtualization 9
- 2.3 Amazon Web Services Architecture 11

3 Literature Review **15**
- 3.1 Virtual Machine Security 15
- 3.2 Network Security Analysis 23

4 Existing Prototype: *SAVE* **29**
- 4.1 Configuration Discovery 29
- 4.2 Analysis 31
- 4.3 Examples 31

5 Configuration Discovery for Public Infrastructure Clouds **34**
- 5.1 Discovery & Data Model for Amazon 34
- 5.2 Integrated Model & Discovery for Infrastructure Clouds 36

6 Multi-tier Virtual Infrastructures in Public Clouds **37**
- 6.1 Configuration Security Audits 38
- 6.2 Comparison of Deployment Methods 50
- 6.3 Analysis of Configuration Changes 53

7 Evaluation & Practical Results **61**
7.1 Performance Evaluation 61
7.2 Reachability Analysis Results 66
7.3 Attack Graph Analysis Results 69

8 Outlook & Open Questions **72**
8.1 Hybrid Cloud Analysis 72
8.2 Amazon Security Group Transformation 73
8.3 Design Tool 73
8.4 Complex Firewall Rules Analysis 73
8.5 SAVEly for Private Clouds 74

9 Conclusion **75**

Bibliography **77**

A Amazon EC2 Architecture Details **87**
A.1 Storage 87
A.2 Networking 88
A.3 Domain Naming 88
A.4 XenStore Dump 88

B Data Models **92**
B.1 Realization Model 92
B.2 Logical Model 94
B.3 Integrated Realization Model 96

C SAVE Examples **98**
C.1 Colored Realization Model 98

D SAVEly Implementation **104**

List of Figures

2.1 The Cloud Cube Model (©Jericho Forum, [Jer09]) 9

4.1 Simple Realization Model Example 32
4.2 Logical Model Example 33

5.1 Amazon Data Model 35

6.1 Visualization of Security Groups Reachability 41
6.2 Relationship between Security Groups and AMIs 45
6.3 Attack Graph of the Multi-Tier Application 46
6.4 Configuration Change between Attack Graphs 56
6.5 Generic Configuration Change 56
6.6 Security Level Monitoring 59

7.1 Security Groups Complete Graph 62
7.2 Complete Attack Graph 64
7.3 SAVEly Reachability Graph 67
7.4 SAVEly Security Group AMI Relationship Graph 67
7.5 SAVEly Attack Graph 69

B.1 *SAVE* Realization Model 93
B.2 *SAVE* Logical Model 95
B.3 *SAVE* Realization Model with Amazon 97

C.1 Complex Analyzed Realization Model Example 103

List of Algorithms

1 Process a Reachability Query 42
2 Verify Exclusiveness of an *only* Policy 43

List of Listings

6.1 `ec2-describe-group` Command Output 40
7.1 Reachability Query File 68
7.2 Reachability Query Output 68
7.3 Reachability Policy File 69
7.4 Attack Query File 70
7.5 Attack Query Output 70
7.6 Attack Policy File 71
7.7 Attack Policy Output 71
A.1 Amazon EC2 XenStore Dump 88
C.1 SAVE Coloring Policy 98
D.1 SAVEly Python Implementation 104

Chapter 1

Introduction

> *"Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."* [MG09b]

In recent years, *Cloud Computing* has gained remarkable popularity due to the economical and technical benefits provided by this new way of delivering computing resources, and the pervasive availability of high-speed networks. Businesses can offload their IT infrastructure into the cloud and benefit from the rapid provisioning and scalability. This allows an on-demand growth of IT resources in addition to a pay-as-you-go pricing scheme, which does not require a high up-front capital investment. These benefits are in particular attractive to small businesses, like start-ups, who often have traffic spikes or a steep growth rate, and who prefer to avoid intensive up-front capital investment in their IT infrastructure. However, cloud computing is not limited to such small business. The US government, one of the largest consumer of information technology, is initiating a move of parts of its IT infrastructure into the cloud, in order to reduce costs and gain productivity [Kun10].

These general principles of cloud computing can be implemented on different abstraction levels. While Infrastructure as a Service, such as Amazon EC2 [Ama10a], provides virtual machines, storage, and networks, higher abstractions include Platform as a Service as well as Software as a Service that provide the actual web-based applications to end-users.

1.1 Motivation & Problem Statement

Despite its benefits, Cloud Computing also induces unique challenges in terms of security [MG09a]. Multi-tenancy requires proper isolation of users, the abstraction of the cloud hinders compliance verification of the underlying architecture, and the sheer complexity of such a system implies a high probability of misconfigurations endangering the overall security.

While the benefits of cloud computing are clear and end-users demand such services, security is a major inhibitor of cloud computing adoption on all levels of abstraction [MG09a]. In numerous studies the security related problems have been pointed out, and in particular noteworthy are [ENI09, Clo09]. One of the top risks exposed in the study is the failure of isolation in the cloud computing environment.

Cloud computing environments are becoming increasingly complex, more tenants are sharing the same physical resources, and the flexibility and possibility of programmatic configurations can lead to unforeseen misconfigurations. For example, network-based storage volumes can be flexibly attached to virtual machines, and potentially a volume will be attached to a wrong virtual machine risking the exposure of sensitive data on that volume. Network security is also flexibly managed through a programmatic interface, which could lead to problems resulting in network services exposed wrongly to the public and opening not properly secured services to other peers. Administrators of such virtual infrastructures must be able to easily understand the complex deployments and ensure that proper security is given. The dynamic and agility of such environments also provides a challenge in ensuring the security over its entire lifetime due to their constant changes.

1.2 Scope

Although cloud computing in general faces security challenges inhibiting the adoption by consumers, in this thesis we will focus on infrastructure clouds. In particular, we chose Amazon has an example for an infrastructure cloud provider, because they are among the widest adopted providers and provide a flexible but also complex architecture, which has a higher potential of misconfigurations leading to security problems. Within the infrastructure cloud, we are in particular interested in the security audit of multi-tier architectures deployed in the cloud with a focus on network security regarding reachability and vulnerability.

1.3 Methodology

In order to successfully address the problem of configuration complexity and potential misconfigurations in cloud computing environments, we narrowed down the problem domain to a specific case of multi-tier applications deployed in infrastructure clouds using a specific cloud provider as an example case. We will study existing literature in the broad domain of virtual machine security, which plays a fundamental part in the security of infrastructure clouds, and network security analysis with a focus on vulnerability assessment and reachability. Based on the insights and inspirations obtained by performing the literature review, we will propose a novel approach in assessing the security of a multi-tier application deployed on the Amazon infrastructure cloud. By implementing our approach and then evaluating it regarding practicality and scalability, we will determine the practical usefulness for detecting misconfigurations even in large-scale deployments. The evaluation is performed both theoretical and practical. The theoretical evaluation is conducted by assuming complex configuration scenarios and analyze the algorithm run-time using an ideal computer. The practical evaluation is performed using the implementation on a sample multi-tier application deployed on Amazon EC2.

1.4 Our Contributions

The main contribution of this thesis is a novel approach in the security evaluation of multi-tier virtual infrastructures, inspired by vulnerability assessment approaches for traditional computing environments and applied for the case of the Amazon infrastructure cloud. The security evaluation consists of an automated security audit process of the currently deployed configuration with regard to a given policy specifying the desired state of the configuration, and an abstract framework for evaluating the security impact of configuration changes.

Besides the main contribution stated above, multiple minor contributions can be pointed out. A comprehensive description of the underlying architecture of the Amazon infrastructure cloud is presented, which was publicly only available in incomplete and fragmented form. We provide a comparison of two methods for deploying multi-tier virtual infrastructures on Amazon with regard to the provided isolation levels. Finally, a data model for representing the configuration of Amazon deployments is presented and integrated into a larger data model capable of representing configurations of different virtualization systems.

1.5 Thesis Outline

The remainder of this thesis is structured the following way:

Chapter 2: We will give a brief introduction of cloud computing, virtualization and the architecture of Amazon Web Services. This will give the reader enough background information to understand the problem space we are working in.

Chapter 3: A literature review of related work will be presented covering the areas of virtual machine security and network security analysis.

Chapter 4: In this chapter we will shortly present the existing SAVE prototype for analyzing private clouds.

Chapter 5: We will present our extension of the SAVE prototype for the discovery of public infrastructure cloud configurations. In particular focusing on Amazon EC2.

Chapter 6: In this chapter we will present our main contributions for the security analysis of public infrastructure clouds. We are focusing on multi-tier applications deployed on Amazon EC2. The analysis consists of discovering and verifying reachability and vulnerability properties of multi-tier services, and an evaluation of changes occurring in the configuration in terms of vulnerability implications.

Chapter 7: We will evaluate and present results of the implementation of the analysis methods presented in Chapter 6.

Chapter 8: This chapter contains an outlook of future work and presents questions remaining open.

Chapter 9: We will conclude the contributions presented in this thesis.

Chapter 2

Background

In this chapter we will present the background information required for understanding the remainder of this thesis. The explanation of *Cloud Computing* given in Chapter 1 will be extended with further technical details and clarifications. *Virtualization*, the fundamental technology of Cloud Computing, will be explained from a technical perspective including the various forms of virtualization depending on the virtualized resource. We will present the underlying architecture of Amazon Web Services (AWS) as an example of the architecture of a public infrastructure cloud provider, and for becoming familiar with the architecture for later analysis purposes.

2.1 Cloud Computing

In this section we will clarify and explain the different kinds of cloud computing, although we will not deal with the technical details which will be presented in Section 2.2.

2.1.1 Service Types

Cloud computing is a broad term combining several different types of service offerings. In general we distinguish between *Software*, *Platform*, and *Infrastructure* as a service, which are offered by the cloud provider. The main focus of this thesis lies on *Infrastructure as a Service*, also called *Infrastructure Clouds*, but for comparison reasons the other types of offerings are also briefly presented.

Software as a Service (SaaS) is the most visible of the three service types, because end-users typically interact with this service directly and perceive that they are more common than the other service types, which are typically hidden from the end-users. Part of SaaS are web applications like web-based email or project management, for instance, Google's Gmail and Salesforce.com Customer Relationship Management (CRM).

Platform as a Service (PaaS) is a new kind of offering where a platform is provided for customers to deploy their applications. The platform provides a complete application stack, therefore the customer is not required to maintain its own server with the components for the application stack, thus reducing its maintenance costs. Furthermore, the platform itself is often built to ensure scalability and fault-tolerance of the deployed applications, i.e., the platform can cope with usage spikes, which might overwhelm the server infrastructure operated by an individual customer himself.

Common examples for PaaS are the Microsoft Azure Platform [Mic10] (.NET application stack), Google's App Engine [Goo10] (Java and Python), and Heroku [Her10] (Ruby on Rails).

Infrastructure as a Service (IaaS) offers basic infrastructure resources, like computing, networking, and storage, to the customers, therefore provides the most flexibility and freedom of choice. Popular providers of IaaS are Amazon Web Services, RackSpace, and GoGrid.

Computing resources are provided in the form of virtual machines. The customer creates a disk image of an operating system installation containing all his required services and software, deploys this image to the IaaS provider, and spawns an arbitrary amount of instances based on this image. The customer can customize its image to an arbitrary extend, therefore has the most flexibility in terms of application stack and platform. Unlike in the PaaS model, the customer has to maintain its platform resulting in additional maintenance and administration costs. IaaS is only advantageous compared to PaaS if the additional flexibility is leveraged. The number of instances can be adjusted depending on the workload of the application, i.e., increase the number of instances if the application faces a traffic spike and decreases when the workload has normalized again. This presumes that the application is designed with vertical scalability in mind, i.e., the application can leverage an additional number of instances, which can be a challenge for the application developers.

Different networking solutions are offered depending on the cloud provider. GoGrid and RackSpace use hardware-based VLAN separation for isolating different customer networks. Amazon uses a software-based approach realized with packet filters called *Security Groups*. Customer virtual machines can be part of a security group and the associated policy applies to inbound traffic for these VMs. Furthermore, Amazon restricts the traffic a VM receives and therefore prevents packet sniffing. Another networking option offered by Amazon, and in preparation at GoGrid, are Virtual Private Networks, or *Virtual Private Clouds* in Amazon's nomenclature. The VMs in the VPN/VPC are separated from the publicly reachable instances on the network and are connected through a VPN-gateway with an existing enterprise network. It enables enterprises to seamlessly extend their network into the cloud and leverages its computing and storage resources.

Storage can be provided in different ways varying among the multiple IaaS providers. Four different forms can be identified in the currently available providers: NAS-like, SAN-like, API-based data objects, and Virtual Machine storage. A virtual machine has typically a fixed-size data storage available, which is equivalent to a harddisk in a regular desktop or server computer. In some cases this type of storage is only intended to be used for temporary data and is itself non-persistent, i.e., after the machine terminates the data is lost. NAS-like storage, like GoGrid Cloud Storage, is accessible from the VMs on a file-based level using standard protocols like CIFS. Amazon Elastic Block Store (EBS) is a SAN-like storage type, which appears to the VM as an additional block-device. An EBS volume can be attached to different VMs, but not to multiple VMs simultaneously, and the size can be adjusted presuming the filesystem on the block-device is resizable as well. The last type of storage is accessible through an API and holds data objects up to a specific size, e.g., in the range of several gigabytes. This is a very scalable kind of storage, i.e., one can store an arbitrary amount of objects, and also provides the possibility of distributing these objects using a Content Distribution Network offered by the provider. Examples of this kind of storage are Amazon Simple Storage Service (S3) and RackSpace CloudFiles.

2.1.2 Cloud Types

Along the different service types cloud computing can be divided into, there also exists different types of clouds. In this section we will discuss a simplified model of dividing clouds into the categories *Public*, *Private*, *Community*, and *Hybrid*, as presented in [MG09b]. We will also briefly discuss a more complex model, the *Cloud Cube Model*, which takes several different properties into account.

Public Clouds are offered by a cloud provider to the general public as a service and typically hosts multiple customers on shared resources. The example providers described previously, e.g., Amazon Web Services, all provide public clouds.

Private Clouds are operated only for a specific customer either on- or off-premise by the customer itself or a third-party operator. Potential security problems with multi-tenancy on shared resources in public clouds, e.g., side-channel attacks, are avoided in the private cloud offering.

Community Clouds are providing services for a limited set of customers from a specific community, e.g., an organization, which have similar requirements. They are either operated by in-house staff or a third party, and hosted either on- or off-premise.

Hybrid Clouds are the results of the composition of private and public clouds. For example a customer can run his critical and security-sensitive processes on an in-house operated private cloud, but outsource certain non-critical processes to a cheaper public cloud.

The Cloud Cube Model is presented in [Jer09] and categorizes clouds based on four criteria:

- external or internal
- proprietary or open
- perimeterised or de-perimeterised
- insourced or outsourced

The first criterion determines if the resources of the cloud are located on- or off-premise. The second one identifies potential interoperability based on the technology used. Open technology allows interoperability between different cloud providers and proprietary technology can lead to a vendor lock-in. The third criterion is used to determine if the cloud resources are part of the customers network perimeter or not. Virtual Private Networks, e.g., Amazon's VPC, can transform a typically de-perimeterised cloud to a perimeterised one. The last criterion states if the cloud is operated by in-house staff or a third party. These criteria can be visualized as shown in Figure 2.1.

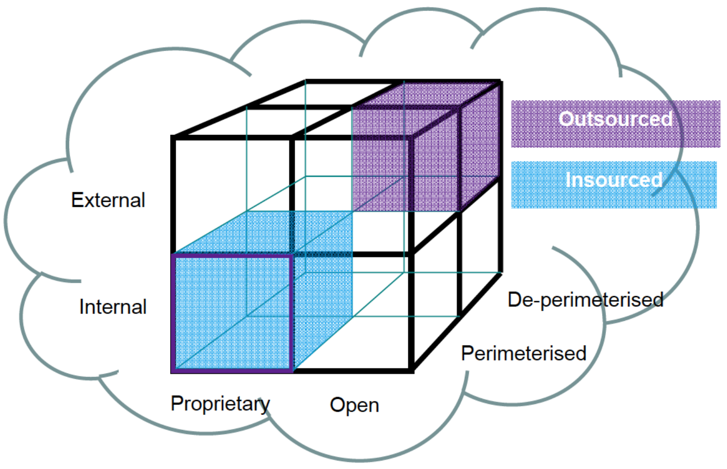


Figure 2.1: The Cloud Cube Model (©Jericho Forum, [Jer09])

2.2 Virtualization

Infrastructure clouds are driven by two major technological components: virtualization of resources and management software. In this section we will briefly explain how the virtualization of different resources is performed, in order to get an insight into the underlying technology of cloud computing.

Virtualization abstracts from physical resources in a way that several virtual resources are multiplexed on a physical one. The virtual resources are isolated from each other and allow higher utilization in multi-tenancy environments. Virtualization exists for various forms of resources, but in this section we will only focus on the resources relevant for cloud computing in the form of IaaS: *Machine*, *Network*, and *Storage*.

2.2.1 Machine Virtualization

In Chapter 2.1 we already explained that computing resources in infrastructure clouds (IaaS) are typically consumed using virtual machines. Using virtualization, a physical machine is divided into several virtual ones each running its own operating system. The system providing the abstraction of the hardware and managing the virtual machines is called Hypervisor or Virtual Machine Monitor (VMM). Examples for such VMMs are Xen [BDF+03], VMware ESX [VMw09], and KVM [Qum06].

Machine virtualization can be implemented using different techniques [VMw07]. *Full Virtualization* allows an unmodified operating system to run in a virtual machine by leveraging either methods for virtualization provided by recent CPUs, e.g., Intel VT-x or AMD-V, called hardware-assisted virtualization or using binary translation, i.e., modifying instructions of the operating system to be suitable for running in a VM. Binary translation is becoming less common than CPU-assisted full virtualization due to performance reasons and the prevalence of CPUs with full virtualization capabilities. Another technique called *Paravirtualization* only allows a specially modified operating system, which is aware that it is being virtualized, to run in a VM. This technique generally yields the best performance results, but proprietary operating systems like Microsoft Windows can not be modified and have to be run using full virtualization.

Xen and VMware ESX are using both paravirtualization and hardware-assisted full virtualization. This hybrid approach yields best performance results for operating systems which can be modified, and allow proprietary operating systems to run as well using full virtualization. KVM on the other hand uses mainly hardware-assisted full virtualization for all operating systems and only has a limited support for providing paravirtualized devices, e.g., network or block devices using virtio [Rus08].

2.2.2 Network Virtualization

Network virtualization can be applied to different components in the networking area. The physical network itself can be abstracted and divided into several virtual networks by different technologies including among others: *VLAN*, *MPLS*, *ATM*, or *VPN*.

The physical network interface can also be virtualized by creating multiple virtual interfaces for the different virtual machines running on a physical machine. The virtual interfaces are connected to a virtual switch, e.g., this can be a bridge device on a Linux setup, which interconnects the virtual machines locally and typically also provides connectivity to external networks through the physical interface. The virtual interfaces can also be combined with VLAN tagging to provide isolation on the network level.

In recent development, Cisco now provides a virtual switch implementation (Cisco Nexus 1000V series [Cis10]), which unifies the management of physical and virtual switches, and also provides advanced functionality compared to regular bridge devices. The OpenSolaris Crossbow project also provides advanced network virtualization to the OpenSolaris platform including functionality for QoS and bandwidth management. In case the physical network interface supports virtualization, Crossbow can directly assign hardware resources to the virtual interface [TDSB09], which increases performance compared to conventional software-based approaches.

2.2.3 Storage Virtualization

Storage virtualization can be applied either locally on the physical harddisk of the machine or on storage provided via network. The most common method of storage virtualization for locally attached harddisks is to use logical volumes on top of the disk. Linux Logical Volume Manager (LVM) is a way of managing and creating logical volumes which can be used as backend storage devices for virtual machines. Unlike partitions of a harddisk, logical volumes can be flexibly created, resized and deleted. Another, but less efficient, method is to use a loopback device which provides a block-device backed by a regular file on the filesystem.

For network-based storage, i.e., Storage Area Network (SAN), block-devices are exported over the network and the consumer does not know how the block-device is backed on the provider side. For example in Sun's ZFS (Zettabyte File System), logical volumes called ZVOLs can be created in a storage pool based on multiple physical harddisks, which are then exported via iSCSI to a consumer. Another example for block-devices exported via a network is GNBD (Global Network Block Device), which is part of Red Hat's cluster suite and suitable for creating a cluster file system using Red Hat's Global File System (GFS).

2.3 Amazon Web Services Architecture

Amazon Web Services is a collection of IaaS offerings, which includes the Elastic Compute Cloud (EC2) for computing, Simple Storage Service (S3) and Elastic Block Storage (EBS) for storage, and Virtual Private Cloud (VPC) and Security Groups for networking. Further services are offered, but they are not of particular interest for our scenario. In this section we will present the underlying architecture of the previously mentioned services. This section is mainly based on [Ama09b, RTSS09] and information obtained through XenStore, a configuration repository on Xen accessible by virtual machines. Further details about the underlying architecture can be found in Appendix A.

2.3.1 Compute

The *Elastic Compute Cloud* is Amazon's service infrastructure cloud which allows customers to deploy and run virtual machines on Amazon's infrastructure. Virtual machines, also called *instances* in EC2, are provisioned from a machine template called Amazon Machine Image (AMI). A machine image contains an installation of an operating system and services required by the customer. Typically, virtual machines are directly connected to the Internet and protected by a firewall-like concept called *Security Groups*.

The machine virtualization is based on a highly customized Xen hypervisor using paravirtualization [Ama09b]. EC2 also allows Windows-based instances, which implies that for these instances they have to use full virtualization. The Xen management domain is based on a version of Linux.

Fault Separation: Amazon EC2 provides fault separation by using multiple geographic regions, which are further divided into Availability Zones. Availability Zones can be considered as physical independent data centers located in one region. At the time of this writing, Amazon provides four regions: on the west and east coast of the US, one in Europe, and one in Asia Pacific with at least two availability zones per region.

2.3.2 Network

In the typical usage scenario every VM will have one private and one public IPv4 address dynamically assigned. Exceptions to this are *Virtual Private Clouds*, which only have one IPv4 address from a user-defined IP range assigned, and *Elastic IP*, which assigns a static public IPv4 address to a VM.

According to [Ols08], a VM only has one network interface with the private address attached to it and NAT is used to map the public address to the private one. The Xen networking setup is a routed one and does not use a bridge device as a virtual switch. Besides using NAT on the IP layer, they also employ NAT on the MAC addresses. Every packet leaving or entering a VM will have `EF:FF:FF:FF:FF:FF` as its MAC address.

Security Groups are Amazon's concept for a set of inbound firewall rules associated with a name [Ama09b]. Outbound traffic is not restricted by a security group and always allowed. A virtual machine can be a member of one or multiple security groups, i.e., the traffic for that particular VM is allowed based on the union of rules specified in the associated security groups. Members of the same security group can only communicate with each other if explicitly allowed in the rules set.

The rules of a security group are applied in the management layer of the host, i.e., the firewalling is done outside the VM. Security groups can be used to simulate security perimeters like a DMZ or internal servers when using other security groups as sources in the rules set. The default firewall policy is *Deny*, therefore all rules in a security group are *Accept* rules. Rules can allow traffic based on protocol (TCP, UDP, ICMP), port range, and source (IP range or another security group).

Packet Spoofing & Sniffing is prevented by a packet filter running in the management layer of the system, where all packets are passing through. The packet filter drops all packets from a VM, which do not have the source IP or MAC address associated with that particular VM. We have to assume that the management layer is aware of the L2 and L3 addresses of all VMs running on that particular host, and therefore sets up *iptables* and *ebtables* rules to prevent packet spoofing. Furthermore, sniffing packets destined for other VMs running on the same host can not be achieved by a VM, even though the network interface can be placed into promiscuous mode. We conclude that further *iptables* and *ebtables* rules exist, which only allow packets transporting to a VM if they have the correct destination addresses.

Intrusion Detection is implemented on each host, at least to some degree, which allows the detection of port scans performed by a VM. Port scans violate the Amazon EC2 Acceptable Use Policy (AUP) and are automatically detected and blocked.

Virtual Private Clouds (VPC) is a concept of integrating cloud resources into the existing enterprise IT infrastructure [Ama10b]. VMs running in the VPC are isolated from the Internet and other resources of the cloud, and have an address assigned from a user-defined network range. The VPC is connected via a VPN gateway to the enterprise network. Routing information between the VPC router and the enterprise router are exchanged, which allow a seamless connection between the enterprise and cloud resources.

2.3.3 Storage

Amazon Web Services provide different types of storage for the VMs. In this section we will explain the different types and their individual advantages and disadvantages.

Instance Storage is a non-persistent storage attached to each VM locally, which has the highest performance of all three available storage types. The storage is only temporary and will not be available to a VM after termination, but remains available to a VM after reboots. The instance storage is visible to a VM as three partitions for root, swap, and extra storage space. Host-based storage virtualization on Linux commonly uses LVM or loopback devices, where the former has the better performance characteristics. Based on information obtained from XenStore, the root partition is backed by a loopback device and the other two partitions are based on logical volumes. The partition for extra storage space also utilizes copy-on-write for the possibility of efficient wiping of the volume after the VM terminates, i.e., only the modified areas have to be wiped instead of the whole volume.

Elastic Block Store (EBS) is a persistent storage with high performance, availability and reliability. When an EBS is attached to a VM it will appear as a block device in the VM. EBS has a high reliability property, although it is not as reliable as S3, because EBS data is only replicated within its own availability zone. The characteristic of EBS indicates that it is based on a SAN setup and XenStore information confirm that it is using GNBD. Further details about EBS are available at [Rig08].

Simple Storage Service (S3) is the most reliable storage of the three different types provided by Amazon Web Services, because the data is replicated among multiple data centers. The structure of S3 is object-oriented, i.e., all data is stored in objects of up to 5 gigabytes of data, which are organized in so-called buckets. S3 can be used to store snapshot of EBS volumes, hereby acting as a highly reliable backup mechanism for EBS.

Chapter 3

Literature Review

In this chapter we will cover the literature of two research areas: virtual machine security and network security analysis. In the former case we will discuss a broad spectrum of security challenges and solutions for virtual machines, and also briefly cover security management in virtualized environments. Since network security is a very broad area, we are focusing on two particular aspects namely reachability analysis and vulnerability assessment using attack graphs.

3.1 Virtual Machine Security

Infrastructure clouds make significant use of virtualization and the clouds provide computational resources which are consumed by the means of virtual machines. Due to this strong connection between these two technologies, security problems associated with virtual machines will have an impact on the overall security of infrastructure clouds. Therefore a review of existing literature on the topic of virtual machine security will give us a useful foundation for analyzing the security of infrastructure clouds and provides an insight in the underlying security challenges.

3.1.1 Overview

A comprehensive overview of virtual machines and their corresponding security challenges and benefits is presented in [GR05], which we will summarize in this section and use as a thread for further details of the exposed challenges and solutions in the remaining sections.

Virtual machines provide a high degree of flexibility by allowing users to easily create, copy, snapshot, rollback, and migrate them. This flexibility results in major adoption of virtual machines by users for different purposes, e.g., for testing of software or configurations using snapshots and the rollback mechanism.

Security Issues

The authors of the paper extracted the following list of security issues related to virtual machines.

Scaling represents the problem that users now have multiple virtual machines, e.g., for testing and development purposes, instead of a very few number of physical ones. Therefore the total number of machines drastically increases within one organization and the workload on the security systems will increases accordingly. **Diversity** in operating systems, OS versions and patch levels increases the complexity in the security management of the infrastructure. VMs typically result in a high diversity, because users have multiple snapshots of VMs and testers can have a collection of different VMs.

Transience is another security issue induced by the flexibility of virtual machines. It mainly deals with the problem that VMs appear and disappear very rapidly in the network which makes security management, e.g., patch management and vulnerability scanning, very difficult. The authors describe this as the missing of a steady state in the network, where the steady state means that all machines are patched and properly managed. The **Mobility** of a VM, i.e., the VM can easily be copied or migrated, imposes multiple security problems: all the hosts, the VM will be executed on, have to be part of the trusted computing base (TCB); sensitive information can leave a security perimeter or malware is introduced, and the theft of VMs can easily be done by simply copying a file.

The traditional **Software Lifecycle**, i.e., a monotonic forward progress of the software state, is broken by virtual machine's snapshot and rollback mechanisms, because the execution of the virtual machine can be forked and be rolled back. In particular the rollback mechanism induces a lot of problems regarding freshness of randomness sources used for cryptographic protocols or critical patches are removed by a rollback. Limited **Data Lifetime**, e.g., for sensitive or cryptographic information, can be compromised due to the rollback mechanism and that the content of the virtual machine's memory might be stored on the disk of the host due to paging, snapshots, or migration.

In traditional computing environments, the **Identity** of a machine is often deduced from a properties like the MAC address, the location, or Ethernet port. Virtual machines however typically use dynamically created MAC addresses and they might migrate from one physical host to another, therefore properties like the location or Ethernet port will change, and make it difficult to assign an identity.

Solutions Directions & Security Benefits

The authors of [GR05] propose two directions to solve the previously described security challenges: introducing an ubiquitous virtualization layer and having Virtual Machine Monitor (VMM) assurance. Such a virtualization layer is based on a high assurance VMM and provides security management and policy functionality. For example, firewalling or anti-virus detection can be performed in the virtualization layer, and policy enforcement can control the VM's mobility and usage. The role of the VMM is to isolate the VMs from each other and the correctness of enforcing this property is crucial, therefore a high assurance VMM is required.

Introducing an extended virtualization layer that overtakes functionality originally performed in the guest operating systems has a certain number of benefits. Users do not have to worry about security management, e.g., firewalling or anti-virus detection, if these mechanisms are provided by the virtualization layer and are operated by a central administration staff. Furthermore, these security services are now independent of the guest operating systems, which results in a higher flexibility because a high diversity of VMs can be securely managed. Regarding the security issue associated with software lifecycle and the rollback feature, the virtualization layer could provide mechanisms to store such sensitive information and to provide strong randomness.

3.1.2 VMM Security

The security of the Virtual Machine Monitor (VMM) is crucial, because it provides the necessary isolation between the hosted VMs and typically runs with the highest privileges on the system.

Introducing a new software layer, such as the one providing virtualization, inherently increases the complexity of the system, which also increases the possibility of software security vulnerabilities. A study of the security of virtualization software presented in [Orm07] revealed a variety of vulnerabilities in the most common virtualization implementations. Such vulnerabilities in the VMM can lead to the break of isolation, i.e., a VM can access another VMs resources. In [Woj08], the author presents a way of exploiting a vulnerability in the Xen virtualization software which gave him access to the management domain of Xen and thereby access to all other VMs.

Different solutions exist to mitigate security problems in the virtualization layer which are based on principles of building secure software: formal verification, security by isolation and disaggregation, and reducing the trusted code size.

An interesting example for formal verification of a software, which is also relevant for our topic of VMM security, is the *seL4* project [KEH+09], a formally verified L4 microkernel. The proof verifies that the implementation in the C programming languages matches the abstract specification of the system and implies that certain software vulnerabilities, like buffer overflows and null pointer dereferences, are absent in the implementation. Microkernel and VMM are very similar (cf. L4Linux [Hoh96], [HL10], [HUL06, HWF+05]), therefore either a formally verified microkernel acting as a VMM can be used, as presented in [vT10], or adapting the formal proof for VMMs, although the size of existing VMMs make formal verification very difficult.

Another approach of improving the security of the VMM is to reduce the complexity and trusted code base (TCB) by means of decomposition. An approach for extracting the domain builder functionality of the Xen dom0 into a separate domain was presented in [MMH08]. With a separate domain builder VM, the user-space of dom0 can be removed from the TCB, because no privileged functionality for VM construction and management need to be exposed to user-space applications, e.g., `xend`. However, in their current state the dom0 kernel is still part of the TCB due to required interaction with physical I/O devices. Besides the dom0 kernel, the Xen hypervisor and the domain builder are part of the TCB.

The recent prototype operating system Qubes OS [RW10] implements, among other security features, disaggregation of Xen dom0 by establishing driver domains which are limited to a specific hardware resource by the means of IOMMU as implemented by Intel VT-d, i.e., the VMM monitors DMA requests and can possibly restrict them. These driver domains can run with limited privileges and the overall complexity of dom0 can be reduced. Thereby a software vulnerability in one of the drivers will not result in a break of isolation when running in a non-privilege driver domain compared to running in dom0. IOMMU would also benefit the disaggregation of Xen using a domain builder VM, because the dom0 kernel could be removed from the TCB when I/O with physical devices is offloaded to driver domains.

The virtualization architecture NOVA [SK10] uses a minimal microkernel, with a size similar to the formally verified seL4, and provides virtualization functionality as user-land applications. Therefore the amount of high privilege code is reduced to a minimal microkernel-based hypervisor.

3.1.3 Information Leakage

The multi-tenancy of virtualization and in particular cloud computing in form of infrastructure as a service, faces the scenario that an attacker will share the same physical resources as other tenants. This sharing of resources could lead to information leakage due to known or unknown covert channels.

A very interesting approach was presented in [RTSS09] which consists of a method for predicting the placements of VMs in the Amazon cloud and discussing potential side-channels and their implications. The placement is in particular interesting for attackers who target a specific victim and want to place a VM on the same physical server. Placing a VM on the same physical server, i.e., establishing co-residency, will allow further attacks using side-channel vulnerabilities to extract potentially sensitive information about the other VM. Work on side-channel vulnerabilities leading to the exposure of cryptographic keys were presented in [Per05] (Intel HyperThreading), [AKS07] (CPU branch prediction), and [Aci07] (I-Cache exploiting).

The paper presents two attacks: estimating traffic rates and a keystroke timing attack. In the first case, an attacker could measure the website traffic of a competitor, thereby gaining information about the website's popularity. The attack uses a cache load measurement and identifies a correlation between traffic rate and the load samples. The second attack tries to identify keystrokes in another VM, which could be used to guess the typing depending on the keystroke intervals, e.g., to identify a typed-in password. The attack uses again the cache load measurements to identify keystrokes, but they are evaluating the attack on a private virtualized host rather than Amazon EC2.

Another problem which could lead to information leakage is a result of the rollback functionality of VMs. Due to the rollback, cryptographic protocols could be affected either due to reuse of keys or reusing the same random numbers generated in an earlier run of the protocol. In [RY10] the issue of randomness problems in cryptographic protocols is discussed and it is shown how such a problem can be exploited in the case of TLS. In order to mitigate the problem related to randomness in virtual machines, they propose a framework for securing existing protocols by the means of hedged cryptography, which means that cryptographic protocols will provide a weaker security notion in the presence of a bad randomness source [BBN+09].

3.1.4 Remote Attestation

Remote attestation tackles the problem of assuring that a remote platform consists of a trusted set of hardware and software resources. This is typically done in open distributed systems to ensure that the other peers are not running malicious software [MMJZ06]. Potentially, remote attestation can also be used to identify VMs, which was pointed out as a problem in the overview section, however privacy concerns lead to an anonymous way of remote attestation called Direct Anonymous Attestation (DAA) [BCC04].

A discussion of different means of attestation for operating systems is given in [Eng08]. The existing attestation methods are presented: code signing with public-key cryptography, small and attestable microkernel/VMMs with late-launch capabilities, property based attestation, semantic attestation, and read-only operating system images. Furthermore, new methods for attestation are presented consisting of specialized OS images, OS authentication using birth certificates, and virtual machine policy attestation.

The *Terra* [GPC$^+$03] architecture is based on a trusted VMM (TVMM) which provides two different execution contexts for VMs: open box and closed box. The first one is equivalent to a regular general-purpose hardware platform and the second one resembles a special-purpose platform which is typically found in closed systems like mobile phones and game consoles. The closed box environment provides, among other capabilities, remote attestation for assuring remote parties about the integrity of the hardware and software stack of a VM.

In case of Terra, the attestation process covers the system firmware, bootloader, TVMM, and the VM. The attestation of the VM distinguishes between two different kind of VM storage: attested and unattested storage. The VM owner can specify the VM's storage kinds on an application specific case. The authors propose two different methods for handling attestable storage. The first method is *Ahead-of-Time Attestation* which attests the entire VM, including attestable storage, and verifies its correctness before executing the VM. This is suitable for small, high-assurance VMs due to the performance of hashing in the startup process. The other method is *Optimistic Attestation* where blocks of the disk are hashed on-demand, and in case of a verification failure of any block the VM execution is stopped immediately.

The usage of attestation in the area of cloud computing, i.e., trusted cloud computing was presented in [Kra09] and [SGR09]. The problem is that for cloud consumers the IaaS providers operate a black box and the consumer does not have any insights about the underlying security of the architecture. The architectures presented in the two papers use attestation to assure that a known and trusted software stack is in use and provides adequate security for the VMs. For example, [SGR09] makes use of a trusted VMM as presented earlier.

An example for the usage of attestation in a real-world application is Enomaly's Elastic Computing Platform, which is a virtualized systems management software, in the *High Assurance Edition* [Coh10].

3.1.5 Virtual Machine Introspection

In the overview section it became clear that many security services can be offloaded from the VMs to the virtualization layer. This has numerous benefits as presented earlier like delegation of security management and OS independence. We will now present two approaches of providing an Intrusion Detection System (IDS) and a rootkit detector using inspection capabilities of the virtualization layer. Examples for inspection capabilities are VMware's VMsafe [VMw10] and Xen's XenAccess [Xen10, PCL07].

The IDS and rootkit detector can leverage certain properties of the VMM for their advantage according to [GR03]. *Isolation* will prevent an attacker, who compromised a VM, to tamper with the IDS. In a traditional computing environment, attackers can usually tamper with the anti-virus system or Host IDS since it is running on the same machine. *Inspection* allows the security monitoring services to inspect all the states of a virtual machine, i.e., CPU registers, memory content, I/O devices states etc. A malicious application or an attacker will have enormous difficulties in evading the monitoring of the IDS. Furthermore, *Interposition*, which is already implemented in the VMM to execute triggers upon the execution of privileged instructions in a VM, can be used to monitor specific instructions of a VM.

The IDS implementation presented in [GR03] consists of three modules: a VMM interface, an OS interface, and a policy engine. The VMM interface allows the IDS to read a VM's memory and set and receive notifications on VM operations. The OS interface provides an insight into the operating system's data structures and information, e.g., process list. In their current implementation they use the Linux crash dump analysis tool `crash` and the ELF binary information tool `readelf` for obtaining symbol information from the kernel. The policy engine consists of a framework which provides a higher level abstraction on the VMM and OS interfaces, and a set of policy modules implemented to check certain behavior and monitor a VM. Examples for the policy modules are a *user program integrity detector* and a *memory access enforcer*. The first one is a polling-based one and periodically hashes the immutable sections of a program in memory, e.g., of a SSH daemon, in order to detect modifications due to a backdoor attempt. The other module is event-based and is triggered in case an attempt is made to modify certain areas of the kernel, e.g., the system call table.

A rootkit detector was presented in [CSS$^+$09], which is capable of identifying the operating system of a VM without prior knowledge and can discover the appropriate data structures after the OS is identified. Monitoring the discovered data structures allows the detection of rootkits, in case unauthorized modifications of essential data structures happen.

3.1.6 Policy Enforcement

In the overview we discussed problems related to the transience and mobility of VMs, namely the difficulty of detecting compromised machines and the possible leakage of sensitive information or infection of the corporate network due to the migration of infected VMs. An approach to counter these problems is presented in [MGHW09] using so-called *Virtual Machine Contracts* (VMC). Infected VMs can be detected in case the behavior of the VM differs from the specification in the contract, e.g., a botnet malware opens a new port for receiving instructions which is not part of the contract. *Virtual Network Access Control* can be implemented using VMCs, e.g., a VM is only allowed to connect to a restricted network to obtain patches in case out-of-date software is detected. In a similar manner, regulatory compliance, i.e., validating the usage of encryption, security services etc. in the VM, can be implemented using VMCs.

A project dealing with the improvement of isolation between VMs and providing a fine-grained mandatory access control mechanism for inter-VM communication and resource sharing is *sHype* [SJV+05]. A sample policy which can be enforced by sHype is a *Chinese Wall policy*. This kind of policy is useful for administrators to prohibit that certain VMs, e.g., two competitors or workloads with different criticality, are running on the same machine and are potentially exposed to side-channel vulnerabilities.

3.1.7 Strong Isolation in Virtualized Environments

Strong isolation is crucial for the security of virtualized environments due to their multi-tenancy and the sharing of common physical resources.

The *Trusted Virtual Datacenter* (TVDc) project [BCP+08] aims at providing strong isolation and integrity guarantees to virtualized environments. Virtual machines and their resources are grouped using an abstraction called Trusted Virtual Domains (TVD) [GJP+05], which could be based on the owner of the workload, e.g., TVD_α for customer A and TVD_β for customer B. The isolation of the TVDs is realized using the sHype hypervisor [SVJ+05] on the machine level and VLANs on the network level. Similar work for providing isolation on the network level using different methods, e.g., VLANs, VPN, and Ethernet encapsulation, was presented in [CDRS07].

The idea of Virtual Private Clouds (VPC) for isolation in infrastructure clouds was presented in [WSG+09]. A VPC is typically isolated from other cloud consumers, and seamlessly and securely integrated into an existing enterprise infrastructure, e.g., using a VPN tunnel. The benefit is to leverage resources from the cloud provider and integrate such resources in the existing enterprise infrastructure, while preserving strong isolation on the cloud provider side. This method is currently offered by Amazon for the infrastructure cloud [Ama10b], however the underlying physical resources are still shared among multiple tenants and subject to potential side-channel vulnerabilities.

3.2 Network Security Analysis

In this section we focus on three different aspects of network security analysis. First, we consider performing reachability analysis of a network in a static manner and discuss advantages of static analysis in this context. We then present work in the field of infrastructure discovery and automated analysis and transformation using VLANs. Finally, we discuss network security analysis based on attack graphs and how attack graphs can be used to compute security metrics for the current network configuration.

3.2.1 Static Network Reachability Analysis

This section summarizes the work presented in [XZM+04].

Analyzing the reachability of a network is useful for verifying the intention of the network designer that certain host in a network should be able or not to communicate with each other. The dynamic approach to reachability analysis could be as simple as performing ICMP echo requests or trace routes from one host to another. The authors argue that a static approach is desired and present a model for calculating the potential network reachability in consideration with dynamic influencing factors like routing protocols, packet filtering, and packet transformation. The relevant parts of this work will be presented in this section.

The analysis is performed on the configuration files of the routers in the network, which are assumed to be stored in a central repository for backup purposes.

Advantages of Static Analysis

The main problem with dynamic reachability analysis is that only the current snapshot of the network is analyzed, i.e., only the current selected route and link is considered. The route may change or a link may fail and the verified reachability does not hold anymore. Furthermore, packet filtering or transformation of the current path may allow ICMP echo requests, but may filter all other traffic, therefore alters the result of the reachability analysis.

From a practical point of view, it is not feasible to perform a reachability analysis using ping or traceroute in a large network, since the complexity is quadratic with regards to the number of hosts, i.e., each host has to ping all other hosts in the network. A static approach has the advantage that the reachability can be verified for numerous different packet types and therefore can take into account potential packet filtering and transformation. A static analysis can be performed on a desired or current configuration, which allows a network designer to analyze new network configurations or verify properties of the current configuration.

Reachability Analysis

The model presented in the paper also considers changes of the network due to dynamic routing protocols, in order to be able to verify reachability even if links fail. This is not relevant for our analysis from a security point of view, where we are mainly interested in the effect of packet filters on the reachability. Therefore, I will only presented a simplified model without consideration of dynamic network changes.

Consider a graph $(V, E, \mathcal{F})$ where V is the set of routers, E is the set of directed edges defining the connectivity between routers, and $\mathcal{F}$ is a labeling function for annotating the edges, e.g. $F_{u,v} \in \mathcal{F}$ represents the flow policies for packets from router u to router v. $R_{i,j}$ denotes the reachability from router i to router j and is the subset of packets the network will transport.

Since we are not interested in dynamic network changes, the reachability can be calculated easier than demonstrated in the paper:

$$R_{i,j} = \bigcup_{\pi \in \mathcal{P}(i,j)} \bigcap_{<u,v> \in \pi} F_{u,v}$$

where $\mathcal{P}(i,j)$ denotes the set of all loop-free paths from i to j in the physical network topology. The $R_{i,j}$ is equivalent to the upper bound estimator $\hat{R}^U_{i,j}$ presented in the paper when ignoring the effect of routing protocols. This models only the effect of the packet filters and it contains all the packets for which at least one allowed path in the network exists. The lower bound estimator contains the packets for which all paths are allowed:

$$\hat{R}^L_{i,j} = \bigcap_{\pi \in \mathcal{P}(i,j)} \bigcap_{<u,v> \in \pi} F_{u,v}$$

The paper presents algorithms for computing the lower and upper reachability bound estimators. For the lower bound, they remove all edges which can not be part of the path from i to j, followed by the intersection of $F_{u,v}$ for the remaining edges. The computation of the upper bound is closely related to computing the transitive closure.

Verification of Network Isolation

The reachability upper bound can be used to verify that two customers, say A and B, are isolated from each other under any circumstances. The isolation is given iff $\forall i \in A, j \in B : \; R^U_{i,j} = \emptyset$. There exists no path from any host i in customer A's network to any host j in B's network, and vice versa.

Further Work

The previously described work on static reachability analysis found further advancements in the papers presented in [KL09b, KL09a]. They consider a more complex model including different kinds of packet transformations, a graph based on routers and subnets, and propose algorithms for computing reachability and solutions for reachability queries.

3.2.2 VLAN Configuration Automation

The focus of the work presented in [KSS+09] is the automated analysis of VLAN configurations for performance and security reasons. The complexity of these configurations can become immense in large enterprise or university network environments, therefore requires automation tools for analyzing the configuration. Common misconfigurations in VLAN settings can result in redundant or missing network links, which can induce performance and security issues.

They propose a set of algorithms to support network operators in VLAN configuration tasks, and visualization and validation of the configurations. The algorithms process a graph of the network, which is constructed using the network configuration files of routers and switches, and network link information for the L2 topology obtained from sources like the Cisco Discovery Protocol (CDP). The individual algorithms are very simple and straightforward, e.g., applying graph traversal or shortest-path algorithms.

Although they mention security issues induced by redundant links due to an increased possibility of ARP poisoning attacks, their main focus is to tackle performance issues and support the network operators in configuring switches.

3.2.3 Attack Graphs

Attack graphs are a way to model network risks in a network using a graph-based approach, where nodes represent a possible attack state, e.g., user privilege escalation, and the edges represent a way of changing states, e.g., using an exploit with certain pre- and post-conditions [SPG97, PS98]. An elaborate review of literature on attack graphs from the year 2005 can be found in [LI05].

Attack graphs can be constructed by security experts, however this manual approach becomes very difficult due to the scalability of the problem. Therefore, an automated approach for the construction and analysis of attack graphs is desired. In [SPEC01] an early tool for the construction and analysis of attack graphs was presented. It uses manually obtained information about attacker and exploit capabilities, and automatically gathered information about the machine configurations and vulnerabilities, in order to construct an attack graph. A shortest-path for the graph is computed which represents the most likely path an attack will take. A similar analysis using shortest-path was also discussed in [SPG97, PS98]. An approach based on symbolic model checking for automatic and efficient graph construction was presented in [SHJ+02]. The authors of [SW04] demonstrate a toolkit for the analysis and construction of attack graphs.

Visualization of reachability, attack graphs, and attack paths is very important for administrators in order to effectively leverage these tools and apply the obtained results for improving the security of the network. Sample work in the visualization of attack graphs can be found in [WLI08, WLI07].

Topological Vulnerability Analysis

One of the most comprehensive approaches in attack graph construction and analysis is *Topological Vulnerability Analysis* (TVA) presented in multiple publications [JLSW09, JN07]. The main idea is to analyze dependencies of attack exploits, in order to find paths of an attacker to compromise specific targets in the network. Information from vulnerability scanners, which provide detailed information of isolated vulnerabilities, are gathered and combined. Based on the attack graphs they can propose changes in the configuration to increase the network security, and an interactive graph visualization tool can help administrators in finding network problems.

Only relying on a vulnerability scanner can limit the insights gained into the possible vulnerable services of a host. For example, client-side vulnerabilities are not covered by vulnerability scanners. Therefore, a new approach was presented in [NEJ+09], which correlates information of a asset management database with a vulnerability database. For instance, a host is running a specific version of a web browser, which is noted in the asset database, and the vulnerability database contains information about that particular version of the web browser.

Several limitations of TVA were pointed out in [LI05]:

- Exploit information, i.e., pre- and post-conditions, entered by hand
- Firewall and router rules not automatically imported
- Poor scaling to large networks
- Requires low-level attack details

Furthermore, a general problem of using vulnerability scanners on production systems is that the probes of the scanner can cause problems and damages on the systems, e.g., due to aggressive behavior of the probes.

Security Metrics using Attack Graphs

Security metrics are important in order to be able to judge network security based on quantitative data [Jaq07]. Using security metrics derived from attack graphs, one can make decisions on the present vulnerabilities and risks in the current network configuration.

A formal framework for calculating security metrics based on attack graphs was presented in [WSJ07a, WSJ07b]. The basic idea is to measure attack resistance, i.e., the resistance of each type of exploits regarding effort and time, and to provide a framework for calculating cumulative resistances with defined composition operators.

A probabilistic approach for computing security metrics using attack graphs was presented in [WIL+08]. The idea is to convert vulnerability ratings into probabilities, i.e., the probability an attacker can and will execute a certain attack to change from one attack state to the next. They present how to compute cumulative scores even in the presence of complex attack graphs with cycles.

Another way of approaching the problem of deriving security metrics from attack graphs is given in [MBZ+06]. The authors adapted the Google PageRank algorithm [BP98], which was invented for ranking the popularity of websites based on the number of incoming links, to attack graphs. Each node in the ranked attack graph will have a rank computed by the PageRank algorithm, and the total rank of all nodes can be used to compute a security metric. However, the model of the PageRank algorithm is not intuitively adaptable to attack graphs, because it considers a user which will abort a path at any given time with a certain probability, and starts a new path at a random other node. A focused attacker will typically not abort its attack path at any given time, but only when the path turns out too difficult or the desired state is reached. The PageRank for attack graphs model is more suited for automated attacker, e.g., a worm exploiting random machines.

Machine learning, and in particular graph neural networks (GNN), are used for applications like website ranking. As we showed earlier the adaption of the Google PageRank algorithm to attack graphs, it is not surprising that machine learning/GNNs are used for the ranking of attack graphs as presented in [LSNH+09].

Chapter 4

Existing Prototype: *SAVE*

The existing prototype built by IBM Research for performing security audits of virtualized systems is called *SAVE*. SAVE stands for Security Audits of heterogeneous Virtual Environments and focuses on virtualized environments where the auditor has full access to the underlying configuration.

In this chapter we will briefly explain the functionality of the existing prototype, since we will built upon it and extend it for infrastructure clouds. SAVE consists of a discovery component, where the configurations of the virtualized systems are gathered and translated into a data model called *Realization* model, which basically represents a detailed view of the low-level configuration. The second component is the analysis of the low-level configuration stored in form of a realization model. Currently the analysis is focused on identifying isolation properties of resources, e.g., to verify that two different tenants do not have common resources. A simplified model, called the *Logical* model, can be constructed based on the analysis, which represents the resource associations of tenants.

4.1 Configuration Discovery

The configuration discovery phase is able to gather low-level configuration information from various types of systems including:

- Xen
- VMware
- IBM pSeries
- LibVirt
- Linux

The way these configuration information are obtained differs between the systems. Xen and VMware provide an API which allows the gathering of such information. IBM pSeries servers are controlled by a centralized administration host from which the configuration of all managed servers can be retrieved. For LibVirt and Linux based systems, we obtain configuration information by performing a SSH login on these systems and execute qualified commands.

The discovery phase takes as input a list of hosts and associated credentials in form of a XML file. The credentials are either for a SSH login in case of pSeries, LibVirt, and Linux, or API credentials in case of Xen and VMware. For each host in the list, we try to run the discovery probes for the various systems mentioned previously, and in many cases we can terminate the discovery earlier upon successfully executing certain probes, e.g., if the Xen probe succeeded then the host can not have a VMware hypervisor installed and we can skip the VMware probe. The output of the probes for all hosts are gathered in a discovery XML file used for the translation into the realization model.

The translation phase consists of multiple translation modules, one for each of the supported systems, which are able to translate the system-specific low-level configuration into the generic realization model. The complete discovery information are translated into one realization model which can be used for the analysis of the discovered systems.

4.1.1 Realization Model

The realization data model for expressing the low-level configuration of the various virtualization systems is given in Appendix B.1. In this section we will give a brief overview of the model without going into the specific details. Basically the model can be subdivided into four sections: physical machine, virtual machine, storage, and network.

Physical Machine: In the upper-left part of the model is the physical machine related configuration expressed. The main focus is the physical machine, the hypervisor, and the management operating system. Physical devices are also associated with the physical machine.

Virtual Machine: The virtual machine related configuration is expressed in the upper-right part of the model. Depending on the virtualization system, the VM can be of different type and is associated with a physical machine, and storage and network resources.

Storage: The lower-right part of the model represents the storage configuration. It differentiates between a front- and back-end device. The front-end can be system specific, e.g., a VMware disk, and the back-end can be provided by multiple sources. Typically the back-end is based on a file used as a disk image or a block-device. Both can be locally or served over the network, e.g., by a NAS or SAN provider.

Network: The part of the model expressing the network related configuration is the most complex one and in the lower-left part of the model. The main components of this part are network interfaces and switches connecting interfaces through ports both in physical and virtual forms.

4.2 Analysis

The analysis of the realization model is currently focused on identifying isolation properties and uses a graph coloring algorithm. The input of the analysis is the realization model and a specification for the coloring. The specification defines the nodes to start the coloring from and which colors to use, and the rules for the following of edges in the coloring process. For example a node with a certain property X should have the seed color blue and a node with property Y always red. We then define in the specification how the colors are propagated from these two seed nodes to the rest of the graph, e.g., always retain the color when traversing from node with property X to a node with property Z.

4.2.1 Logical Model

After the coloring of the realization model is completed, we can construct a logical model from the colored realization model, which focuses on the resource usage within the color domains and how the domains are isolated from each other. Resources with the same color are grouped together in a logical domain, and in case one resource has multiple colors, a possible intersection of logical domains exist, i.e., an isolation break. The logical model is illustrated in Appendix B.2.

4.3 Examples

In this section we will give a selection of examples in order to illustrate the functionality and capabilities of the SAVE prototype. We will demonstrate all three steps of the process, namely the configuration discovery and representation in the realization data model, the analysis of the realization model using graph coloring, and the transformation of the analyzed model into a logical representation.

4.3.1 Simple Realization Model

A simple example for a realization model constructed from a Xen discovery probe is illustrated in Figure 4.1. Since only the Xen discovery probe was used in this example, the produced realization model is very simple but nevertheless illustrates the general idea of the output of the SAVE discovery process.

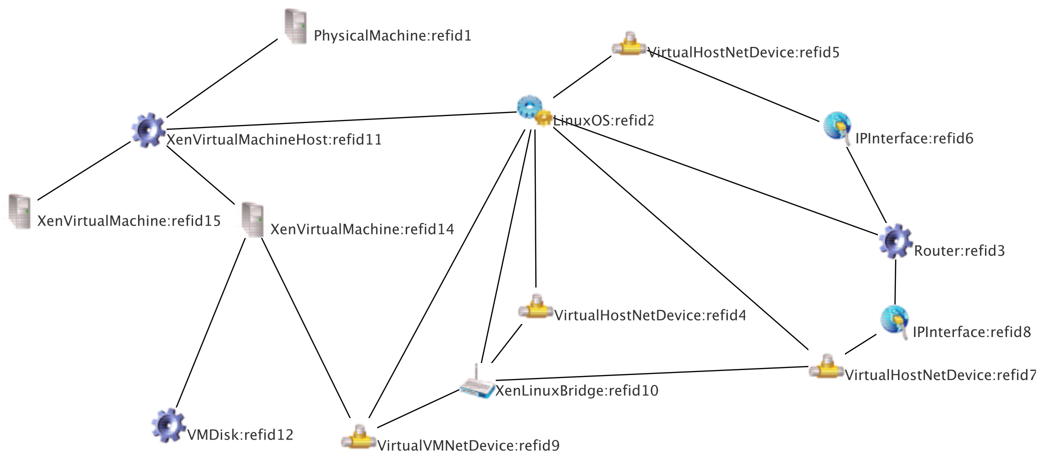

PhysicalMachine:refid1
VirtualHostNetDevice:refid5
LinuxOS:refid2
IPInterface:refid6
XenVirtualMachineHost:refid11
XenVirtualMachine:refid15
XenVirtualMachine:refid14
Router:refid3
VirtualHostNetDevice:refid4
IPInterface:refid8
VirtualHostNetDevice:refid7
XenLinuxBridge:refid10
VMDisk:refid12
VirtualVMNetDevice:refid9

Figure 4.1: Simple Realization Model Example

It basically shows one physical server running the Xen hypervisor with two virtual machines hosted on that machine, where one virtual machine is the Xen dom0. The other VM is connected to two types of resources: storage in form of a VMDisk and a network device. The LinuxOS node is the management OS of Xen running in the dom0 VM and is providing networking to the VMs in form of a bridged setup, i.e., all VMs are connected to a bridge device within dom0. Since the model is constructed from data only provided by a Xen probe, we do not know how the storage to the VM is provided in the Linux OS, for which we would require the information from the Linux probe.

4.3.2 Complex Colored Realization Model

This example demonstrates a realization model constructed from the discovery of a real-world VMware server. Due to space reasons, the coloring policy for the analysis of the model and the colored model itself are presented in Appendix C.1.

The blue domain is the management domain controlling the management operating system and virtual machine monitor of that particular physical server. The red domain contains the virtual machines and their resources, which are using the network filesystem located in the lower right of the model. Based on these two seed nodes, i.e., the management OS for blue and the filesystem for red, the rest of the realization model is colored according to the given coloring policy.

4.3.3 Logical Model

Figure 4.2 illustrates the logical model obtained from the previously described analyzed realization model, but reduced to one logical domain, i.e., the red domain.

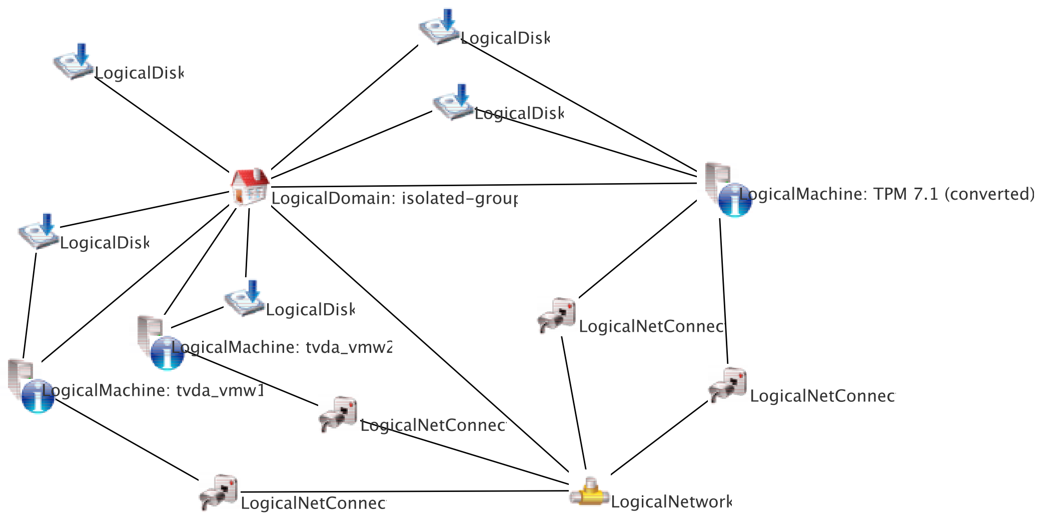


Figure 4.2: Logical Model Example

The logical model illustrates the isolation properties for the discovered VMware server configuration. Since the red logical domain is completely separated from other logical domains, e.g., the blue one, the two domains are isolated from each other and do not share common resources.

Chapter 5

Configuration Discovery for Public Infrastructure Clouds

In this chapter, we describe what kind of information we can discover in Amazon EC2 and in which degree we have to extend the existing data model, in order to cope with configurations from public infrastructure clouds and in particular for Amazon EC2. The existing data model is illustrated in Appendix B and explained in Chapter 4.

5.1 Discovery & Data Model for Amazon

The first step in the audit process is to obtain the current configuration from Amazon and transform it into a model suitable for later analysis. Amazon provides an API for management purposes, which also includes the functionality of extracting the current configuration. We are interested in the following subset of the available information:

- Instances
- Volumes
- Snapshots
- Security Groups
- Regions
- AvailabilityZones
- VPCs
- Subnets
- VpnConnections
- VpnGateways

The configuration is provided in XML format and we transform it into the data model illustrated in Figure 5.1. The data model represents the configuration deployed at Amazon with particular focus on virtual machines and their resources.

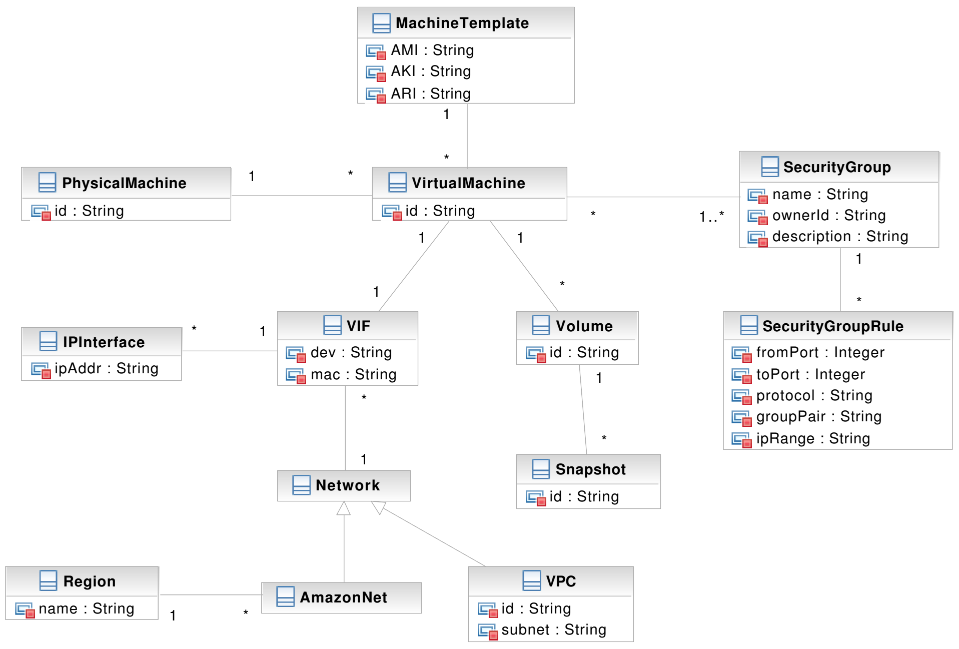


Figure 5.1: Amazon Data Model

A *VirtualMachine*, or *instance* in Amazon nomenclature, is provisioned by a *MachineTemplate* given by an AMI (Amazon Machine Image), AKI (Kernel Image), and ARI (Ramdisk Image). The VM is running on a physical machine which we determine using heuristics based on the IP naming scheme of Amazon. A VM is a member of one or multiple *Security Groups*, which have a set of inbound firewall rules. Furthermore, a VM can have network and storage resources attached in the form of *VIFs* (Virtual InterFaces) or *Volumes* respectively.

In Amazon's case, the VM only has one VIF which has a private and implicitly a public IP addresses attached. The VIF is either attached to an abstract *AmazonNet* or *VPC* (Virtual Private Cloud). The AmazonNet is equivalent to an Amazon availability zone, i.e., a network within one data center, and part of a certain geographical *Region*. Besides local storage, a VM can also have network storage volumes attached to it, which also provides the functionality of snapshots.

5.2 Integrated Model & Discovery for Infrastructure Clouds

The isolated data model for Amazon configurations has to be integrated into the generic data model used within the SAVE project, in order that the SAVE prototype can handle both public and private clouds using the same underlying data model. In this section we will briefly discuss the integration of the models and discovery implementation.

5.2.1 Data Model

The Amazon data model illustrated in Figure 5.1 was integrated into the existing SAVE realization model. The integrated model is presented in Appendix B.3. Since Amazon is based on Xen, we extended the Xen virtual machine host and virtual machine classes for the Amazon case. The rules of Amazon Security Groups are integrated into the generic filtering rules part of the networking part of the model, and a VM is a member of one or more Security Groups. We consider an AmazonNet as a network switch, i.e., all VMs are connected on the same AmazonNet switch for a specific availability zone. At the current state of the model, the VPC functionality has not be integrated yet, because we do not consider hybrid clouds yet. Since the data model already contained a class for machine templates used for the provisioning of VMs, the Amazon Machine Template just extends this class with further attributes. Volumes in Amazon can either be local disks, i.e., instance storage, or remote disks, i.e., EBS volumes.

5.2.2 Discovery

The discovery probe for Amazon Web Services is implemented using the Amazon EC2 Java library (for API version 2009-11-30 [Ama10c]) in less than 170 lines of Java code. The translation from the discovery data in XML into the integrated realization model is also implemented in Java in around 330 lines of code, which contains all the Amazon specific part related to mapping discovery data to classes in the data model.

Chapter 6

Multi-tier Virtual Infrastructures in Public Clouds

For hosting enterprise-grade applications and infrastructures, it is important that common security architectures can be established in the public cloud. One common security architecture is to split complex applications into multiple tiers that correspond to security zones, such as Internet, demilitarized zone (DMZ), and intranet.

This chapter shows how multi-tier security architectures can be implemented in a public cloud. Using Amazon EC2, we focus on how the security of such multi-tier architectures can be assessed. In the first part of this chapter we will present our approach that discovers the actual configuration of such architectures and then validates it against a desired configuration. Furthermore we assess the vulnerability of such a configuration using attack graphs and recommending potential improvements.

In the second part we compare two methods of deploying multi-tier virtual infrastructures on Amazon EC2 and outline the offered isolation levels for both methods. In the final part, we propose a method for detecting and evaluating configuration changes in multi-tier virtual infrastructure deployments. This method can be used to track the security levels of such architectures and indicate degradations of the security.

6.1 Configuration Security Audits

6.1.1 Introduction

Today, public clouds such as Amazon EC2 are used to host multi-tier infrastructures. Such infrastructures, e.g., comprise interconnected web, application, and database servers that may then be synchronized with databases in the enterprise. While this approach provides scalability, it exposes private personal or critical company data to attacks.

In order to mitigate this risk, security concepts similar to today's well-known security zones have been introduced. The so-called security groups of Amazon allow users to group machines while restricting communication through firewall-like rules. Nevertheless, the resulting configurations can be complex and thus error prone. According to [Woo04], more than half of 37 analyzed corporate firewall configurations contained 9 out of 12 possible mistakes or problems, therefore we can assume a similar number defects when using firewall-like concepts in the cloud.

In this section, we propose a novel approach towards assessing the actual security of such multi-tier set-ups and the corresponding security policies. We focus on Amazon's EC2 as one example of a public cloud. We first show how the correct configuration of Amazon's security groups can be visualized and validated. We then proceed by assessing the resulting threats using attack-graphs that show the risk of attacks for a given multi-tier set-up. Finally, we show how a given multi-tier set-up can be automatically transformed such that the functionality remains while the complexity and vulnerabilities are reduced.

Scenario

Throughout this section we will use the same scenario to illustrate the different audit and analysis methods. We consider an example configuration of a multi-tier web application widely used in real-world deployments consisting of web, application, and database servers. The web servers are reachable on the two common web server ports 80 (http) and 443 (https) over TCP from any source. The application servers are only reachable on an application specific port, e.g., 8080 TCP, from the web servers. Furthermore, the database servers are only reachable from the application servers on port 3306 (mysql) TCP. For maintenance purposes, all servers allow ssh access (22 TCP) from the corporate network, e.g., 1.2.3.4/24, and the servers accept ICMP packets from any source.

6.1.2 Reachability Audit

In this subsection we will discuss the audit of security group configurations with regard to reachability, i.e., analyzing the information flow allowed by the configuration. Visualization of the allowed information flow and a reachability query language are presented which can support the administrator in developing new configurations and in discovering potential mistakes in the current configuration. A policy language and automated analysis are shown for the periodic verification of the configuration correctness.

The current configuration of the security groups and related information, e.g., the security group membership of VMs, are obtained in the configuration discovery phase described in Section 5.1.

Reachability Graph

The visualization and the later proposed automated analysis is based on a directed multi-graph constructed from the configuration information obtained through the Amazon API. The vertices of the graph represent the set of sources and security groups defined in the configuration. The edges denote the allowed information flow specified in the rules of a security group between the sources and that particular group. For example, security group *web* allows the Internet, i.e., *0.0.0.0/0*, to access on port 80 TCP. The graph would consist of two vertices for the security group and Internet source IP with an directed edge between them labeled *80/tcp*.

Visualization

Visualizing the reachability graph is useful for manual inspection of the correctness of the current security group configuration. According to [Goo07], visualization takes advantage of vision, the highest bandwidth input device, and of the human perceptual abilities to make anomalies obvious to the user.

Listing 6.1 shows the output of the `ec2-describe-group` command, which displays the current security group configuration and is part of the command-line management suite of Amazon EC2. Even in such a simple example, it is difficult to assess the correctness of the given configuration. Amazon allows up to 100 defined security groups with multiple filter rules per group, which would result in a highly complex configuration that is even more difficult to evaluate.

In contrast Figure 6.1 illustrates the visualization of the reachability graph of the same configuration. In our opinion, the visualization can be more intuitively understood and judged for correctness. The multi-tier structure of

```
GROUP 1234  app application server
PERMISSION  1234  app ALLOWS  tcp 8080  8080  FROM  USER  1234↩
      GRPNAME web

GROUP 1234  db  database server
PERMISSION  1234  db  ALLOWS  tcp 3306  3306  FROM  USER  1234↩
      GRPNAME app

GROUP 1234  web web server
PERMISSION  1234  web ALLOWS  tcp 80  80  FROM  CIDR  0.0.0.0/0
PERMISSION  1234  web ALLOWS  tcp 443 443 FROM  CIDR  0.0.0.0/0

GROUP 1234  default default group
PERMISSION  1234  default ALLOWS  tcp 22  22  FROM  CIDR  ↩
    1.2.3.4/24
PERMISSION  1234  default ALLOWS  icmp  −1  −1  FROM  CIDR  ↩
    0.0.0.0/0
```

Listing 6.1: `ec2-describe-group` Command Output

the security groups is immediately obvious to the auditor and the external sources, i.e., IP ranges, are clearly pointed out.

Potential misconfigurations can easily be spotted in the visualization. For example, if the database security group would allow any source to access the database, rather than just the application group, it can easily be detected during the inspection due to an edge between the vertices *0.0.0.0/0* and *db*.

Query & Policy Language

The visualization of the security group reachability is only useful for a manual inspection of the correctness of an initial security groups configuration. Afterwards, a periodic verification of the configuration against a policy specification is desired, in order to retain the correctness of the configuration, because changes in the configuration over its lifetime might cause violations with regard to its original intentions. Furthermore, queries can be used to answer questions about the reachability of the current configuration, e.g., for resolving reachability problems.

Query Language: Reachability queries are specified in the following form: `from` s `to` d `port` p `proto` p'. s is either an IP address (specified as a single address or IP range), a security group, or *any* for matching all sources. d is either a security group or *any*. p can be one specific port or a port range $p_1 - p_2$. Both are transformed to a tuple (p_1, p_2), where in the former case

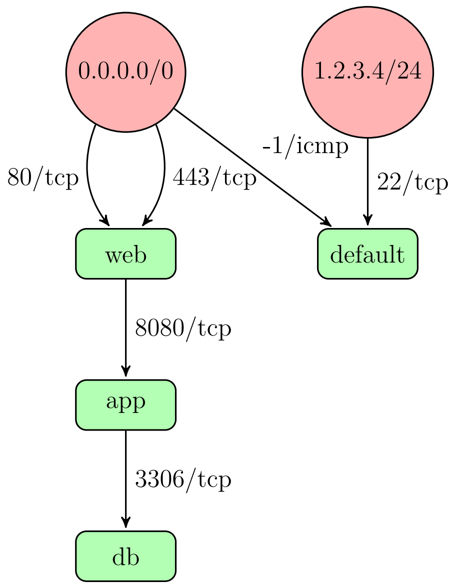


Figure 6.1: Visualization of Security Groups Reachability

$p_1 = p_2$. Valid values for p' are $tcp, udp, icmp$. p and p' can be set to *any* in case one is not interested in the specific port or protocol.

Policy Language: For the policy language we will consider two cases. A *never* policy specifies a reachability which should never be established between a source and destination. An *only* policy allows only a specific reachability, i.e., the given information flow is allowed but any other flow is considered a violation of the policy. *never* policies are specified similarly to queries: `never from` s `to` d `port` p `proto` p'.

For an *only* policy we have to extend this syntax slightly to allow the specification of multiple port and protocol pairs per source and destination: `only from` s `to` d `port` p_1 `proto` p'_1 `and` $\ldots$ `and port` p_n `proto` p'_n. Consider as an example a web server group which should only be reachable to port 80/tcp and port 443/tcp: `only from 0.0.0.0/0 to web port 80 proto tcp and port 443 proto tcp`.

Reachability Analysis & Policy Verification

The processing and verification of the reachability queries and policies is realized using two algorithms which are explained in the following.

Reachability Analysis: The algorithm to process the reachability queries is given in Algorithm 1. We consider a sample input query: `from` s `to` d

`port` $p_1 - p_2$ `proto` p', where the values $(s, d, (p_1, p_2), p')$ are passed as input parameters to the algorithm. The algorithm returns $True$ if the reachability specified in the query is established for a given reachability graph $G_R = (V, E)$.

Algorithm 1: Process a Reachability Query

```
Input: Reachability Graph $G_R$, Query $(s, d, (p_1, p_2), p')$
Output: $True$ or $False$ if query matches $G_R$
$E' \leftarrow \emptyset$
foreach $e \in E$ do
    if ($s = any$ or $s \subseteq e.source$) and ($d = any$ or $d = e.destination$)
    then
        $E' \leftarrow E' \cup \{e\}$
foreach $e \in E'$ do
    $c \leftarrow e.constraint$
    if ($p' = any$ or $p' = c.proto$) and ($p_1 = any$ or ($c.p_1 \leq p_1$ and
    $p_2 \leq c.p_2$)) then
        return $True$
return $False$
```

In the first step, we construct a set of edges E' containing all edges between the given source and destination vertices. Since a source can be specified as an IP range, we have to check if the given source s is a subset of the source of the edge, e.g., 10.0.0.0/24 $\subset$ 0.0.0.0/0. In case the source of the edge and s are security groups, $\subseteq$ acts as an ordinary equality operator. Otherwise, when either the source or destination is specified as *any*, we will include the edge regardless of the source or destination respectively.

In the second step, we compare the constraints of the edges in E' (indicated by *e.constraint*) with the constraints specified in the query. *any* for the ports and protocol short circuits the check, i.e., the query is only interested in an edge between s and d. In the other case, the protocols must be equal and the port range in the query enclosed in the port range of the edge constraint.

Policy Verification: The process of the verification of reachability policies leverages the previously described query processing algorithm. For *never* policies in the form `never from` s `to` d `port` $p_1 - p_2$ `proto` p', we simply convert them to a query $(s, d, (p_1, p_2), p')$ which has to evaluate to $False$. Otherwise the given policy is not satisfied.

In case of an *only* policy like `only from` s `to` d `port` $p_{1,1} - p_{1,2}$ `proto` p'_1 `and` ... `and port` $p_{n,1} - p_{n,2}$ `proto` p'_n, we have to do a two step process for the verification. First we convert the policy into n queries of the form

$(s, d, (p_{i,1}, p_{i,2}), p'_i)$ which all have to evaluate to $True$. Otherwise the reachability specified in the policy is not satisfied. Furthermore, we have to verify that other information flows are not possible in the reachability graph for the particular source and destination, i.e., queries of the complement $(s, d, \bar{p}, \bar{p}')$ have to evaluate to $False$. This exclusiveness of the reachability specified in the *only* policy is checked with Algorithm 2.

```
Algorithm 2: Verify Exclusiveness of an only Policy
  Input: Reachability Graph $G_R$, Policy
         $(s, d, [((p_{1,1}, p_{1,2}), p'_1), \ldots, ((p_{n,1}, p_{n,2}), p'_n)])$
  Output: $True$ or $False$ if the policy is satisfied
  [Construction of $E'$ equal to Algorithm 1]
  foreach $e \in E'$ do
      $B \leftarrow [True, True, \ldots]$
      $c \leftarrow e.constraint$
      for $i = 1$ to $n$ do
          if $(p'_i \neq any$ and $p'_i \neq c.proto)$ or $(p_{i,1} \neq any$ and $(c.p_1 < p_{i,1}$
          or $p_{i,2} < c.p_2))$ then
              $B_i \leftarrow False$
      if $True \notin B$ then
          return $False$
  return $True$
```

The construction of the set of relevant edges E' is equal to the part in Algorithm 1. For each edge in this set, we initialize an array B, with a size equal to the number of port-protocol pairs specified in the policy, with $True$ values. For each such pair, we check the edge constraint if it allows further information flow than already allowed by the pair. For example the policy requires only port (p_1, p_2) but the edge constraint allows $(p_1 - 1, p_2 + 1)$, therefore allows further information flow with two more ports and thereby violates the *only* policy.

If the policy specifies *any* for the port or protocol then we can skip the constraint check. Otherwise, if the protocols of the policy and edge constraint are different, or the port range of the edge constraint is larger than the one specified in the policy, we mark a violation in the array B by setting the array slot corresponding to the port-protocol pair to $False$. After checking all pairs for a particular edge constraint, we test if B does not contain any $True$ value. If this is the case, then the policy is violated, because none port-protocol pair matched the edge constraint. In case of a non-violation, at least one pair would have matched and resulted in a $True$ value in B.

The periodic verification uses a policy specification consisting of a set of policies, which all have to evaluate to *True*, in order that the verification is successful and the configuration does not contain any violations.

6.1.3 Audit using Attack Graphs

While the technique presented in the previous subsection targeted single edges, we now extend this approach to inspect the whole graph. Since pure reachability is a rather weak security measure, we extend the edges with a weight of how likely it is that they will be vulnerable to an attack. This results in a kind of attack graph [TASB07, WLI07, WLI08]. The audit of security configurations using these attack graphs is concerned about the impact of security group rules with regard to services security, and is based on the previously presented reachability analysis.

Attack Graph

An attack graph for security groups consists of vertices based on IP ranges and AMIs (Amazon Machine Images), where the information flow between the vertices is given by the rules of the security groups the VMs — provisioned from the AMIs — are members of. Furthermore, the edges are labeled with a severity rating for the service running in the AMI and allowed by a security group rule.

For example, assume that *VM 1* is a member of security group *web* and provisioned from *AMI 1*. There exists a rule in *web* that allows *0.0.0.0/0* to access on port *443/tcp*. Suppose the web server running in *AMI 1* has a known vulnerability, then there would be an edge between *0.0.0.0/0* and *AMI 1* in the attack graph with a label *443/tcp medium*.

Graph Construction

The previously described attack graph can be automatically constructed in three steps.

The first step is to establish the relationship between AMIs and security groups. Using the Amazon API, we can obtain the currently running VMs, including information on the AMIs used for provisioning them and the security groups they are members of. Typically, the number of different AMIs per security group should be rather small, because the role of the security group is reflected by a specific AMI, e.g., multiple instances of a web server AMI would be member of the *web* security group. In our example we assume the existence of four different AMIs; *AMI 1* is a member of security group *web*, *AMI 2* of *app*, *AMI 3* and *AMI 4* of *db*. Furthermore, all AMIs are member of *default*. Figure 6.2 illustrates the relationships between security groups and AMIs for our scenario.

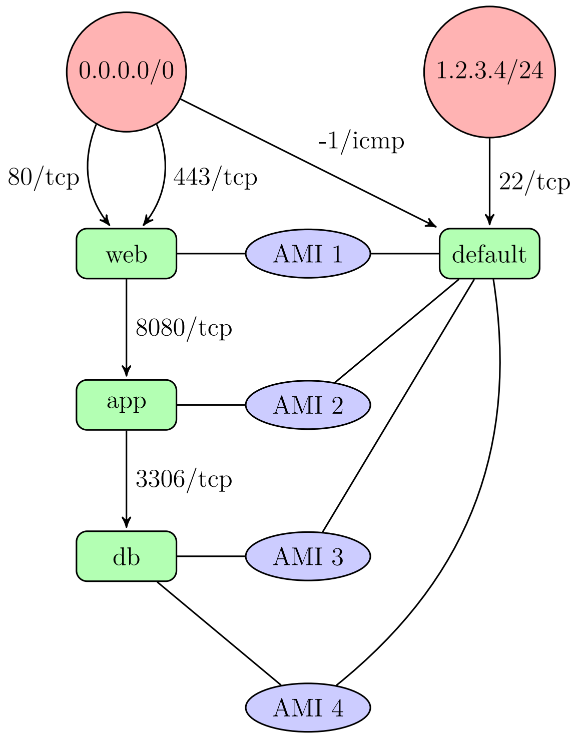


Figure 6.2: Relationship between Security Groups and AMIs

The second step in the construction is to obtain a severity rating for the services running in the AMIs. Using Amazon EC2 we can start each AMI in a separate VM for analysis purposes, thereby not affecting the production systems. The AMI's security from an external point of view can be determined in the VM using vulnerability scanners like Nessus [Ten10]. Further information such as patch levels and version numbers can be obtained from an internal point of view by logging into the AMI instance. For each port of an AMI we thus obtain a severity rating from a domain of vulnerability ratings. Here we use the range of *Low*, *Medium*, or *High*. For typical applications this level of granularity seems to be sufficient, but a more fine-grained rating can be achieved using Common Vulnerability Scoring System (CVSS) [MSR07].

The final step in the construction is to combine the previously obtained information with the reachability graph of security groups. A security group is replaced by all the AMIs related to that particular group, and the edges

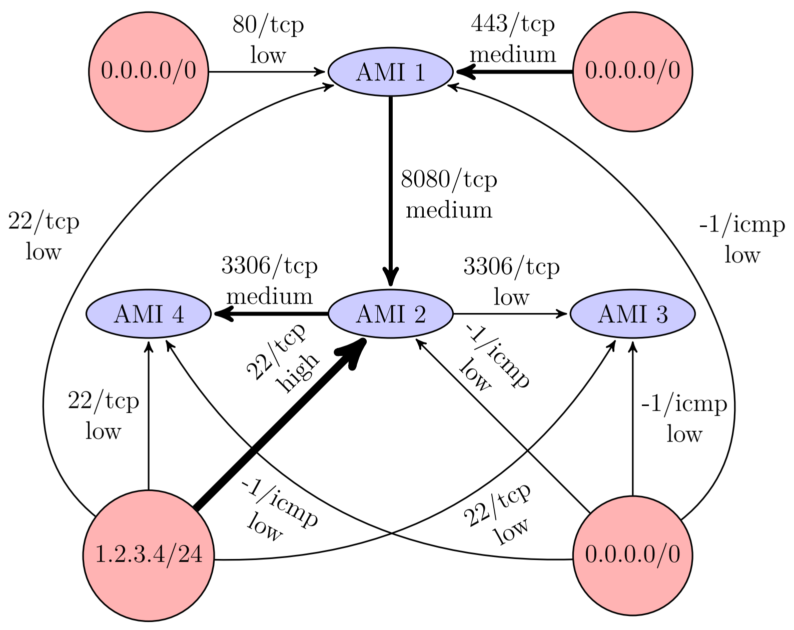


Figure 6.3: Attack Graph of the Multi-Tier Application

between the sources and the AMIs are additionally labeled with the severity rating for the port and protocol associated with each edge. Figure 6.3 illustrates the attack graph for the example scenario where thicker edges represent higher vulnerability ratings.

In the current configuration, an attacker in the corporate network could compromise a VM of the application server group, which are provisioned using *AMI 2*, through a vulnerability in the ssh service. Afterwards, further attacks can be launched against the medium rated mysql service running on instances of *AMI 4*, potentially compromising the database.

Query & Policy Language

When analyzing attack graphs constructed from cloud configurations, one is particularly interested in the *weakest* path from one vertex to another. The weakest path is the shortest path with the highest vulnerability rating, i.e., the most likely path an attacker would take to compromise a specific resource, because less vulnerabilities with high severity ratings are more likely to be exploited successfully.

Query Language: Reachability queries are specified in the following form: `from` s `to` d `vuln` v. We drop protocols and ports in the query, since we are mostly interested in the vulnerability, not how an attack is performed. s is either an IP address (specified as a single address or IP range), an AMI, or *any* for matching all sources, and d is either an AMI or *any*. The vulnerability value v can be any value from the domain of vulnerability ratings, or the special value *any*. The result of such a query is a set of shortest paths P from s to d. If *any* is specified for v, then all paths are considered, otherwise only paths with a minimum vulnerability higher than or equal to v are considered.

Policy Language: For the policy language we consider the same cases as in Subsection 6.1.2. As before, a *never* policy specifies an unwanted connection between a source and a destination. They are specified similarly to queries: `never from` s `to` d `vuln` v, where s, d, and v can have the same values as for queries. The interpretation of a *never* policy is that there may never be a connection with a vulnerability rating *higher* than or equal to v.

Unlike before, *only* policies are very similar to *never* policies when dealing with attack graphs. This is because we only want to restrict the vulnerability ratings allowed on connections between nodes. These policies have the form `only from` s `to` d `vuln` v, that is the only difference is the initial keyword. Again, s, d, and v can have the same values as for queries. The interpretation of an *only* policy is that there may only be connections with vulnerability ratings *lower* than or equal to v.

Weakest Path Algorithm & Policy Verification

The analysis of queries on the attack graph, and the testing of attack graphs against policies, is performed by means of Dijkstra's shortest path algorithm on a weighted graph. The weight of the edges is based on the vulnerability rating where the weight relation is the following: $High < Medium < Low$. Since Dijkstra's algorithm will determine the shortest path with the lowest weight, we will obtain the most likely path an attacker would take. In case the query or policy contains a vulnerability parameter unequal to *any*, annotated by $\overline{any}$, we have to transform the attack graph before performing the Dijkstra's algorithm by removing all edges with a vulnerability rating lower than the one specified.

Based on the four different combinations of query/policy source and destination parameters, i.e., $(\overline{any}, \overline{any})$, $(\overline{any}, any)$, $(any, \overline{any})$, and (any, any), we have to perform a different variation of the Dijkstra's algorithm. For the case $(\overline{any}, \overline{any})$, i.e., a specific source and destination, the Dijkstra's algorithm can be terminated upon finding the shortest path for the destination. In the other case where the destination is *any*, the regular algorithm will be performed which determines the shortest path between a single source and all other vertices. For $(any, \overline{any})$, i.e., the source is *any*, we reverse the attack graph and start the single source Dijkstra's algorithm from the destination. Due to the reversal of the attack graph, we also have to reverse the paths obtained by the Dijkstra's algorithm. In case of (any, any), all-pairs shortest paths is determined using Dijkstra's algorithm starting from every vertex. Alternatively, the Floyd-Warshall algorithm could be used to determine the all-pairs shortest paths.

The policy processing can be based on the approach of finding the weakest path. A *never* policy basically states the fact that no weakest path with the properties specified in the policy should exist. In case of *only* policies, they state that never should exist a path with a vulnerability *higher* than the one specified. For example, the policy specifies a medium vulnerability, then no path with a high vulnerability should exist.

6.1.4 Security Groups Transformation

The purpose of the transformation process is to reduce the complexity of the configuration by removing unnecessary rules and extracting common ones, and to improve the security by splitting existing security groups.

Splitting of Groups: Based on the attack graph and a specification of the desired services dependency, we can propose a new security group configuration by splitting existing security groups. The splitting process has to balance between increasing configuration complexity and security. One extreme case would be to place each AMI in a separate security group, therefore minimizing the affect of a vulnerability in one AMI to other AMIs. Generally, high or medium rated AMIs are isolated in separate groups, while AMIs with a low or no severity rating can remain in the same group.

Closing Unnecessary Open Ports: Using the AMI analysis of open ports and the rules of the corresponding security group the instances of that AMI are a member of, we can identify unnecessary rules in the configuration if ports are opened in the configuration, but no service is listening on that particular port in the AMI instances. The difference of the set of ports opened by the security group and the set of ports used by the AMIs represents the ports which are unnecessarily opened. These ports can be automatically removed from the security group rules set during a transformation phase.

Extracting Common Ports: A VM can be a member of several security groups and we can leverage this fact to reduce the complexity of the configuration by identifying common ports in the rules of the existing groups and extract them into a separate group. For example, if all groups would allow ssh access from a corporate network and allow ICMP packets, we could automatically extract these rules into a separate group. All instances would be additionally a member of that group. The principle is similar to *Refactoring* found in software engineering, where common functionality is extracted.

6.1.5 Implementation

We implemented the previously described construction of reachability and attack graphs, as well as the processing of queries and policies for such graphs in Python. The implementation, called *SAVEly*, was straight forward given the detailed algorithms presented earlier and it is given in Listing D.1 in Appendix D. We are using the *boto* [bot10] library for obtaining the configuration from Amazon EC2, and the *NetworkX* [Net10] library for the graph handling and shortest-path algorithms.

The attack graph construction is using the OpenVAS vulnerability scanner [Ope10]. For each discovered AMI we spawn a new instance in a specialized *savely_scan* security group which allows all traffic from the scanner machine's IP address, instead of scanning the production servers directly. This approach of scanning will not affect the production servers due to aggressive scanning probes.

Practical results for the tool will be given in Chapter 7.

6.1.6 Improvements for Security Groups

In this section we propose a set of potential improvements for security groups, in order to increase the security of deployments based on such a concept.

Outbound Filtering: An obvious limitation of the current security group concept is the missing functionality of filtering outbound traffic. A VM can contact any host on the Internet which could result in a leakage of sensitive information in case a VM is infected with malware. The current concept is simple and probably good enough for most current customers, but in case of a wider adoption and enterprise-grade deployments, the demand for outbound filtering will raise.

Logging: The management layer does not provide any insights in the kind of traffic which was blocked by the inbound filtering. This information can be useful for setting up intrusion detection systems and initiate prevention of attacks in case of traffic anomalies from a certain source. Providing security logging information in an accessible format could create an ecosystem of third-party security appliances, which monitor such logs and act upon malicious events.

AMI Membership Policies: An AMI can have services installed which do not have proper authentication and authorization mechanism and are intended to run in an isolated environment. An example for such a service is *memcached*, a popular network-accessible caching service used by a large number of websites, which should only be reachable by the web or application servers of a deployment. However, currently an AMI can not be accompanied by a policy specifying the security group properties required for a secure operation of that AMI. In case of the memcached example, the AMI would specify that only other security groups, or more specifically the web or app groups, can access the memcached service.

A similar concept was proposed in [MGHW09] in the form of network contracts, which are part of the concept of Virtual Machine Contracts.

6.2 Comparison of Deployment Methods

In this section we will compare two different deployment methods for multi-tier virtual infrastructures. We consider a scenario, where an existing enterprise IT is extended into a public infrastructure cloud and that the cloud will host a multi-tier application. We investigate the following two methods: a setup using security groups as used in Section 6.1 and one using Amazon Virtual Private Cloud (VPC). In particular we are interested in the security properties of the two deployment methods and what kind of isolation levels they can provide.

6.2.1 Security Groups

Instead of allowing the Internet (0.0.0.0/0) to access the first tier of the multi-tier application used in Section 6.1, we restrict this access to the enterprise network. If all other tiers only allow access from another tier, then the multi-tier application is separated from the Internet and only accessible from the enterprise. The connection between the first tier and the enterprise might need to be further secured using a VPN tunnel, in case sensitive information are transferred between the enterprise and the multi-tier virtual infrastructure. The same applies to inter-tier communication, in case the tiers are located in different Amazon regions or availability zones and traverse the Internet, or in case that the Amazon network is not trusted.

6.2.2 Virtual Private Cloud

Amazon Virtual Private Cloud (VPC) was already briefly introduced in Section 2.3.2. A VPN endpoint is provided by Amazon in order to secure the network communication between the enterprise and the cloud resources using a VPN tunnel. The IP addresses of VMs running within a VPC are user-specified, therefore allows a seamless integration of cloud resources into the enterprise network. The subnet assigned to a VPC can further divided into multiple subnets to host the VMs, which are arranged in a star topology with a router in the middle. Each tier can be placed in a different subnet as part of a VPC. At the time of this writing, only one VPC per customer is allowed.

6.2.3 Isolation Levels

A VPC can be divided into multiple subnets each hosting a different tier of a multi-tier deployment, but the communication between the tiers can not be restricted in any way, because customers do not have access to the router placed in the middle of the subnets star topology. Applying network filtering to only allow certain tiers to communicate with each other is not possible in the current form of the VPC offering. Furthermore, security groups are also not available in combination with VPC, which would be useful with another VPC subnet as a possible source in a security group rule. The only way left is to do filtering based on VPC subnets within the VMs, which inhibits the flexibility of the cloud and could be deactivated by an administrator of the VM or malware.

In case of a security group based setup, the inter-tier communication can easily be configured in a flexible way, and the filtering is performed outside of the VM, therefore it can not be tampered by a VM administrator or malware.

A VPC shares the same underlying physical infrastructure as the rest of the Amazon cloud and is not completely isolated from other customers. Therefore, intra-VPC traffic also needs to be further secured in case one does not trust the network in Amazon data centers.

Since a VPC is isolated in terms of networking, a misconfiguration which would expose an instance of a VPC to the Internet is less likely compared to a security group setup. All external traffic of a VPC passes through the enterprise network and can be filtered and analyzed using existing security infrastructure in the enterprise. For the security group setup, misconfigurations are more likely but can be mitigated using the security audit process described in Section 6.1.

6.2.4 Conclusions

The advantages of a VPC are clearly the seamless integration into an enterprise network by using user-specified IP ranges and an Amazon provided VPN endpoint. The network isolation of a VPC leaves less possibility of misconfigurations, which would expose a VM of a VPC to the Internet. In case of security groups, a misconfigured rule could potentially expose a VM to the Internet, but a security audit process could reduce the possibility of such a problem.

A huge disadvantage of a VPC based deployment of a multi-tier infrastructure is the missing functionality of restricting inter-tier communication on Amazon level. All tiers, i.e., VPC subnets, can freely communicate with each other, if not otherwise restricted by the individual VMs, because customer access to the VPC router or security groups in a VPC are not available at the time of this writing. In contrast, a security group based deployment allows an easy configuration of inter-tier communication restrictions. An alternative by placing each tier in a separate VPC and filter inter-VPC traffic on the customer VPC endpoint is also not possible, because the number of VPCs per customer is limited to one.

In case one tier requires access to external resources or provides a service to hosts not part of the multi-tier application or enterprise network, such traffic would increase the overall traffic costs in a VPC based deployment. We have to take into account the traffic costs between Amazon and the enterprise, and between the enterprise and the external host. In a security group based deployment, such traffic costs are limited to the traffic between Amazon and the external host.

We can conclude that a security group based deployment for multi-tier infrastructures is preferred, due to the limited functionality of VPC regarding inter-tier communication restriction and costs. However, the current VPC offering is still in testing phase, so it might improve until a final stage is reached and might be superior to security groups. Furthermore, for a security group setup, an automated audit process is recommended to mitigate the problems of misconfigurations and accidental exposure of VMs.

6.3 Analysis of Configuration Changes

The configuration of virtual infrastructures deployed on public clouds can be very agile: new machines are added or modified, firewall settings are changed, and the overall complexity increases. We are interested in automatically monitoring these changes with regard to different properties. For example, one could track the overall security of the virtual infrastructure, i.e., the risk of potential security breaches, by monitoring the vulnerability ratings of the deployed services.

In this section we will propose a method for the discovery and evaluation of configuration changes in a multi-tier application deployment. The changes can be evaluated with regard to various properties and we will focus on vulnerability ratings. Another potential property is *expected availability*, with regard to the replication of services, which we will briefly outline.

6.3.1 Discovery of Changes

The first step in the process of analyzing configuration changes is the discovery of changes by comparing two sets of configurations. Consider we have a configuration from time t_1, C_1, and a configuration from time t_2, C_2. We now want to detected the changes between these two configurations.

Changes Reflected in Attack Graphs

In this section we will present the potential changes in the configuration which will have an impact on the security of the system. We will argue that these changes are reflected in the corresponding attack graphs and that attack graphs are a suitable representation of a configuration for a given point in time. Two graphs can be compared in order to detect the security-related changes. In case of an analysis with regard to vulnerability ratings, we will see that attack graphs have all the desired properties for detecting and evaluating changes. Consider the following list of possible changes to a configuration:

- Security Group
  - Adding a new Group
  - Removing a Group
- AMI
  - Adding a new AMI to a Security Group (SG)
  - Removing an AMI from a SG
  - Migrate an AMI from one SG to another

    - Changes within the AMI, e.g., new services
- Ports (SG rules)
    - Open a new port for a SG
    - Close a port for a SG
    - Change the source of a rule

We will now analyze the changes in order to determine their impact on the overall security, i.e., the risk of security breaches depending on the vulnerability ratings.

Security Group: The impact of adding or removing a security group depends on the rating of the AMIs grouped in that SG and the allowed communication for that SG. For example, a new SG containing multiple AMIs with several high severity rated services, which are all exposed to the Internet by the SG, will have a significant negative impact on the overall security of the virtual infrastructure. Removing such a SG will have of course an equivalent positive impact.

AMI: In case of configuration changes involving AMIs we can determine the following impacts. The impact of adding a new AMI depends on the severity rating of the services located in that particular AMI and the allowed communication by the SG the AMI is a member of. If the SG denies all communication to the services of the new AMI, there will be no security impact, because the AMI will be isolated. We have to assume that the AMI itself is not malicious and does not try to extract internal information by sending them outwards, because outbound filtering is not available for Amazon Security Groups. If a vulnerable services is allowed by the SG, the negative impact depends on the rating and the allowed source of the communication. For example, a medium rated services only accessible by the corporate internal network could have a lower impact than a low rated service accessible by anyone.

Removing an AMI will have an equivalent positive impact on the security of the overall virtual infrastructure. Migration of an AMI can be considered as removal and adding. Changes in an AMI are mainly concerned regarding the security of the offered services, i.e., degradation or improvement of the vulnerability ratings of the services.

Port: Finally we are considering changes in the SG rules, i.e., opening, changing, or closing ports. The impact of opening a new port depends again on the services offered by AMIs located in that particular SG. If no services on the newly opened port are offered then there will be no security impact. Otherwise the negative impact will depend on the severity rating of the opened service. Removing a port will have again an equivalent positive impact on the security. Changing the source of a rule can result in a negative or positive impact. If the old source is a subset of the new one, e.g., 1.2.3.4/24 $\subset$ 0.0.0.0/0, the result is a negative impact on the overall security, because the group of potential attackers was increased. In the other case, a reduction of the group of potential attackers will have a positive impact.

Conclusion: A negative or positive impact of the previously described changes depends in most cases on the severity rating of services, which are reflected in the attack graphs. For example, in case a new AMI is added with a service exposed by its security group, a new edge and vertex will appear in the corresponding attack graph. Therefore attack graphs are suitable for detecting changes related to vulnerability ratings.

Detecting Changes using Attack Graphs

As previously demonstrated, attack graphs can be used to represent the configuration of the virtual infrastructure when analyzing changes with regard to vulnerabilities. In case a change will have an impact on the overall security of the deployed application, the change will be reflected as a new vertex or edge in the attack graph. Since all vertices are uniquely labeled and edges are uniquely identifiable, we can easily detect differences between two given attack graphs by performing a set difference operation on the sets of vertices and edges, instead of using a costly graph isomorphism and difference technique.

Consider the attack graphs for the two configurations C_1 and C_2: $G_{C_1} = (V_{C_1}, E_{C_1})$ and $G_{C_2} = (V_{C_2}, E_{C_2})$. The set of new edges appearing in the second configuration, e.g., due to opened ports, can be determined by $E_{new} = E_{C_2} \setminus E_{C_1}$ and analogously the set of deleted edges by $E_{del} = E_{C_1} \setminus E_{C_2}$. New or deleted vertices can be determined similarly: $V_{new} = V_{C_2} \setminus V_{C_1}$ and $V_{del} = V_{C_1} \setminus V_{C_2}$, but we are mainly interested in the edges as they represent possible attack paths.

6.3.2 Evaluation of Changes

We are now presenting the method for evaluating the set of changed vertices and edges in the attack graphs, and determining the security impact of these changes. Unlike previous work in the field of ranking attack graphs [MBZ+06], we are not evaluating the individual attack graphs for two separate configurations and compare the quantified results, but rather evaluating the differences between two attack graphs and propose a quantified result based on these relative changes.

We are using the scenario presented in Section 6.1 as an example to demonstrate the evaluation. The attack graph shown in Figure 6.3 acts as the representation of configuration C_1. We are now considering changes in *AMI 1* and security group *web* resulting that a medium-rated service on port 8080 is accessible from anywhere. The difference is illustrated in Figure 6.4 which lead to the following set of changed edges: $E_{new} = \{(0.0.0.0/0, AMI1, 8080, tcp, medium)\}$ and $E_{del} = \emptyset$.

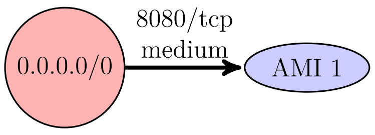


Figure 6.4: Configuration Change between Attack Graphs

In general we will have a set of edges added or removed between two configurations in the generic form illustrated in Figure 6.5. s can either be an IP range or another AMI, a is always an AMI in our attack graphs, and the edge is annotated with port p, protocol p', and vulnerability rating v.

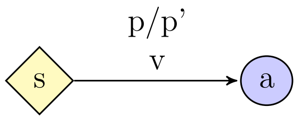


Figure 6.5: Generic Configuration Change

We will evaluate the impact of the individual components of such a change and then combine them to determine the impact of the overall change between two configurations. The evaluation is performed on an abstract conceptual level, because the choice of concrete probabilities and numbers will vary significantly between different scenarios.

Evaluate Source

In order to evaluate the impact of the source of a change, we have to classify the source into three possible cases with different attack probabilities:

- arbitrary IP range
- known IP range (e.g. a corporate network)

- another AMI

For the three different classes we assume three evaluation functions $f(ip)$, $g(ip)$, and $h(ami)$, which return a negative impact for the given input parameter. An intuitive relation between the output results of these functions is $f > g > h$, that is, the highest negative impact is given for an arbitrary IP range, because in this case we have no information on restrictions of such an attacker and have to assume the worst-case scenario, i.e., the most powerful attacker. For the other two case, an attacker must either be an insider (e.g. in the corporate network) and/or has to have compromised a specific AMI in the deployed multi-tier application.

The negative impact result of these functions can vary depending on the input parameter. In case of $f(ip)$, the negative impact depends on the "wideness" of the given IP range. For example, *1.2.3.4/24* would result in a lower negative impact compared to *0.0.0.0/0*, because the set of potential attackers is smaller in the first case. Of course, this is a probabilistic approach, because a very determined attacker could reside in a small specified IP range compared to a number of regular users in a large IP range.

In order to determine the negative impact of $g(ip)$, we require a specification of known IP ranges with a rating of their attack probabilities. For example, the corporate network should have a lower probability assigned than a third-party vendor network. However, overall the negative impact should be smaller compared to f, because the attack probabilities are semi-deterministic and we do not have to assume the most powerful attacker.

Finally we consider the function $h(ami)$ which returns a negative impact for a given AMI. Since the AMI is part of the deployed multi-tier application, an attacker has to be either an insider or one who has compromised the specific AMI. We can either specify an attack probability for the insider attacker case or base the attack probability on the fact how likely the AMI will be compromised. In the latter case, the number of incoming edges and their vulnerability rating can be an indicator for the probability that the AMI might get compromised.

Evaluate Service Vulnerability

Determining the negative impact on the security of the system for the service vulnerability is straight forward. A *high* vulnerability rating will have a greater negative impact compared to a *low* rating. In the current form we use a coarse-grained rating system consisting of high, medium, and low, but a fine-grained and numerical rating system could be adopted. For example, the *Common Vulnerability Scoring System* (CVSS) [MSR07] could be used.

Evaluate AMI

The negative impact on the security for an AMI depends on the criticality of the AMI. The criticality can either determine the impact in terms of loss, e.g., of financial nature or loss of reputation, in case the AMI is getting compromised, or the usefulness of the AMI for an attack to compromise other AMIs reachable from that AMI. For the first case, a risk assessment has to be done to determine the criticality of the deployed AMIs, which will be used for the evaluation process. The usefulness can be determined by the number of outgoing edges and their vulnerability ratings.

Overall Evaluation

In order to perform an overall evaluation of the changes between two configurations, we first combine the evaluation of the components of the changed edges and then combine the results for all edges. The combination of the edge component evaluation is given:

$$i(e) = \alpha i'(e.source) + \beta i'(e.vuln) + \gamma i'(e.ami)$$

i' is an evaluation function for individual components of edge e, based on the concepts presented before, returning a negative value, i.e., a negative security impact. i combines the results using weight factors α, β, and γ, for the impacts of the components.

We combine the evaluation of the new and removed edges using:

$$I = \sum_{e \in E_{new}} i(e) - \sum_{e \in E_{del}} i(e)$$

This will return either a positive or negative impact for the changes performed between two given configurations.

6.3.3 Application

We are now presenting a possible application for the configuration change evaluation method presented before. The application monitors relative changes in the security of a deployed application, i.e., the risk that security breaches occur, and plots the security level in respect to a time line. This can also be used to compare a new configuration with a desired or base configuration.

The construction of attack graphs can be performed periodically for a deployed multi-tier application and the newly obtained attack graph will be compared with the one from the previous periodic run. The impact on the security of the application can be calculated for the two attack graphs or configurations, assuming we have suitable parameters for the actual attack probabilities, and the relative change can be plotted in a graph.

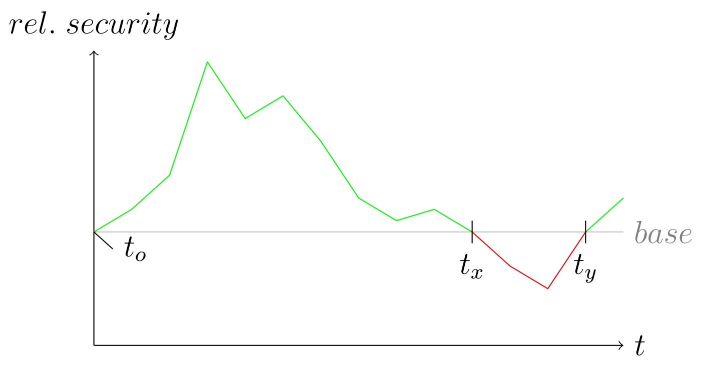


Figure 6.6: Security Level Monitoring

Figure 6.6 illustrates such a graph with sample values. At time t_o we obtain the attack graph for the initial configuration called the *base* configuration. Desirably, the security of the deployed application will only improve in the future in relation to this base configuration. As we can see in the plot, the configuration evolves positively in comparison to the base configuration, although it also degrades locally, until time t_x where configuration changes lead to a degraded security state compared to the base configuration. An administrator should be alarmed at this point to restore a proper configuration to obtain a security level at least as good as the one of the base configuration, which happens after time t_y.

6.3.4 Beyond a Vulnerability Perspective

After evaluating configuration changes with regard to vulnerability ratings of services, we will now give an outlook for the evaluation regarding replication and availability.

Comparing configuration changes by the means of attack graphs is no longer suitable for an evaluation perspective regarding replication. Important aspects of the configuration are now:

- Dependencies of AMIs
- Number of instances of an AMI
- Replication of services
- Diversity (OS, physical location)

These aspects of the configuration could be captured in a directed graph, where the vertices are AMIs again and the edges denote that a service is replicated by another AMI or that a dependency between the two AMIs exist. Each vertex could have a rating assigned based on the number of instances for that particular AMI and attributes about the operating system and physical location.

A high amount of dependency edges has a negative impact on the overall availability of the deployed virtual infrastructure, because the likelihood of cascading failures increases. Furthermore, a low number of instances for a particular AMI has also a negative impact due to possible failures of instances breaking replication or dependency associations. On the other hand, a high diversity in the infrastructure, e.g., different operating systems in different physical locations, has a positive impact on the availability level of the infrastructure. Replication among the services will also have a significant positive impact.

Similarly to the evaluation of changed edges between two attack graphs, we can apply the same approach to the analysis of changed edged between two graphs of the previously described form. Additionally, the analysis of vertices properties is required to evaluate AMI diversity and the number of instances.

Chapter 7

Evaluation & Practical Results

In this chapter we will evaluate the prototype implementation for analyzing multi-tier application deployments on Amazon with regard to performance and functionality. We will present practical results for analyzing an example application deployed on Amazon.

7.1 Performance Evaluation

In this section we will present the complexity of the analysis algorithms in general, and discuss hypothetical and practically realistic complexities of the reachability and attack graphs. Furthermore, we will show time measurements of using the SAVEly tool in practice.

7.1.1 Algorithm Complexity Analysis

Determining the complexities of the algorithms presented in Section 6.1 with regard to the input graph parameters is straight-forward. For the reachability graph algorithms Algorithm 1 and Algorithm 2, it is obvious that they iterate over the set of edges of the input graph. For the query algorithm we obtain a complexity of $O(|E|)$, and for the policy algorithm $O(|E| \cdot n)$ where n is the number of ports specified in the policy, because each edge has to be tested for all specified ports. However, typically $n \ll |E|$ which leads to $O(|E|)$ complexity for both algorithms.

The attack graph analysis algorithms are based on variations of Dijkstra's algorithm for single-source and all-pairs shortest paths. According to [CSRL01], we obtain a complexity for single-source algorithm using an array as underlying data structure of $O(|V|^2 + |E|)$, which is in many cases dominated by $|V|^2$ therefore $O(|V|^2)$. For sparse graphs, we can use a Fibonacci heap as the data structure and obtain a complexity of $O(|E| + |V|log|V|)$. Using Dijkstra's algorithm for all-pairs shortest path, we have to perform the single-source Dijkstra's algorithm for each vertex in the graph, yielding a complexity of $O(|V|^3 + |V||E|)$ for the array-based variant, which is in many cases dominated by the $|V|^3$ therefore $O(|V|^3)$. For a sparse graph with Dijkstra's algorithm based on a Fibonacci heap starting from all vertices, we obtain $O(|V|^2 log|V| + |V||E|)$.

7.1.2 Reachability Graph Upper Bound Complexity

In order to evaluate the scalability of the reachability analysis algorithms, we try to define an upper bound for the complexity a reachability graph. We consider a complete multi-graph of Security Groups, meaning each SG is connected to each other SG, itself, and to all IP sources using all possible port/protocol and ICMP combinations. Figure 7.1 illustrates such a complete reachability graph, where the dotted edges represent a set of edges for all possible port/protocol combinations.

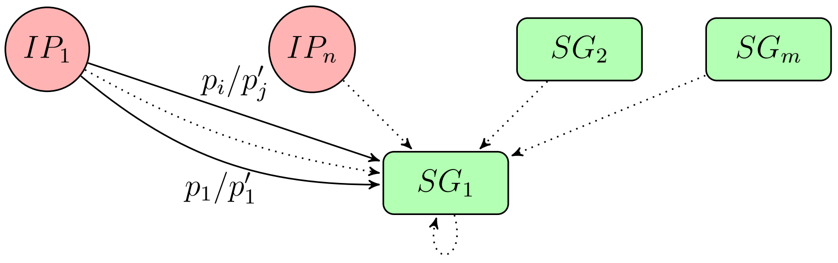


Figure 7.1: Security Groups Complete Graph

In order to calculate the number of edges and vertices in such a graph, a number of parameters and assumptions are required. The following parameters are given:

- Number of Security Groups: 100
- Port Protocol Combinations: $2 \cdot 65536$
- ICMP Combinations: $256 \cdot 14$

The maximum number of Security Groups is specified by Amazon in [Ama09a]. However, empirical testing showed that this limit is not enforced, but we will assume it as an upper bound which might be enforced in the future. The number of port protocol combinations is given by 2 possible protocols, i.e., TCP and UDP, where each protocol allows up to 65536 possible ports. The number of possible ICMP combinations is given by the ICMP type field which allows up to 256 different values (we also consider reserved values), and to each type a number of possible codes are associated where the maximum number occurs for the ICMP type 3 with 14 possible values. We assume here that the number of rules per SG is not limited, therefore allow all possible combinations of port/protocol rules within a SG, otherwise the number of edges would be less than the upper bound we are defining.

The number of edges between a source (IP source or another SG) and a SG is $|E'| = 2 \cdot 65536 + (256 \cdot 14)$ resulting from all possible port/protocol combinations. Since all IP sources and SGs are connected with all SGs using directed edges and allow loops, we obtain a total number of edges $|E| = |SG| \cdot (|SG| + |IP|) \cdot |E'|$. For the number of vertices we obtain $|V| = |SG| + |IP|$.

In case we are assume that all possible IPv4 addresses, i.e., 2^{32}, are used as IP sources, the number of vertices and edges increases dramatically. The number of edges is in the order of $5.8e18$ and the number of edges approximately 2^{32}. On a 1GHz computer, i.e., $1ns = 10^{-9}s$ per operation, the reachability algorithms would take approximately 185 years, which is clearly not feasible (at least on a single computer).

We will now consider more realistic scenarios to demonstrate the run-time of the algorithms, although they do not represent the hypothetical worst-case scenario demonstrated above. If we consider $|IP| = 1e3$, the run-time is approximately $15s$ which can be easily done. Even if we consider $|IP| = 1e6$, the duration increases to approximately $3.74hrs$ which still is realistic to achieve.

In conclusion we can say that the hypothetical worst-case scenario is not feasible to perform on a single computer, mainly due to the number of possible IP addresses. However, we showed that even for realistic complex scenarios the algorithms perform in reasonable time, which follows that even very large real-world deployment can be analyzed with the proposed algorithms in short time.

7.1.3 Attack Graph Upper Bound Complexity

Defining an upper bound complexity for the attack graphs in the analysis is similar to the previous reachability graph complexity approach. Figure 7.2 illustrates a complete attack graph.

Instead of considering security groups for the vertices, we are now considering AMIs. The number of allowed AMIs per SG is important for the complexity calculation, but no official limit exists. We will assume that a

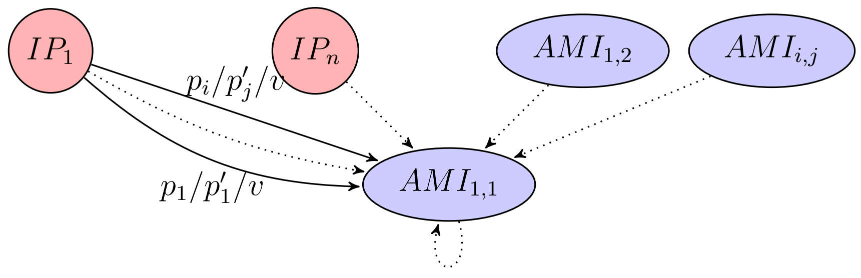


Figure 7.2: Complete Attack Graph

maximum of 100 AMIs per SG is allowed, similar to the limit of the overall number of SGs. Therefore, $|AMI| = |SG| \cdot 100$. The number of edges can then be determined by $|E| = |AMI| \cdot (|AMI| + |IP|) \cdot |E'|$, which is equivalent to the number of edges in the complete reachability graph. The number of vertices is $|V| = |AMI| + |IP|$.

Since we are dealing with a complete multi-graph, the following holds: $|V| \lll |E|$, therefore the complexity of the Dijkstra's algorithm is dominated by $|E|$. Furthermore, the number of vertices and edges is even higher than in the reachability graph example, therefore the hypothetical worst-case scenario assuming 2^{32} IP sources is again not feasible.

We are considering 100 AMIs per SG, therefore we can calculate the factor with which the number of edges increases in the complete attack graph compared to the reachability graph: $d(|IP|) = \frac{10^4+|IP|}{1+|IP|\cdot 10^{-2}}$ and therefore $|E|_{AG} = d(|IP|) \cdot |E|_{RG}$.

For the scenario $|IP| = 1e3$, we are now getting a factor of 1000, therefore the run-time is $15000s$, i.e., approximately $4.1hr$, which is a significant decrease in terms of practicality of the algorithms. In order to find a practical but still highly complex scenario, which would act as a upper bound for real-world applications, we have to consider the edges between sources and AMIs. Currently $|E'|$ represents all possible combinations of port and protocol, but this means that each AMI has a number of vulnerable services in the order of $130 \cdot 10^3$, which is very unlikely. We now consider $|E'| = 1e3$ and $|IP| = 1e3$, and we obtain a running time in the order of $1.8min$, which is very reasonable. For $|E'| = 1e4$ the run-time is still a practically reasonable $18.3min$.

Due to the increased number of vertices and especially their interconnects, the complexity of the complete attack graph is much higher than compared to a reachability graph, resulting in longer run-times of the algorithms. The hypothetical worst-case scenario is again not feasible, but considering highly complex practical scenarios, which act as a lower upper bound than the hypothetical one, shows reasonable run-times.

7.1.4 Time Measurements

Besides discussing the algorithmic complexity and upper bounds of the graph complexity, we will give time measurements for the analysis on example deployments on Amazon using the SAVEly tool. The accuracy of the time measurements is not very important, because we mainly want to convey a general overview of the time consumptions rather than using the measurements for performance improvements and profiling. The measurements were obtained on a regular laptop machine with a 2.13GHz Pentium M processor, 2 GB of RAM, and running Gentoo Linux.

Reachability Graph Analysis

The deployed configuration results in a reachability graph with *257 vertices* (resulting from the security groups) and *505 edges* (based on the security groups rules). We obtained the following measurements for the individual parts of the analysis:

- Obtain SGs from Amazon: $1.7s$
- Build reachability graph: $0.04s$
- Perform reachability queries (7): $0.025s$
- Perform policy check (2 never, 1 only): $0.01s$

Clearly obtaining the configuration from Amazon is the slowest part in the process, because a cryptographically signed transaction over the Internet to Amazon's API server has to be performed.

Attack Graph Analysis

For the construction of the attack graph we use the OpenVAS [Ope10] vulnerability scanner with 585 plugins enabled. Furthermore, we have to obtain the security groups and running instances from Amazon. We obtain the following measurements with one running instance found and scanned:

- Get Security Groups: $1.6s$
- Get Instances: $0.15s$
- Build Attack Graph: $2min52s$

The construction of the attack graph is by far the most time consuming part in the analysis process, because it involves starting a new instance at Amazon EC2 and performing a vulnerability scan, where the vulnerability scan is the dominating factor.

Since the constructed attack graph is rather simple, because it only involves one AMI, we are performing the analysis on an attack graph similar to the one presented in Section 6.1 in Figure 6.3. However, the ICMP edges were removed. We obtain the following time measurements for performing queries and policy checks on the attack graph:

- Queries (5): $0.01s$
- Policies (1 never, 1 only): $0.0015s$

Evidently for such simple attack graphs the query and policy checks can be performed in an instant.

The most time consuming part in the attack graph analysis is the vulnerability scan. A possible improvement for this step could be to parallelize it, i.e., start all instances at the same time and perform the vulnerability scan from an equal number of scanning instances. However the scanning instances can not be run on Amazon EC2, because their policy prohibits the port scan of other instances.

7.2 Reachability Analysis Results

We will now present practical results from the reachability analysis. We consider a security group configuration for a multi-tier web application, which we already presented in Chapter 6.

7.2.1 Graph Construction

Figure 7.3 illustrates the automatically constructed reachability graph using the security group configuration obtained from Amazon. The visualization is done by the tool using the *DOT* language and *graphviz* [gra10].

Figure 7.4 extends the reachability graph with membership information of AMIs, i.e., which security group contains instances of specific AMIs. For demonstration purposes we only consider one web server image located in the *default* and *web* security groups.

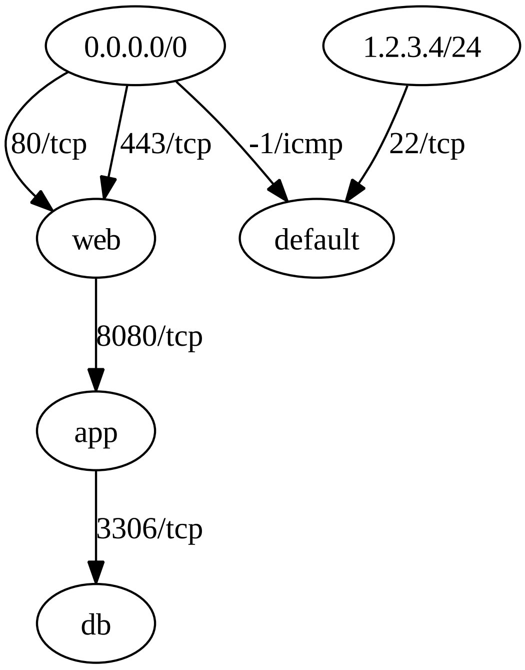


Figure 7.3: SAVEly Reachability Graph

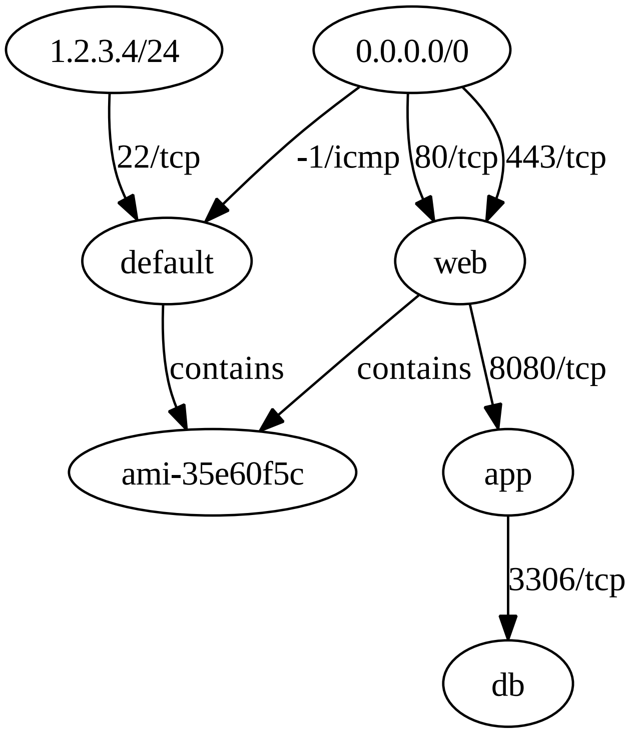


Figure 7.4: SAVEly Security Group AMI Relationship Graph

7.2.2 Analysis

We want to analyze the reachability graph of the discovered multi-tier application configuration. In the first part we get insights into the reachability by performing queries and to verify that the desired reachability is fulfilled by the current configuration. In the second part we specify undesired behavior. The file containing the reachability queries is given in Listing 7.1. It is naturally derived from the specification of the multi-tier application.

```
from 0.0.0.0/0 to web port 80 proto tcp
from 0.0.0.0/0 to web port 443 proto tcp
from web to app port 8080 proto tcp
from app to db port 3306 proto tcp
from 0.0.0.0/0 to default port −1 proto icmp
from 1.2.3.4/24 to default port 22 proto tcp
```

Listing 7.1: Reachability Query File

The software will check for each query if the specified reachability is fulfilled in the current configuration. The output for the previously shown query file is given in Listing 7.2.

```
from 0.0.0.0/0 to web port 80 proto tcp
=> True
from 0.0.0.0/0 to web port 443 proto tcp
=> True
from web to app port 8080 proto tcp
=> True
from app to db port 3306 proto tcp
=> True
from 0.0.0.0/0 to default port −1 proto icmp
=> True
from 1.2.3.4/24 to default port 22 proto tcp
=> True
```

Listing 7.2: Reachability Query Output

In the second part we now specify the undesired reachability behavior and verify if the current configuration allows such undesired behavior. The policy file is shown in Listing 7.3 and implements the idea that except from the *web* tier all other tiers should not be directly reachable from the Internet and that the *web* can only be reached on the two common web server ports.

In this case the current configuration is compliant with the policy and the software will indicate that by printing `Reachability Policy valid: True`.

```
never from 0.0.0.0/0 to app port any proto any
never from 0.0.0.0/0 to db port any proto any
only from 0.0.0.0/0 to web port 80 proto tcp and port 443 proto↩
    tcp
```

Listing 7.3: Reachability Policy File

7.3 Attack Graph Analysis Results

The more interesting security analysis is based on attack graphs which we will demonstrate in this section.

7.3.1 Graph Construction

Since we only use one web server image for the demonstration, the resulting attack graph is very simple. The web server contained in the image has a medium rated vulnerability which can be potentially exploited by any attacker. The resulting attack graph is illustrated in Figure 7.5.

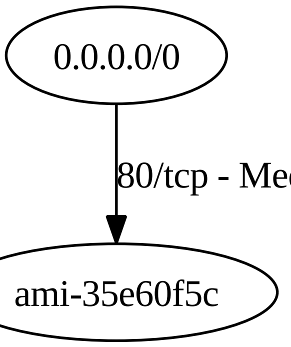


Figure 7.5: SAVEly Attack Graph

7.3.2 Analysis

Due to the simplicity of the resulting attack graph of the demonstration case, we consider the more complex attack graph shown in Chapter 6 in Figure 6.3 to demonstrate the analysis. In the first part we demonstrate the query language which supports an administrator to get an insight into the vulnerability of services in the current configuration. The second part deals with the policy language for specifying undesired properties of the configuration in terms of service vulnerabilities and attack paths.

A sample query file for the attack graph analysis is given in Listing 7.4. The queries test the service vulnerability exposed to the Internet and the corporate network, and the general vulnerability within the whole deployment.

```
from 0.0.0.0/0 to any vuln medium
from 0.0.0.0/0 to any vuln high
from 1.2.3.4/24 to any vuln medium
from any to any vuln high
from any to AMI2 vuln medium
```

Listing 7.4: Attack Query File

The output of the query analysis is given in Listing 7.5. For each query the tool also provides the most likely attack paths which fulfill the query. For example the first query, which tests which resources could be compromised by any attackers exploiting medium rated vulnerabilities, provides us with three potential paths resulting in the compromise of AMI1, AMI2, and AMI4.

```
from 0.0.0.0/0 to any vuln medium
=> [['0.0.0.0/0', 'AMI1', 'AMI2', 'AMI4'], ['0.0.0.0/0', 'AMI1↩
    '], ['0.0.0.0/0', 'AMI1', 'AMI2']]
True
from 0.0.0.0/0 to any vuln high
=> []
False
from 1.2.3.4/24 to any vuln medium
=> [['1.2.3.4/24', 'AMI2', 'AMI4'], ['1.2.3.4/24', 'AMI2'], ↩
    ['0.0.0.0/0', 'AMI1', 'AMI2', 'AMI4'], ['0.0.0.0/0', 'AMI1↩
    '], ['0.0.0.0/0', 'AMI1', 'AMI2']]
True
from any to any vuln high
=> [['1.2.3.4/24', 'AMI2']]
True
from any to AMI2 vuln medium
=> [['1.2.3.4/24', 'AMI2'], ['0.0.0.0/0', 'AMI1', 'AMI2'], ['↩
    AMI1', 'AMI2']]
True
```

Listing 7.5: Attack Query Output

Now we demonstrate the ability of the tool to verify policies, i.e., to check for undesired behavior. A simple policy file is shown in Listing 7.6 which implements the idea that an administrator wants to be sure that no high rated vulnerabilities exists in the deployed multi-tier application and that at maximum low rated vulnerabilities are exposed to any attacker.

```
never from any to any vuln high
only from 0.0.0.0/0 to any vuln low
```

Listing 7.6: Attack Policy File

The output of the policy analysis is given in Listing 7.7. Reconsidering the query output previously shown, it is not surprising that both policies are violated by the current configuration. The second last query returned a potential attack paths using a high rated vulnerability between `1.2.3.4/24` and `AMI2`, which can be manually verified using the illustration of the attack graph in Figure 6.3. The second policy is violated because several attack paths for any attacker exist using medium rated vulnerabilities as shown using the first query.

```
Attack Policy valid:
 policy ('any', 'any', 'high') violation: [['1.2.3.4/24', 'AMI2↩
    ']]
 policy ('0.0.0.0/0', 'any', 'medium') violation: ↩
    [['0.0.0.0/0', 'AMI1', 'AMI2', 'AMI4'], ['0.0.0.0/0', '↩
    AMI1'], ['0.0.0.0/0', 'AMI1', 'AMI2']]
False
```

Listing 7.7: Attack Policy Output

The policy violation detection with given counter-examples will be very useful for an administrator to fix discovered vulnerabilities in the deployed complex multi-tier application.

Chapter 8

Outlook & Open Questions

In this chapter we will give an outlook of possible future work, and we will present remaining open questions for future research directions.

8.1 Hybrid Cloud Analysis

The current state of the SAVE prototype covers the discovery and analysis of private clouds, and this thesis extended the prototype for the discovery of public clouds. In case an organization operates a private cloud and also leverages resources provided by a public cloud provider, the current SAVE prototype will handle these two clouds as individual entities rather than having an integrated view on this hybrid cloud setup.

Future work would consist of extending the SAVE prototype in a way to handle such hybrid clouds. Amazon Virtual Private Cloud (VPC), which was already discussed in Section 2.3, could be used as a specific example scenario for an initial implementation. The basic idea is to discover VPN endpoints in the private cloud and match them with the VPN configuration deployed on Amazon, therefore "stitching" together both clouds by the concept of a VPN tunnel.

Having this integrated view opens up further analysis possibilities, e.g., what resources of the public cloud are used by applications in the organization's own private cloud and how information are flowing from the private to the public cloud. A possible scenario would be the following one: A file server in the private cloud of an enterprise is becoming low on storage capacity and leverages additional storage resources from a public cloud provider. Since the file server abstracts the usage of public cloud resources, confidential or sensitive information might be stored on this external storage resources.

8.2 Amazon Security Group Transformation

In Section 6.1 we presented a method to transform Amazon Security Group configuration based on insights obtained in the analysis of reachability and attack graphs. We identified two approaches for future work in order to improve and increase the accuracy of the transformation method.

Besides splitting security groups, it would also be interesting to split AMIs. In case an AMI contains multiple services with different severity ratings, the high severity services should be isolated from the other ones. This splitting of an AMI is more difficult to automate, since we have to understand the configuration of the service we want to isolate, in order to move it to a separate AMI. The open question is how to safely extract services from an operating system image without breaking the functionality of the service and potential dependencies of services hosted on the same image.

Another improvement for increasing the accuracy of the transformation and automating the whole process is the automated discovery of service dependencies. For example, if we could discovery that two AMIs communicate using a certain protocol and port in one direction, we could place these AMIs in different security groups and only specify for one group to allow that specific port and protocol.

8.3 Design Tool

The current main functionality of the SAVE and SAVEly prototypes is to audit and visualize the current configuration of virtualized environments and multi-tier application deployments respectively. In order to further support administrators of such environments and deployments regarding security decisions, a design tool providing immediate feedback on possible changes would be desired. For example, an administrator could move VMs/AMIs from one security group to another and the tool would provide immediate feedback on reachability and service vulnerability issues. In case the change does not have any undesired side effects, it can be deployed by the administrator in an easy way. In general we are interested in What-If analysis for changes done in virtualized environments or multi-tier application deployments.

8.4 Complex Firewall Rules Analysis

Amazon opted for a simple firewall scheme implemented in their security groups suitable for the majority of customers but limited in its functionality. Outbound filtering can not be achieved, changing the default firewall behavior to accept and explicitly specify deny rules is not possible, and further firewall capabilities like stateful rules and handling protocols besides TCP, UDP, and ICMP, are not implemented.

In case that Amazon will introduce more advanced firewall features to cope with enterprise demands, the complexity of the firewall rules used by security groups will increase and hence requires more sophisticated analysis approaches. A wide body of research in complex firewall rules analysis, e.g., [BMNW04, MWZ00, YMS$^+$06, MWZ00], could be applied to this new scenario.

8.5 SAVEly for Private Clouds

Another open question is how the analysis techniques presented for the Amazon cloud can be transferred to private clouds where information about the configuration are harder to extract. For example, can we use the concept of security groups also in the private cloud scenario, i.e., a set of VMs hosted on a physical server which is protected by a firewall can be in a "security group" depending on the firewall configuration. Understanding the relationship between the firewall rules and VMs is crucial for extracting security groups from a private cloud.

Furthermore, spawning instances of production VMs for security analysis purposes, in order that the original VM is not affected by the analysis, can be more difficult in private cloud environments where the management infrastructure could be less automated, not ready for programmatic use, and more heterogeneous.

Chapter 9

Conclusion

In this thesis we presented a novel approach of assessing the security of multi-tier applications deployed in infrastructure clouds using Amazon EC2 as an example case. The security assessment consists of a discovery step for obtaining the current configuration of the deployment, which is transformed into a generic data model capable of handling heterogeneous virtual environments. Based on the obtained configuration, we are able to analyze the system with regard to two properties: reachability and services vulnerabilities. For both cases, a query language and processor allow administrators to get an insight into the reachability and vulnerability of the services, e.g., in order to detect misconfigurations which expose the wrong services or to find vulnerable services. Furthermore, a policy language allows the specification of the desired state of the system which can be periodically verified. We discuss and compare two possible methods of deploying multi-tier applications in the Amazon cloud with regard to the provided isolation levels. Finally, another process of the security assessment is presented which monitors and evaluates configuration changes with regard to vulnerability impact. This allows an administrator to track the vulnerability of the deployment over time and get alarmed in case the security decreases with regard to a base configuration. We also give an outlook how the configuration changes evaluation can be extended beyond a vulnerability perspective in order to cover different aspects like expected availability and replication.

We implemented the security assessment in Python and evaluated it against a sample multi-tier application on Amazon EC2. Violations in the vulnerability policies were successfully detected, and possible attack paths were presented to the administrator in order to enable him to secure the involved services. Besides that practical evaluation, we also evaluated the theoretical complexity of the algorithms involved in the analysis process for large input configurations. We discussed hypothetical and realistic upper-bounds for the complexity of the input configurations, in order to successfully demonstrate the feasibility of the assessment even for large deployments.

Finally, we discussed open questions and provided an outlook of future research directions. For example the presented tools could be extended to a comprehensive design tool for virtual infrastructures including the visualization and analysis of the current state, deploying changes made in the tool directly for the virtual infrastructure, and also provide a framework for doing what-if analysis for trying out different configuration changes and their impact before they are deployed.

In conclusion, we can say that our security assessment will be a valuable tool for administrators of multi-tier applications deployed in infrastructure clouds like Amazon EC2. Troubleshooting using the query languages can be done and the policy language allows a specification of the desired state, in order that the deployment remains secure. The monitoring of the configuration changes regarding vulnerability impact will allow an administrator to track the security of the virtual infrastructure over its lifetime, and enables him to intervene in case the security degrades below a certain threshold. Our practical and theoretical evaluation demonstrates the feasibility of this approach even for large deployments.

Bibliography

[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, New York, NY, USA, 2007. ACM.

[AKS07] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320, New York, NY, USA, 2007. ACM.

[Ama09a] Amazon Web Services. Amazon EC2 - Network Security Concepts. Available at `http://docs.amazonwebservices.com/AWSEC2/latest/UserGuide/index.html?concepts-security.html`, last accessed June 2010, November 2009.

[Ama09b] Amazon Web Services. Amazon Web Services: Overview of Security Processes, November 2009.

[Ama10a] Amazon. The Amazon Elastic Compute Cloud (EC2). Available at `http://aws.amazon.com/ec2/`, last accessed March 2010, 2010.

[Ama10b] Amazon Web Services. Extend Your IT Infrastructure with Amazon Virtual Private Cloud, January 2010.

[Ama10c] Amazon Web Services. Java Library for Amazon EC2. Available at `http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1674`, last accessed June 2010, 2010.

[BBN$^+$09] Mihir Bellare, Zvika Brakerski, Moni Naor, Thomas Ristenpart, Gil Segev, Hovav Shacham, and Scott Yilek. Hedged public-key encryption: How to protect against bad randomness. In *ASIACRYPT*, pages 232–249, 2009.

[BCC04] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct Anonymous Attestation. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, pages 132–145, New York, NY, USA, 2004. ACM.

[BCP$^+$08] Stefan Berger, Ramón Cáceres, Dimitrios Pendarakis, Reiner Sailer, Enriquillo Valdez, Ronald Perez, Wayne Schildhauer, and Deepa Srinivasan. TVDc: Managing Security in the Trusted Virtual Datacenter. *SIGOPS Oper. Syst. Rev.*, 42(1):40–47, 2008.

[BDF$^+$03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[BMNW04] Yair Bartal, Alain Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A Novel Firewall Management Toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[bot10] boto. boto - Python interface to Amazon Web Services. Available at http://code.google.com/p/boto/, last accessed June 2010, 2010.

[BP98] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Seventh International World-Wide Web Conference (WWW 1998)*, 1998.

[CDRS07] Serdar Cabuk, Chris I. Dalton, HariGovind Ramasamy, and Matthias Schunter. Towards Automated Provisioning of Secure Virtualized Networks. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2007. ACM.

[Cis10] Cisco. Cisco Nexus 1000V Series Switches. Available at https://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9902/data_sheet_c78-492971.html, last accessed June 2010, 2010.

[Clo09] Cloud Security Alliance. Security Guidance for Critical Areas of Focus in Cloud Computing, December 2009.

[Coh10] Reuven Cohen. Announcing Enomaly ECP High Assurance Edition for Trusted Cloud Computing.

Available at http://www.elasticvapor.com/2010/04/announcing-enomaly-ecp-high-assurance.html, last accessed June 2010, 2010.

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CSS+09] Mihai Christodorescu, Reiner Sailer, Douglas Lee Schales, Daniele Sgandurra, and Diego Zamboni. Cloud Security Is Not (Just) Virtualization Security. In *CCSW '09: Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 97–102, New York, NY, USA, 2009. ACM.

[Eng08] Paul England. Practical Techniques for Operating System Attestation. In *Trust '08: Proceedings of the 1st international conference on Trusted Computing and Trust in Information Technologies*, pages 1–13, Berlin, Heidelberg, 2008. Springer-Verlag.

[ENI09] ENISA. Cloud Computing Risk Assessment. Technical report, ENISA, 2009.

[GJP+05] John Linwood Griffin, Trent Jaeger, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Ramón Cáceres. Trusted Virtual Domains: Toward secure distributed services. In *In Proc. of the First Workshop on Hot Topics in System Dependability (Hotdep05*. IEEE Press, 2005.

[Goo07] John R. Goodall. Introduction to Visualization for Computer Security. In *VizSEC*, pages 1–17, 2007.

[Goo10] Google. Google App Engine. Available at http://code.google.com/appengine/, last accessed June 2010, 2010.

[GPC+03] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A Virtual Machine-Based Platform for Trusted Computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, 2003.

[GR03] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[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.

[gra10] graphviz. Graphviz - Graph Visualization Software. Available at http://www.graphviz.org/, last accessed June 2010, 2010.

[Her10] Heroku. Heroku - Ruby Cloud Platform as a Service. Available at http://heroku.com/, last accessed June 2010, 2010.

[HL10] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, New Delhi, India, Aug 2010.

[Hoh96] Michael Hohmuth. Linux-Emulation auf einem Mikrokern. Technical report, TU Dresden, 1996.

[HUL06] Gernot Heiser, Volkmar Uhlig, and Joshua LeVasseur. Are Virtual-Machine Monitors Microkernels Done Right? *SIGOPS Oper. Syst. Rev.*, 40(1):95–99, 2006.

[HWF+05] Steven Hand, Andrew Warfield, Keir Fraser, Evangelos Kotsovinos, and Dan Magenheimer. Are Virtual Machine Monitors Microkernels Done Right? In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 1–1, Berkeley, CA, USA, 2005. USENIX Association.

[Jaq07] Andrew Jaquith. *Security Metrics: Replacing Fear, Uncertainty, and Doubt.* Addison-Wesley Professional, 2007.

[Jer09] Jericho Forum. Cloud Cube Model: Selecting Cloud Formations for Secure Collaboration, April 2009.

[JLSW09] Sushil Jajodia, Peng Liu, Vipin Swarup, and Cliff Wang. *Cyber Situational Awareness: Issues and Research*, chapter Topological Vulnerability Analysis, pages 139–154. Springer, 2009.

[JN07] Sushil Jajodia and Steven Noel. *Algorithms, Architectures, and Information Systems Security*, chapter Topological Vulnerability Analysis: A Powerful New Approach for Network Attack Prevention, Detection, and Response. World Scientific Press, 2007.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, pages 207–220, Big Sky, MT, USA, Oct 2009. ACM.

[KL09a] 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.

[KL09b] Amir R. Khakpour and Alex X. Liu. Quantifying and Verifying Network Reachability. Poster at 16th ACM Conference on Computer and Communications Security, 2009.

[Kra09] F. John Krautheim. Private Virtual Infrastructure for Cloud Computing. In *HotCloud '09: Workshop on Hot Topics in Cloud Computing*. USENIX, 2009.

[KSS+09] Sunil D. Krothapalli, Xin Sun, Yu-Wei E. Sung, Suan Aik Yeo, and Sanjay G. Rao. A toolkit for automating and visualizing vlan configuration. In *SafeConfig '09: Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*, pages 63–70, New York, NY, USA, 2009. ACM.

[Kun10] Vivek Kundra. State of Public Sector Cloud Computing, May 2010.

[LI05] R. P. Lippmann and K. W. Ingols. An Annotated Review of Past Papers on Attack Graphs, 2005.

[LSNH+09] Liang Lu, Rei Safavi-Naini, Markus Hagenbuchner, Willy Susilo, Jeffrey Horton, Sweah Liang Yong, and Ah Chung Tsoi. Ranking attack graphs with graph neural networks. In *ISPEC '09: Proceedings of the 5th International Conference on Information Security Practice and Experience*, pages 345–359, Berlin, Heidelberg, 2009. Springer-Verlag.

[MBZ+06] Vaibhav Mehta, Constantinos Bartzis, Haifeng Zhu, Edmund M. Clarke, and Jeannette M. Wing. Ranking Attack Graphs. In *RAID*, pages 127–144, 2006.

[MG09a] Peter Mell and Tim Grance. Effectively and Securely Using the Cloud Computing Paradigm, October 2009.

[MG09b] Peter Mell and Tim Grance. The NIST Definition of Cloud Computing, October 2009.

[MGHW09] Jeanna Matthews, Tal Garfinkel, Christofer Hoff, and Jeff Wheeler. Virtual Machine Contracts for Datacenter and Cloud Computing Environments. In *ACDC '09: Proceedings of the 1st workshop on Automated control for datacenters and clouds*, pages 25–30, New York, NY, USA, 2009. ACM.

[Mic10] Microsoft. Windows Azure Platform. Available at `http://www.microsoft.com/windowsazure/`, last accessed June 2010, 2010.

[MMH08] Derek Gordon Murray, Grzegorz Milos, and Steven Hand. Improving Xen Security Through Disaggregation. In *VEE '08: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 151–160, New York, NY, USA, 2008. ACM.

[MMJZ06] Wenbo Mao, Andrew Martin, Hai Jin, and Huanguo Zhang. Innovations for grid security from trusted computing. In *Fourteenth International Workshop on Security Protocols*, LNCS. Springer-Verlag, 2006.

[MSR07] Peter Mell, Karen Scarfone, and Sasha Romanosky. A Complete Guide to the Common Vulnerability Scoring System Version 2.0. Available at `http://www.first.org/cvss/cvss-guide.html`, last accessed June 2010, June 2007.

[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 Computer Society.

[NEJ$^+$09] Steven Noel, Matthew Elder, Sushil Jajodia, Pramod Kalapa, Scott O'Hare, and Kenneth Prole. Advances in Topological Vulnerability Analysis. In *CATCH '09: Proceedings of the 2009 Cybersecurity Applications & Technology Conference for Homeland Security*, pages 124–129, Washington, DC, USA, 2009. IEEE Computer Society.

[Net10] NetworkX Developers. NetworkX. Available at `http://networkx.lanl.gov/`, last accessed June 2010, 2010.

[Ols08] David P Olshefski. Amazon EC2. IBM internal presentation, 2008.

[Ope10] OpenVAS. OpenVAS, Open Vulnerability Assessment System. Available at http://www.openvas.org, last accessed May 2010, 2010.

[Orm07] Tavis Ormandy. An Empirical Study into the Security Exposure of Hosts of Hostile Virtualized Environments, 2007.

[PCL07] Bryan D. Payne, Martim Carbone, and Wenke Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, December 2007.

[Per05] Colin Percival. Cache missing for fun and profit, May 2005.

[PS98] Cynthia Phillips and Laura Painton Swiler. A graph-based system for network-vulnerability analysis. In *NSPW '98: Proceedings of the 1998 workshop on New security paradigms*, pages 71–79, New York, NY, USA, 1998. ACM.

[Qum06] Qumranet. KVM: Kernel-based Virtualization Driver, 2006.

[Rig08] RightScale. Amazon's Elastic Block Store explained. Available at http://blog.rightscale.com/2008/08/20/amazon-ebs-explained/, last accessed June 2010, August 2008.

[RTSS09] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 199–212, New York, NY, USA, 2009. ACM.

[Rus08] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, 2008.

[RW10] Joanna Rutkowska and Rafal Wojtczuk. Qubes OS Architecture, January 2010.

[RY10] Thomas Ristenpart and Scott Yilek. When Good Randomness Goes Bad: Virtual Machine Reset Vulnerabilities and Hedging Deployed Cryptography. In *Proceedings of Network and Distributed Security Symposium – NDSS '10*, 2010.

[SGR09] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards Trusted Cloud Computing. In *HotCloud '09: Workshop on Hot Topics in Cloud Computing*. USENIX, 2009.

[SHJ$^+$02] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. Automated Generation and Analysis of Attack Graphs. In *SP '02: Proceedings of the 2002 IEEE Symposium on Security and Privacy*, page 273, Washington, DC, USA, 2002. IEEE Computer Society.

[SJV$^+$05] Reiner Sailer, Trent Jaeger, Enriquillo Valdez, Ramon Caceres, Ronald Perez, Stefan Berger, John Linwood Griffin, and Leendert van Doorn. Building a mac-based security architecture for the xen open-source hypervisor. In *ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference*, pages 276–285, Washington, DC, USA, 2005. IEEE Computer Society.

[SK10] Udo Steinberg and Bernhard Kauer. NOVA: A Microhypervisor-Based Secure Virtualization Architecture. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 209–222, New York, NY, USA, 2010. ACM.

[SPEC01] L.P. Swiler, C. Phillips, D. Ellis, and S. Chakerian. Computer-attack graph generation tool. In *DARPA Information Survivability Conference Exposition II, 2001. DISCEX '01. Proceedings*, volume 2, pages 307 –321 vol.2, 2001.

[SPG97] Laura Painton Swiler, Cynthia Phillips, and Timothy Gaylor. A Graph-Based Network-Vulnerability Analysis System. In *Sandia National Laboratories, Albuquerque,New*, pages 97–3010. ACM Press, 1997.

[SVJ$^+$05] Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Van Doorn, John Linwood Griffin, Stefan Berger, Reiner Sailer, Enriquillo Valdez, Trent Jaeger, Ronald Perez, Leendert Doorn, John Linwood, and Griffin Stefan Berger. sHype: Secure Hypervisor Approach to Trusted Virtualized Systems. In *IBM Research Report RC23511*, 2005.

[SW04] Oleg Sheyner and Jeannette Wing. Tools for generating and analyzing attack graphs. In *IN PROCEEDINGS OF FORMAL METHODS FOR COMPONENTS AND OBJECTS, LECTURE NOTES IN COMPUTER SCIENCE*, pages 344–371, 2004.

[TASB07] Tung Tran, Ehab Al-Shaer, and Raouf Boutaba. PolicyVis: Firewall Security Policy Visualization and Inspection. In *LISA'07: Proceedings of the 21st conference on Large Installation System*

Administration Conference, pages 1–16, Berkeley, CA, USA, 2007. USENIX Association.

[TDSB09] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied. Crossbow: From Hardware Virtualized NICs to Virtualized Networks. In *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 53–62, New York, NY, USA, 2009. ACM.

[Ten10] Tenable Network Security. Nessus, the Network Vulnerability Scanner. Available at http://www.nessus.org, last accessed March 2010, 2010.

[VMw07] VMware. Understanding Full Virtualization, Paravirtualization, and Hardware Assist, 2007.

[VMw09] VMware. VMware ESX and VMware ESXi, 2009.

[VMw10] VMware. VMware VMsafe Security Technology. Available at http://www.vmware.com/technical-resources/security/vmsafe/security_technology.html, last accessed June 2010, 2010.

[vT10] Michael von Tessin. Towards High-Assurance Multiprocessor Virtualisation. In *Proceedings of the 6th International Verification Workshop*, Edinburgh, UK, Jul 2010.

[WIL$^+$08] Lingyu Wang, Tania Islam, Tao Long, Anoop Singhal, and Sushil Jajodia. An attack graph-based probabilistic security metric. In *Proceeedings of the 22nd annual IFIP WG 11.3 working conference on Data and Applications Security*, pages 283–296, Berlin, Heidelberg, 2008. Springer-Verlag.

[WLI07] Leevar Williams, Richard Lippmann, and Kyle Ingols. An Interactive Attack Graph Cascade and Reachability Display. In *VizSEC*, pages 221–236, 2007.

[WLI08] Leevar Williams, Richard Lippmann, and Kyle Ingols. GARNET: A Graphical Attack Graph and Reachability Network Evaluation Tool. In *VizSec '08: Proceedings of the 5th international workshop on Visualization for Computer Security*, pages 44–59, Berlin, Heidelberg, 2008. Springer-Verlag.

[Woj08] Rafal Wojtczuk. Adventures with a certain Xen vulnerability (in the PVFB backend), October 2008.

[Woo04] Avishai Wool. A Quantitative Study of Firewall Configuration Errors. *Computer*, 37(6):62–67, 2004.

[WSG+09] Timothy Wood, Prashant Shenoy, Alexandre Gerber, K.K. Ramakrishnan, and Jacobus Van der Merwe. The Case for Enterprise-Ready Virtual Private Clouds. In *HotCloud '09: Workshop on Hot Topics in Cloud Computing*. USENIX, 2009.

[WSJ07a] Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Measuring the Overall Security of Network Configurations Using Attack Graphs. In *Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security*, pages 98–112, Berlin, Heidelberg, 2007. Springer-Verlag.

[WSJ07b] Lingyu Wang, Anoop Singhal, and Sushil Jajodia. Toward Measuring Network Security Using Attack Graphs. In *QoP '07: Proceedings of the 2007 ACM workshop on Quality of protection*, pages 49–54, New York, NY, USA, 2007. ACM.

[Xen10] XenAccess. XenAccess Library. Available at `http://code.google.com/p/xenaccess/`, last accessed June 2010, 2010.

[XZM+04] Geoffrey G. Xie, Jibin Zhan, David A. Maltz, Hui Zhang, Albert Greenberg, Gisli Hjalmtysson, and Jennifer Rexford. On Static Reachability Analysis of IP Networks, 2004.

[YMS+06] Lihua Yuan, Jianning Mai, Zhendong Su, Hao Chen, Chen-Nee Chuah, and Prasant Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *SP '06: Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 199–213, Washington, DC, USA, 2006. IEEE Computer Society.

Appendix A

Amazon EC2 Architecture Details

In this appendix chapter we will provide more details about the underlying architecture of Amazon EC2. The information in this chapter were gathered from XenStore, which holds configuration information about all domains. A domain can read its own configuration information from XenStore using `xenstore-ls`, which is part of the xen utils package. The actual XenStore dump is presented in Listing A.1.

A.1 Storage

A.1.1 Instance Storage

Instance Storage appears as 3 partitions to the VM: sda1 for root, sda2 for extra storage space (/mnt), and sda3 for swap. Typically the backend of these virtual block devices are based on loopback devices and/or LVM logical volumes. Logical volumes are considered to have a better performance and reliability compared to loopback devices. Surprisingly, sda1 is using a loopback backend as noted as *node = "/dev/loop13"* in the XenStore VBD entry. The actual file used by the loopback device is *params = "/mnt/instance_image_store_3/262768"*. The suffix "_3" of the directory is probably based on the local domain number. The numerical filename of the image is related to the AMI ID, but the external AMI is in fact encoded.

The swap device is a LVM logical volume denoted as *params = "/dev/VolGroupDomU/instance_swap_store_3"* in XenStore. The extra storage space /mnt is also using a logical volume backend: *params = "/dev/mapper/cow-VolGroupDomU-instance_ephemeral_store_3"*. As indicated by *cow*, this volume is using copy-on-write functionality.

A.1.2 Elastic Block Storage

The characteristics of the Elastic Block Store (EBS) lead to the conclusion that it is a SAN-based setup, i.e., based on iSCSI volumes. In fact it is based on Global Network Block Device (GNBD) as indicated by the backend device of a EBS: *params = "/dev/gnbd89"*.

A.2 Networking

Amazon uses a routed Xen network setup with DHCP providing private IP addresses to the VMs, which can be derived from the network script name: *script = "/etc/xen/scripts/ec2-vif-route-dhcpd"* in XenStore. A traceroute will show the private IP address of the router in Dom0, as well as the external IP address of Dom0.

A.3 Domain Naming

Via XenStore one can also determine the name of a VM, which is in this example *domain = "dom_32504936"*. Assuming this suffix number of the name is incremental and unique throughout the lifetime of EC2, one could deduce usage numbers from the domain names.

A.4 XenStore Dump

```
vm = "/vm/00000000-0000-0000-0000-0ec232504936"
device = ""
 vif = ""
  0 = ""
   backend-id = "0"
   mac = "12:31:39:0B:25:77"
   handle = "0"
   state = "4"
   backend = "/local/domain/0/backend/vif/151/0"
   tx-ring-ref = "768"
   rx-ring-ref = "769"
   event-channel = "6"
   request-rx-copy = "0"
   feature-rx-notify = "1"
   feature-sg = "1"
   feature-gso-tcpv4 = "1"
 vbd = ""
  2049 = ""
   backend-id = "0"
   virtual-device = "2049"
   device-type = "disk"
   state = "4"
   backend = "/local/domain/0/backend/vbd/151/2049"
   ring-ref = "770"
   event-channel = "7"
   protocol = "x86_32-abi"
  2050 = ""
   backend-id = "0"
   virtual-device = "2050"
   device-type = "disk"
   state = "4"
   backend = "/local/domain/0/backend/vbd/151/2050"
   ring-ref = "771"
   event-channel = "8"
   protocol = "x86_32-abi"
  2051 = ""
   backend-id = "0"
   virtual-device = "2051"
   device-type = "disk"
   state = "4"
   backend = "/local/domain/0/backend/vbd/151/2051"
   ring-ref = "772"
   event-channel = "9"
   protocol = "x86_32-abi"
control = ""
error = ""
device-misc = ""
 vif = ""
  nextDeviceID = "1"
console = ""
 ring-ref = "2365586"
 port = "2"
 limit = "1048576"
 tty = "/dev/pts/4"
name = "dom_32504936"
domid = "151"
cpu = ""
 0 = ""
  availability = "online"
memory = ""
 target = "1740800"
store = ""
 ring-ref = "2365587"
 port = "1"
```

```
VBD:

domain = "dom_32504936"
frontend = "/local/domain/151/device/vbd/2050"
dev = "sda2"
state = "4"
params = "/dev/mapper/cow-VolGroupDomU-↩
    instance_ephemeral_store_3"
mode = "w"
online = "1"
frontend-id = "151"
type = "phy"
physical-device = "fd:2c"
hotplug-status = "connected"
sectors = "312705024"
info = "0"
sector-size = "512"

domain = "dom_32504936"
frontend = "/local/domain/151/device/vbd/2051"
dev = "sda3"
state = "4"
params = "/dev/VolGroupDomU/instance_swap_store_3"
mode = "w"
online = "1"
frontend-id = "151"
type = "phy"
physical-device = "fd:10"
hotplug-status = "connected"
sectors = "1835008"
info = "0"
sector-size = "512"

domain = "dom_32504936"
frontend = "/local/domain/151/device/vbd/2049"
dev = "sda1"
state = "4"
params = "/mnt/instance_image_store_3/262768"
mode = "w"
online = "1"
frontend-id = "151"
type = "file"
node = "/dev/loop13"
physical-device = "7:d"
hotplug-status = "connected"
sectors = "20971520"
info = "0"
```

```
sector-size = "512"

VIF:

domain = "dom_32504936"
handle = "0"
script = "/etc/xen/scripts/ec2-vif-route-dhcpd"
state = "4"
frontend = "/local/domain/151/device/vif/0"
mac = "12:31:39:0B:25:77"
online = "1"
frontend-id = "151"
feature-sg = "1"
feature-gso-tcpv4 = "1"
feature-rx-copy = "1"
hotplug-status = "connected"

EBS:

domain = "dom_32504936"
frontend = "/local/domain/151/device/vbd/2128"
dev = "sdf"
state = "4"
params = "/dev/gnbd89"
mode = "w"
online = "1"
frontend-id = "151"
type = "phy"
physical-device = "fc:59"
sectors = "2097152"
info = "0"
sector-size = "512"
hotplug-status = "connected"
```

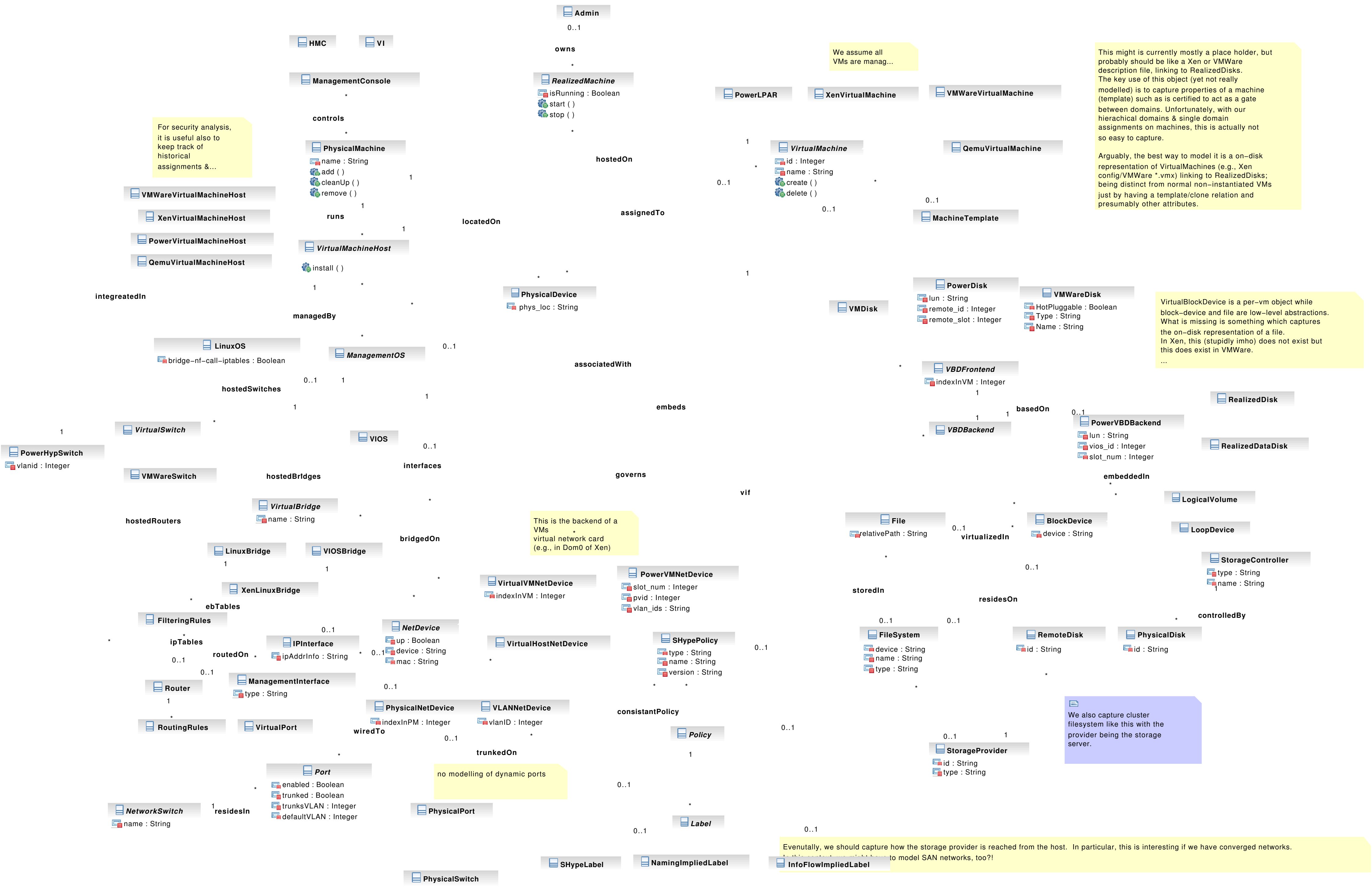
Listing A.1: Amazon EC2 XenStore Dump

Appendix B

Data Models

B.1 Realization Model

Figure B.1 on page 93.



We assume all VMs are manag...

This might is currently mostly a place holder, but probably should be like a Xen or VMWare description file, linking to RealizedDisks. The key use of this object (yet not really modelled) is to capture properties of a machine (template) such as is certified to act as a gate between domains. Unfortunately, with our hierachical domains & single domain assignments on machines, this is actually not so easy to capture.

Arguably, the best way to model it is a on-disk representation of VirtualMachines (e.g., Xen config/VMWare *.vmx) linking to RealizedDisks; being distinct from normal non-instantiated VMs just by having a template/clone relation and presumably other attributes.

For security analysis, it is useful also to keep track of historical assignments &...

VirtualBlockDevice is a per-vm object while block-device and file are low-level abstractions. What is missing is something which captures the on-disk representation of a file. In Xen, this (stupidly imho) does not exist but this does exist in VMWare.
...

This is the backend of a VMs virtual network card (e.g., in Dom0 of Xen)

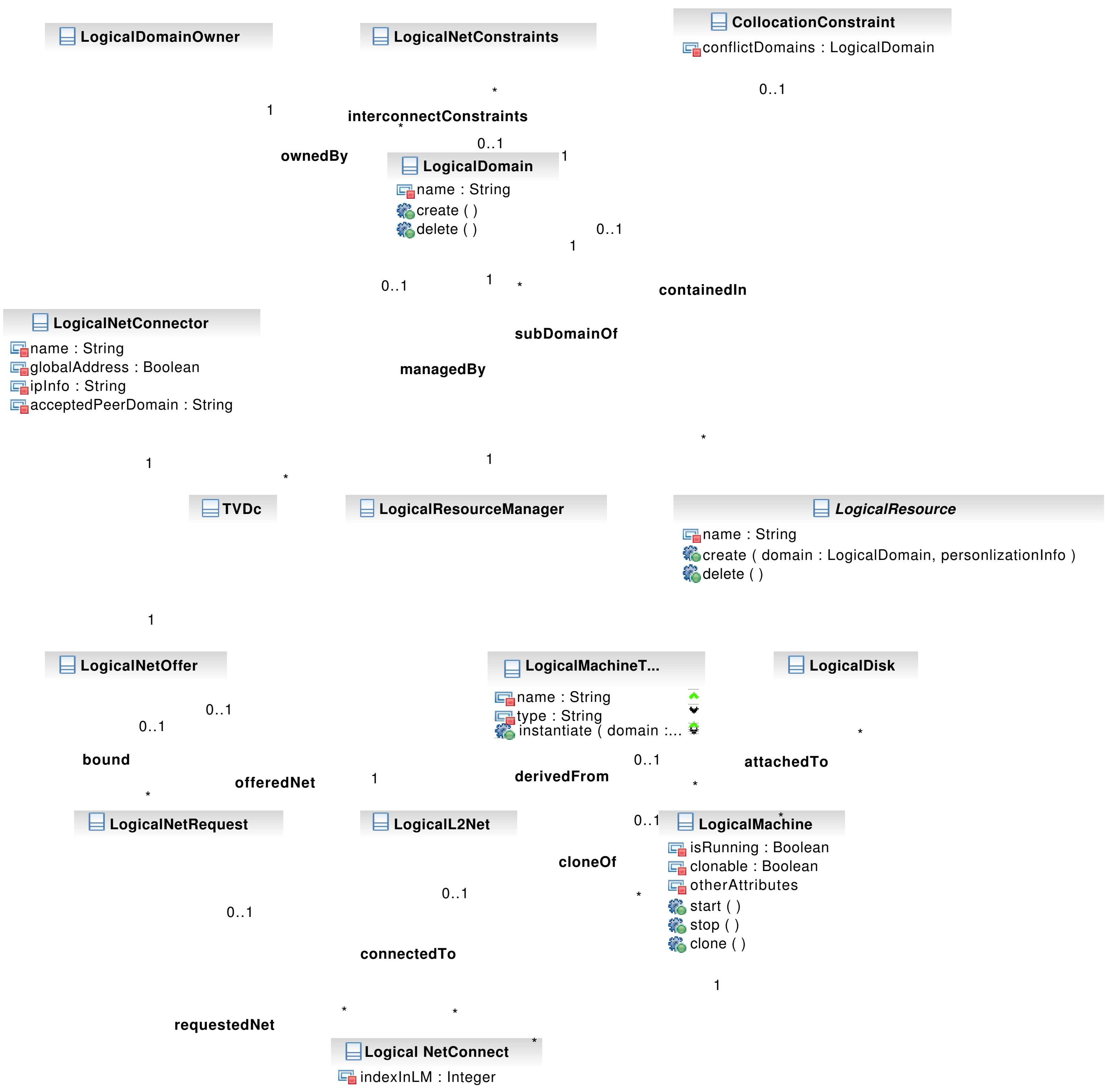
no modelling of dynamic ports

We also capture cluster filesystem like this with the provider being the storage server.

Evenutally, we should capture how the storage provider is reached from the host. In particular, this is interesting if we have converged networks. In this context, we might have to model SAN networks, too?!

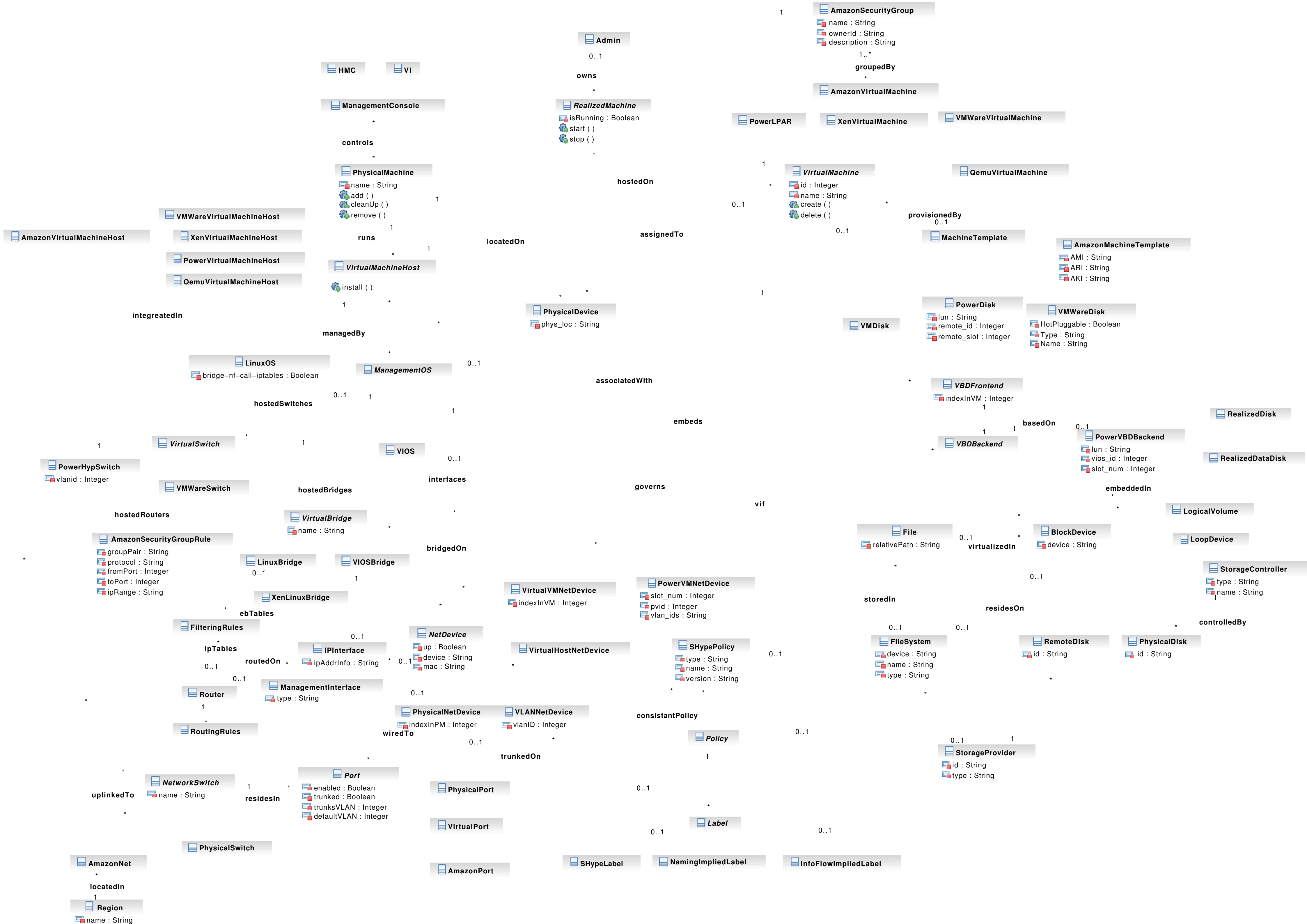
B.2 Logical Model

Figure B.2 on page 95.


LogicalDomainOwner
LogicalNetConstraints
CollocationConstraint
conflictDomains : LogicalDomain
0..1
*
1
interconnectConstraints
*
0..1
ownedBy
LogicalDomain
1
name : String
create ( )
delete ( )
0..1
1
0..1
1
*
containedIn
LogicalNetConnector
name : String
globalAddress : Boolean
ipInfo : String
acceptedPeerDomain : String
subDomainOf
managedBy
*
1
1
*
TVDc
LogicalResourceManager
LogicalResource
name : String
create (domain : LogicalDomain, personlizationInfo)
delete ( )
1
LogicalNetOffer
LogicalMachineT...
name : String
type : String
instantiate (domain :...
LogicalDisk
0..1
0..1
*
bound
0..1
attachedTo
offeredNet
1
derivedFrom
*
*
LogicalNetRequest
LogicalL2Net
0..1
LogicalMachine
*
isRunning : Boolean
clonable : Boolean
otherAttributes
start ( )
stop ( )
clone ( )
cloneOf
0..1
*
0..1
connectedTo
1
requestedNet
*
*
*
Logical NetConnect
indexInLM : Integer

B.3 Integrated Realization Model

Figure B.3 on page 97.


AmazonSecurityGroup
name : String
ownerId : String
description : String
1..*
groupedBy
AmazonVirtualMachine
Admin
0..1
owns
HMC
VI
ManagementConsole
controls
RealizedMachine
isRunning : Boolean
start ( )
stop ( )
PowerLPAR
XenVirtualMachine
VMWareVirtualMachine
PhysicalMachine
name : String
add ( )
cleanUp ( )
remove ( )
hostedOn
VirtualMachine
id : Integer
name : String
create ( )
delete ( )
QemuVirtualMachine
provisionedBy
VMWareVirtualMachineHost
AmazonVirtualMachineHost
XenVirtualMachineHost
PowerVirtualMachineHost
QemuVirtualMachineHost
runs
locatedOn
assignedTo
MachineTemplate
AmazonMachineTemplate
AMI : String
ARI : String
AKI : String
VirtualMachineHost
install ( )
integreatedIn
PhysicalDevice
phys_loc : String
PowerDisk
lun : String
remote_id : Integer
remote_slot : Integer
VMWareDisk
HotPluggable : Boolean
Type : String
Name : String
VMDisk
managedBy
LinuxOS
bridge−nf−call−iptables : Boolean
ManagementOS
associatedWith
VBDFrontend
indexInVM : Integer
hostedSwitches
embeds
basedOn
RealizedDisk
PowerVBDBackend
lun : String
vios_id : Integer
slot_num : Integer
VirtualSwitch
VIOS
VBDBackend
RealizedDataDisk
PowerHypSwitch
vlanid : Integer
VMWareSwitch
interfaces
hostedBridges
governs
embeddedIn
vif
hostedRouters
VirtualBridge
name : String
LogicalVolume
File
relativePath : String
virtualizedIn
BlockDevice
device : String
LoopDevice
AmazonSecurityGroupRule
groupPair : String
protocol : String
fromPort : Integer
toPort : Integer
ipRange : String
LinuxBridge
VIOSBridge
bridgedOn
VirtualVMNetDevice
indexInVM : Integer
PowerVMNetDevice
slot_num : Integer
pvid : Integer
vlan_ids : String
StorageController
type : String
name : String
XenLinuxBridge
ebTables
storedIn
residesOn
controlledBy
FilteringRules
NetDevice
up : Boolean
device : String
mac : String
IPInterface
ipAddrInfo : String
ipTables
VirtualHostNetDevice
SHypePolicy
type : String
name : String
version : String
FileSystem
device : String
name : String
type : String
RemoteDisk
id : String
PhysicalDisk
id : String
routedOn
ManagementInterface
type : String
Router
PhysicalNetDevice
indexInPM : Integer
VLANNetDevice
vlanID : Integer
consistantPolicy
RoutingRules
wiredTo
Policy
StorageProvider
id : String
type : String
trunkedOn
NetworkSwitch
name : String
Port
enabled : Boolean
trunked : Boolean
trunksVLAN : Integer
defaultVLAN : Integer
PhysicalPort
uplinkedTo
residesIn
VirtualPort
Label
PhysicalSwitch
AmazonNet
locatedIn
AmazonPort
SHypeLabel
NamingImpliedLabel
InfoFlowImpliedLabel
Region
name : String

Appendix C

SAVE Examples

C.1 Colored Realization Model

C.1.1 Coloring Policy

```
<AnalysisParams>
  <colorFacts>
    <EnumeratedColorFact target="itvdc.ManagementOS">
      <prefix>management</prefix>
      <count>1</count>
    </EnumeratedColorFact>
    <CompositeFact name="DevTest" explicit="false">
      <target>
        <NodeTypeMatch type="itvdc.FileSystem"/>
        <NodeAttrMatch attrName="name" matchRegExpr="↩
            salva_vmware"/>
      </target>
      <value class="FixedValue">
        <value>isolated-group</value>
      </value>
    </CompositeFact>
  </colorFacts>
  <colorRules>
    <FollowRule source="itvdc.ManagementOS" target="itvdc.↩
        VirtualMachineHost" bidir="true"/>
    <FollowRule source="itvdc.VirtualMachineHost" target="itvdc↩
        .PhysicalMachine" bidir="true"/>

    <FollowRule source="itvdc.ManagementOS" target="itvdc.↩
        Router" bidir="true"/>
    <FollowRule source="itvdc.Router" target="itvdc.IPInterface↩
        " bidir="true"/>
```

```
<FollowRule source="itvdc.NetDevice" target="itvdc.↩
    IPInterface" bidir="true"/>

<CompositeRule name="StopAtDisabledPort" action="stop">
  <source>
    <NodeTypeMatch type="itvdc.NetDevice"/>
  </source>
  <target>
    <NodeTypeMatch type="itvdc.Port"/>
    <NodeAttrMatch attrName="enabled" matchRegExpr="false"/↩
        >
  </target>
</CompositeRule>
<FollowRule source="itvdc.NetDevice" target="itvdc.Port" ↩
    bidir="true"/>

<CompositeRule name="VlanEncapDefault" action="follow">
  <source>
    <NodeTypeMatch type="itvdc.Port"/>
    <OrNodeMatch>
      <NodeAttrMatch attrName="defaultVLAN" matchRegExpr="0↩
          "/>
      <InvNodeMatch>
        <NodeAttrMatch attrName="defaultVLAN" matchRegExpr=↩
            ".*"/>
      </InvNodeMatch>
    </OrNodeMatch>
  </source>
  <target>
    <NodeTypeMatch type="itvdc.NetworkSwitch"/>
  </target>
</CompositeRule>

<CompositeRule name="VlanEncap" action="follow">
  <source>
    <NodeTypeMatch type="itvdc.Port"/>
    <NodeAttrMatch attrName="defaultVLAN" matchRegExpr=".*"↩
        />
    <InvNodeMatch>
      <NodeAttrMatch attrName="defaultVLAN" matchRegExpr="0↩
          "/>
    </InvNodeMatch>
  </source>
  <target>
    <NodeTypeMatch type="itvdc.NetworkSwitch"/>
  </target>
  <colorTransform class="ColorTagPush">
    <tagName>vlan</tagName>
    <tagValue class="SourceNodeAttr">
```

```
          <attrName>defaultVLAN</attrName>
        </tagValue>
      </colorTransform>
    </CompositeRule>

    <CompositeRule name="VlanDecap" action="follow">
      <source>
        <NodeTypeMatch type="itvdc.NetworkSwitch"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.Port"/>
        <NodeAttrMatch attrName="defaultVLAN" matchRegExpr=".*"↩
            />
      </target>
      <color>
        <ColorTagMatch name="vlan">
          <value class="TargetNodeAttr">
            <attrName>defaultVLAN</attrName>
          </value>
        </ColorTagMatch>
      </color>
      <colorTransform class="ColorTagPop"/>
    </CompositeRule>

    <CompositeRule name="VlanDecapDefault" action="follow">
      <source>
        <NodeTypeMatch type="itvdc.NetworkSwitch"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.Port"/>
        <OrNodeMatch>
          <NodeAttrMatch attrName="defaultVLAN" matchRegExpr="0↩
              "/>
          <InvNodeMatch>
            <NodeAttrMatch attrName="defaultVLAN" matchRegExpr=↩
                ".*"/>
          </InvNodeMatch>
        </OrNodeMatch>
      </target>
      <color>
        <InvColorMatch>
          <ColorTagMatch name="vlan"/>
        </InvColorMatch>
      </color>
    </CompositeRule>

    <CompositeRule name="VlanDecapTrunked" action="follow">
      <source>
        <NodeTypeMatch type="itvdc.NetworkSwitch"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.Port"/>
        <NodeAttrMatch attrName="trunked" matchRegExpr="true"/>
      </target>
    </CompositeRule>

    <FollowRule source="itvdc.VirtualMachine" target="itvdc.↩
       VirtualVMNetDevice" bidir="true"/>

    <CompositeRule name="IgnoreHotpluggable" action="stop">
      <target>
        <NodeTypeMatch type="itvdc.VMWareDisk"/>
        <NodeAttrMatch attrName="hotPluggable" matchRegExpr="↩
            true"/>
      </target>
    </CompositeRule>

    <FollowRule source="itvdc.VirtualMachine" target="itvdc.↩
       VBDFrontend" bidir="true"/>
    <FollowRule source="itvdc.VBDFrontend" target="itvdc.↩
       VBDBackend" bidir="true"/>

    <CompositeRule name="FilePath" action="follow">
      <source>
        <NodeTypeMatch type="itvdc.File"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.FileSystem"/>
      </target>
      <colorTransform class="ColorTagPush">
        <tagName>filePath</tagName>
        <tagValue class="SourceNodeAttr">
          <attrName>relativePath</attrName>
        </tagValue>
      </colorTransform>
    </CompositeRule>

    <CompositeRule name="RemoteStorage" action="follow">
      <source>
        <NodeTypeMatch type="itvdc.FileSystem"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.StorageProvider"/>
      </target>
      <color>
          <ColorNoTagsMatch/>
      </color>
      <colorTransform class="ColorTagPush">
```

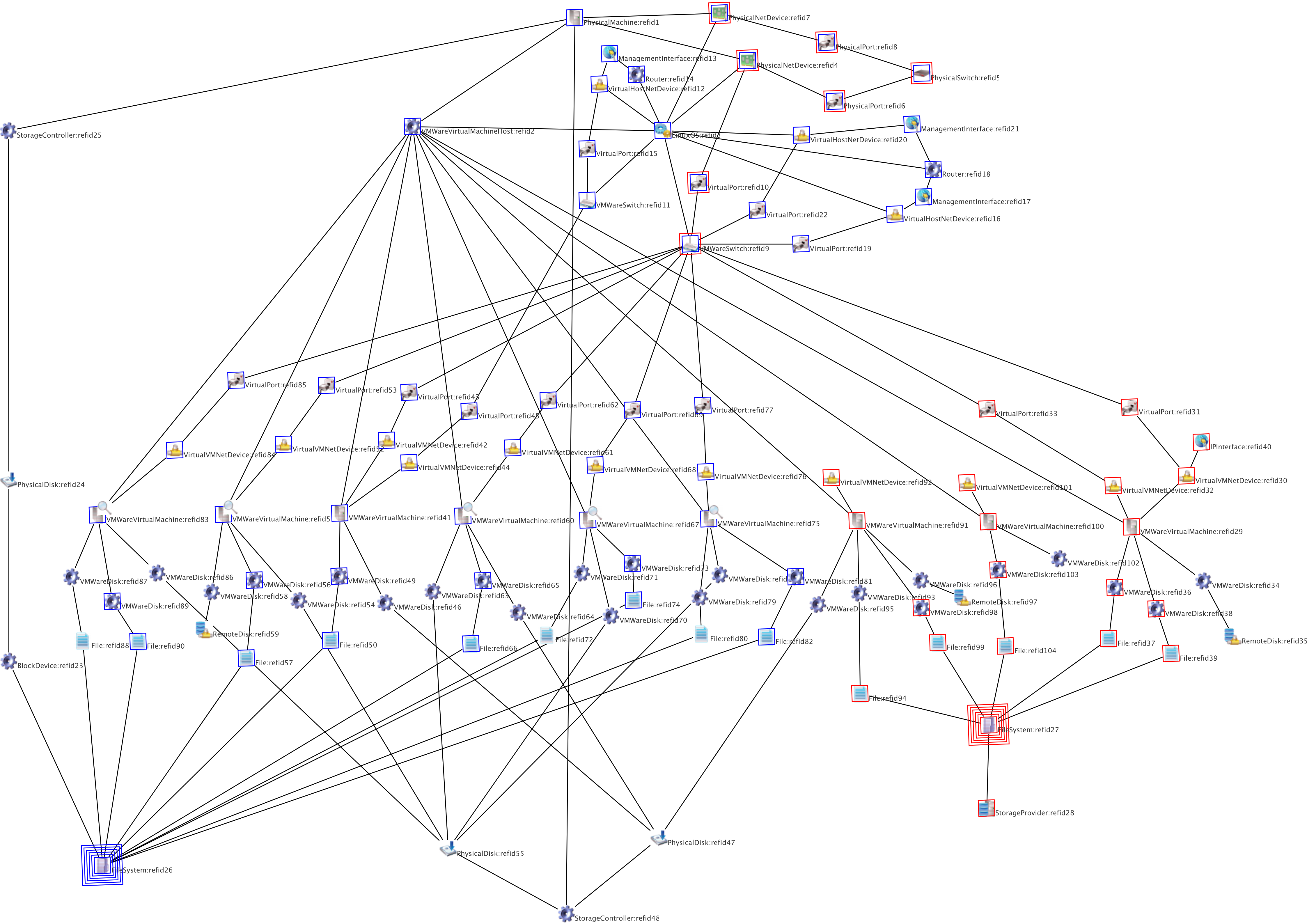
```
        <tagName>share</tagName>
        <tagValue class="SourceNodeAttr">
          <attrName>device</attrName>
        </tagValue>
      </colorTransform>
    </CompositeRule>

    <CompositeRule name="FileSystemBaseColored" action="follow"↩
       >
      <source>
        <NodeTypeMatch type="itvdc.FileSystem"/>
      </source>
      <target>
        <NodeTypeMatch type="itvdc.Node"/>
      </target>
      <color>
          <ColorNoTagsMatch/>
      </color>
    </CompositeRule>
  </colorRules>
</AnalysisParams>
```

Listing C.1: SAVE Coloring Policy

C.1.2 Figure

Figure C.1 on page 103.


PhysicalMachine:refid1
PhysicalNetDevice:refid7
PhysicalPort:refid8
ManagementInterface:refid13
PhysicalNetDevice:refid4
Router:refid14
PhysicalSwitch:refid5
VirtualHostNetDevice:refid12
PhysicalPort:refid6
StorageController:refid25
VMWareVirtualMachineHost:refid2
LinuxOS:refid3
VirtualHostNetDevice:refid20
ManagementInterface:refid21
VirtualPort:refid15
Router:refid18
VirtualPort:refid10
VMWareSwitch:refid11
ManagementInterface:refid17
VirtualPort:refid22
VirtualHostNetDevice:refid16
VMWareSwitch:refid9
VirtualPort:refid19
VirtualPort:refid85
VirtualPort:refid53
VirtualPort:refid43
VirtualPort:refid62
VirtualPort:refid45
VirtualPort:refid69
VirtualPort:refid77
VirtualPort:refid33
VirtualPort:refid31
IPInterface:refid40
VirtualVMNetDevice:refid84
VirtualVMNetDevice:refid52
VirtualVMNetDevice:refid42
VirtualVMNetDevice:refid44
VirtualVMNetDevice:refid61
VirtualVMNetDevice:refid68
VirtualVMNetDevice:refid76
VirtualVMNetDevice:refid92
VirtualVMNetDevice:refid101
VirtualVMNetDevice:refid30
VirtualVMNetDevice:refid32
PhysicalDisk:refid24
VMWareVirtualMachine:refid83
VMWareVirtualMachine:refid5
VMWareVirtualMachine:refid41
VMWareVirtualMachine:refid60
VMWareVirtualMachine:refid67
VMWareVirtualMachine:refid75
VMWareVirtualMachine:refid91
VMWareVirtualMachine:refid100
VMWareVirtualMachine:refid29
VMWareDisk:refid102
VMWareDisk:refid87
VMWareDisk:refid86
VMWareDisk:refid56
VMWareDisk:refid49
VMWareDisk:refid65
VMWareDisk:refid73
VMWareDisk:refid71
VMWareDisk:refid
VMWareDisk:refid81
VMWareDisk:refid96
VMWareDisk:refid103
VMWareDisk:refid36
VMWareDisk:refid34
VMWareDisk:refid89
VMWareDisk:refid58
VMWareDisk:refid54
VMWareDisk:refid46
VMWareDisk:refid63
VMWareDisk:refid64
VMWareDisk:refid70
VMWareDisk:refid79
VMWareDisk:refid95
VMWareDisk:refid93
VMWareDisk:refid98
VMWareDisk:refid38
RemoteDisk:refid59
RemoteDisk:refid97
RemoteDisk:refid35
File:refid88
File:refid90
File:refid57
File:refid50
File:refid66
File:refid72
File:refid74
File:refid80
File:refid82
File:refid99
File:refid104
File:refid37
File:refid39
File:refid94
BlockDevice:refid23
FileSystem:refid27
StorageProvider:refid28
FileSystem:refid26
PhysicalDisk:refid55
PhysicalDisk:refid47
StorageController:refid48

Appendix D

SAVEly Implementation

```
import sys
import time
import copy
import os
import tempfile
import ConfigParser
import subprocess
import re
from datetime import datetime
from xml.etree.ElementTree import ElementTree

import networkx as nx

from boto.ec2.connection import EC2Connection

import netaddr
from netaddr import IPSet


# Global AWS connection and configuration
aws_conn = None
config = None

# Caches
cache_subeq = {}



#
# Helper Functions
#

# Return dictionary keyed by security group name with a list of↩
    firewall rules
```

```
# in form of a tuple:
# (source, type of source, low port, high port, protocol)
def get_sgs_rules(sgs):
    rules = {}
    for sg in sgs:
        if not sg.name in rules:
            rules[sg.name] = []
        for r in sg.rules:
            for gr in r.grants:
                # IP-based source
                if gr.cidr_ip:
                    s, t = gr.cidr_ip, 'ip'
                # Security Group-based source
                else:
                    s, t = gr.name, 'sg'
                rules[sg.name].append((s, t, r.from_port, r.↩
                   to_port,
                    r.ip_protocol))
    return rules

# Return dictionary keyed by security group name with a list of↩
    AMIs running in
# that security group
def get_sgs_amis(res):
    amis = {}
    for r in res:
        for g in r.groups:
            amis[g.id] = []
            for i in r.instances:
                i.update()
                if i.state in ['pending', 'running']:
                    amis[g.id].append(i.image_id)
    return amis

# Returns true if s1 is subset or equal to s2
# Check is slow, therefore we cache
def is_src_subeq(s1, s2):
    global cache_subeq
    if (s1, s2) in cache_subeq:
        return cache_subeq[(s1, s2)]
    try:
        # Try CIDR
        s1_, s2_ = IPSet([s1]), IPSet([s2])
        b = s1_.issubset(s2_)
        cache_subeq[(s1, s2)] = b
        return b
    except netaddr.core.AddrFormatError:
        # Sec Groups
        b = s1 == s2
        cache_subeq[(s1, s2)] = b
        return b

def visual_graph(g, file):
    d = nx.to_pydot(g, file)
    d.write_png(file)
    #d.write_ps(file+'.ps')



#
# Reachability Graphs Construction
#

# Construct the Reachability Graph based on the discovered ↩
    Security Groups
def build_sg_reach(sgs):
    def edge_label(p1, p2, proto):
        p = str(p1)
        if p1 <> p2:
            p = str(p1) + '-' + str(p2)
        return p + '/' + proto

    rules = get_sgs_rules(sgs)
    g = nx.MultiDiGraph()

    for sg in sgs:
        g.add_node(sg.name)
        for (s, t, p1, p2, proto) in rules[sg.name]:
            if not s in g:
                g.add_node(s, type=t)
            g.add_edge(s, sg.name, label=edge_label(p1, p2, ↩
                proto))

    return g

# Extend a Reachability Graph with AMI memberships
def build_sg_ami(sgs, res):
    g = build_sg_reach(sgs)
    amis = get_sgs_amis(res)

    for (sg, amis) in amis.iteritems():
        for ami in amis:
            if not ami in g:
                g.add_node(ami)
            if not g.has_edge(sg, ami):
                g.add_edge(sg, ami, label='contains')

    return g
```

```
#
# Attack Graph Construction
#

# Analyze (i.e. vulnerability scan) a specific running instance↩
     and return a
# list of tuples:
# (port, proto, max severity rating)
def analyze_instance(ins):
    def rmax(rs):
        w = {'None': 0, 'Low': 1, 'Medium': 2, 'High': 3}
        max = 0
        maxr = 'None'
        for r in rs:
            if w[r] > max:
                maxr = r
                max = w[r]
        return maxr

    rate = {}

    print 'analyze:', ins.image_id, ins.public_dns_name

    user = config.get('OpenVAS', 'user')
    pw = config.get('OpenVAS', 'pass')
    host = config.get('OpenVAS', 'host')
    port = config.get('OpenVAS', 'port')

    # create target file
    t = os.path.join(tempfile.gettempdir(), 'savely_target_' + ↩
        ins.image_id)
    f = open(t, 'w')
    f.write(ins.dns_name + '\n')
    f.close()

    # filename for scan result
    r = os.path.join(tempfile.gettempdir(), 'savely_scan_' + ↩
        ins.image_id)

    # execute OpenVAS
    ret = subprocess.call(['OpenVAS-Client', '-T', 'xml', '-q',↩
         host, port, user,
        pw, t, r])

    # Parse OpenVAS/Nessus scan result
    x = ElementTree()
    x.parse(r)
    ports = x.findall('.//ports/port')
    risk = re.compile('Risk factor : (\w+)')
    for p in ports:
        if not 'portid' in p.attrib:
            continue
        port = p.attrib['portid']
        proto = p.attrib['protocol']
        if not port in rate:
            rate[(port, proto)] = []
        data = p.findall('information/data')
        for d in data:
            m = risk.search(d.text)
            if not m is None:
                rate[(port, proto)].append(m.group(1))

    # Delete temp files
    os.remove(r)
    os.remove(t)

    return map(lambda ((port, proto), rs): (port, proto, rmax(↩
        rs)), rate.iteritems())

# Analyze a list of AMIs by spawning new instances of each AMI ↩
    in a special
# scanning security group and return a vulnerability rating for↩
    each AMI's
# services
def analyze_amis(ami_ids):
    rate = {}
    runs = []

    # Create secgroup for scanning
    scan = aws_conn.create_security_group('savely_scan', 'scan ↩
        for savely')
    source = config.get('Scan', 'source')
    scan.authorize('tcp', 1, 65535, source)
    scan.authorize('udp', 1, 65535, source)
    scan.authorize('icmp', -1, -1, source)

    # Launch AMI
    amis = aws_conn.get_all_images(image_ids=ami_ids)
    for a in amis:
        print 'run instance', a.id
        runs.append(a.run(security_groups=[scan]))

    # Wait for running instances and then scan
    rs = copy.copy(runs)
    while len(rs) > 0:
        sys.stdout.write('.')
        sys.stdout.flush()
        time.sleep(5)
        for r in rs:
            i = r.instances[0]
            i.update()
            if i.state == 'running':
                rs.remove(r)
                rate[i.image_id] = analyze_instance(i)

    # Terminate instances
    for r in runs:
        r.instances[0].stop()

    # Remove scan secgroup
    scan.delete()

    return rate

def build_attack(sgs, res):
    # Obtain severity ratings for AMIs
    def get_ratings():
        ami_ids = []
        for r in res:
            for i in r.instances:
                if i.state <> 'running':
                    continue
                id = i.image_id
                if id in ami_ids:
                    continue
                ami_ids.append(id)
        return analyze_amis(ami_ids)

    def is_port_allowed(port, proto, rule):
        (s, t, p1, p2, pr) = rule
        return pr == proto and (int(p1) <= int(port) and int(↩
            port) <= int(p2))

    g = nx.MultiDiGraph()

    rates = get_ratings()
    rules = get_sgs_rules(sgs)

    # construct attack graph
    amis = get_sgs_amis(res)
    for (sg, amis) in amis.iteritems():
        for ami in amis:
            for (port, proto, rate) in rates[ami]:
                if rate == 'None':
                    continue
                for rule in rules[sg]:
                    if is_port_allowed(port, proto, rule):
                        # add rule source
                        if not rule[0] in g:
                            g.add_node(rule[0], type=rule[1])

                        # add AMI
                        if not ami in g:
                            g.add_node(ami)

                        # edge between source and ami with port↩
                            and rating
                        l = port + '/' + proto + ' - ' + rate
                        g.add_edge(rule[0], ami, label=l, rate=↩
                            rate)

    return g


#
# Reachability Query and Policy Processing
#

def _reach_parse(q, prefix=''):
    # TODO: fix space sensitivity
    r = prefix + 'from (?P<src>.+) to (?P<dst>.+) port \
(?P<p1>.+?)(?:-(?P<p2>.+))? proto (?P<proto>\w+)'
    m = re.match(r, q)
    if m is None:
        raise Exception('parse error in: ' + q)
    return m.groups()

# get edges concerning source 's' and destination 'd'
def get_rel_edges(g, s, d):
    es = []
    for e in g.edges(data=True):
        if (s == 'any' or is_src_subeq(s, e[0])) \
                and (d == 'any' or d == e[1]):
            es.append(e)
    return es

def parse_edge_label(label):
    ps, proto = label.split('/')
    m = re.match('(\d+)-(\d+)', ps)
    if not m is None:
        p1, p2 = int(m.group(1)), int(m.group(2))
    else:
        p1 = p2 = int(ps)
    return (p1, p2, proto)
```

```
def parse_only_policy(p):
    if not 'and' in p:
        return [_parse(p, 'only ')]

    pre = p.split('port')[0]
    ports = map(lambda x: x.strip(), p.split('and'))
    r = [_reach_parse(ports[0], 'only ')]
    for port in ports[1:]:
        r.append(_reach_parse(pre + port, 'only '))

    return r

def parse_never_policy(p):
    return _reach_parse(p, 'never ')

def reach_never_policy(g, p):
    return reach_query(g, p) == False

# verify exclusiveness of only policy
def is_only_exclusive(g, ps):
    s, d = ps[0][:2]

    for e in get_rel_edges(g, s, d):
        b = dict([(p, True) for p in ps])
        p1_, p2_, proto = parse_edge_label(e[2]['label'])

        for p in ps:
            (p1, p2, pr) = p[2:]
            if p2 is None:
                p2 = p1
            if (pr <> 'any' and pr <> proto) or \
                    (p1 <> 'any' and (p1_ < int(p1) or int(p2) ↩
                       < p2_)):
                        b[p] = False

        if not True in b.values():
            return False

    return True

def reach_only_policy(g, ps):
    reach = all(map(lambda p: reach_query(g, p), ps))
    excl = is_only_exclusive(g, ps)
    return reach and excl

def process_reach_policy_file(g, file):
    def process(l):
        if l.startswith('never'):
            return reach_never_policy(g, parse_never_policy(l))
        if l.startswith('only'):
            return reach_only_policy(g, parse_only_policy(l))
        raise Exception('unsupported policy: ' + l)

    return all(map(lambda x: process(x.strip()), open(file).↩
        readlines()))

def reach_query(g, q):
    (s, d, p1, p2, pr) = q
    if p2 is None:
        p2 = p1

    for e in get_rel_edges(g, s, d):
       p1_, p2_, proto = parse_edge_label(e[2]['label'])
       if (pr == 'any' or pr == proto) and (p1 == 'any' or (p1_↩
            <= int(p1) and
             int(p2) <= p2_)):
            return True

    return False

def parse_reach_query(q):
    return _reach_parse(q)

def process_reach_query_file(g, file):
    for l in open(file).readlines():
        print l, '=>', reach_query(g, parse_reach_query(l))



#
# Attack Graph Query and Policy Processing
#

def _attack_parse(q, prefix=''):
    # TODO: fix space sensitivity
    r = prefix + 'from (?P<src>.+) to (?P<dst>.+) vuln (?P<vuln↩
       >.+)'
    m = re.match(r, q)
    if m is None:
        raise Exception('parse error in: ' + q)
    return m.groups()

# find the weakest path for query 'q'
def attack_weakest_path(g, q):
    def rate2weight(r):
        d = {'high': 1, 'medium': 2, 'low': 3}
        return d[r.lower()]

    def rate2num(r):
```

```
    d = {'high': 3, 'medium': 2, 'low': 1}
    return d[r.lower()]

def is_wrong_loop(k, v, loops):
    return len(v) == 1 and k == v[0] and not k in loops

(s, d, v) = q

g_ = g.copy()

# Set edge weight based on vulnerability rating
# Remove edges with too low vuln rating regarding query
for e in g_.edges(data=True):
    r = e[2]['rate']
    if v <> 'any' and rate2num(r) < rate2num(v):
        g_.remove_edge(e[0], e[1])
        continue
    e[2]['weight'] = rate2weight(r)

loops = g_.nodes_with_selfloops()

if s <> 'any':
    # Find all nodes superset of s
    ss = filter(lambda x: is_src_subeq(s, x), g_.nodes_iter↩
        ())
    ps = []
    for s in ss:
        if d <> 'any':
            try:
                ps.append(nx.dijkstra_path(g_, s, d))
            except nx.exception.NetworkXError:
                continue
        else:
            # filter loop paths
            ps.extend([v for (k,v) in nx.↩
                single_source_dijkstra_path(g_, s).↩
                iteritems() \
                    if not is_wrong_loop(k, v, loops)])
    return ps
else:
    if d <> 'any':
        # Start Dijkstra from destination node in reversed ↩
            graph
        g_.reverse(copy=False)
        return [v[::-1]
                for (k,v) in nx.single_source_dijkstra_path↩
                    (g_, d).iteritems() \
                if not is_wrong_loop(k, v, loops)]
        else:
            # All pairs dijkstra
            return [v_
                    for (k, v) in nx.all_pairs_dijkstra_path(g_↩
                        ).iteritems() \
                    for (k_, v_) in v.iteritems() \
                    if not (is_wrong_loop(k_, v_, loops) and k ↩
                        == k_)]

# check if any path exist for query 'q'
def attack_query(g, q):
    try:
        p = attack_weakest_path(g, q)
    except KeyError as e:
        print 'unknown node in query:', e
        return False
    print p
    return len(p) > 0

def parse_attack_query(q):
    return _attack_parse(q)

def process_attack_query_file(g, file):
    for l in open(file).readlines():
        print l, '=>', attack_query(g, parse_attack_query(l))

def parse_attack_policy(p):
    i = p.find('from')
    return _attack_parse(p[i:])

def attack_only_policy(g, p):
    (s, d, v) = p
    # any path for v+1?, if v == high/any -> true
    if v == 'high' or v == 'any':
        return True
    if v == 'low':
        return attack_never_policy(g, (s, d, 'medium'))
    if v == 'medium':
        return attack_never_policy(g, (s, d, 'high'))
    raise Exception('unsupported vuln: ' + v)

def attack_never_policy(g, p):
    # No attack path
    a = attack_weakest_path(g, p)
    if len(a) > 0:
        print 'policy', p, 'violation:', a
    else:
        return True
```

```
def process_attack_policy_file(g, file):
    def process(l):
        if l.startswith('never'):
            return attack_never_policy(g, parse_attack_policy(l↩
                ))
        if l.startswith('only'):
            return attack_only_policy(g, parse_attack_policy(l)↩
                )
        raise Exception('unsupported policy: ' + l)

    return all(map(lambda x: process(x.strip()), open(file).↩
        readlines()))

# construct test attack graph
def test_ag():
    g = nx.MultiDiGraph()
    i = '0.0.0.0/0'
    c = '1.2.3.4/24'
    a = ['AMI0', 'AMI1', 'AMI2', 'AMI3', 'AMI4']
    g.add_nodes_from([i, c] + a[1:])
    g.add_edges_from([(i, a[1], {'rate': 'low'}),
        (i, a[1], {'rate': 'medium'}),
        (a[1], a[2], {'rate': 'medium'}), (a[2], a[4], {'rate':↩
             'medium'}),
        (a[2], a[3], {'rate': 'low'}), (c, a[1], {'rate': 'low'↩
            }),
        (c, a[2], {'rate': 'high'}), (c, a[3], {'rate': 'low'})↩
            ,
        (c, a[4], {'rate': 'low'})
        ])

    for (s,t,d) in g.edges_iter(data=True):
        d['label'] = d['rate']

    return g


def main():
    global config
    config = ConfigParser.ConfigParser()
    config.read('savely.cfg')
    akey = config.get('AWS', 'access')
    skey = config.get('AWS', 'secret')
    conn = EC2Connection(akey, skey)

    global aws_conn
    aws_conn = conn

    # SG Reachability Graph
    sgs = conn.get_all_security_groups()
    rg = build_sg_reach(sgs)
    visual_graph(rg, 'reach.png')

    process_reach_query_file(rg, 'queries')
    print 'Reachability Policy valid:', \
        process_reach_policy_file(rg, 'policies')

    # AMI-SG Graph
    res = conn.get_all_instances()
    arg = build_sg_ami(sgs, res)
    visual_graph(arg, 'ami.png')

    # Attack Graph
    ag = build_attack(sgs, res)
    #ag = test_ag()
    visual_graph(ag, 'attack.png')

    process_attack_query_file(ag, 'queries2')
    print 'Attack Policy valid:', process_attack_policy_file(ag↩
        , 'policies2')

if __name__ == '__main__':
    main()
```
Listing D.1: SAVEly Python Implementation