Self-adaptable Security Monitoring for IaaS Cloud Environments

Anna Giannakou

To cite this version:

Anna Giannakou. Self-adaptable Security Monitoring for IaaS Cloud Environments. Cryptography and Security [cs.CR]. INSA de Rennes, 2017. English. NNT : 2017ISAR0021 . tel-01653831v2

HAL Id: tel-01653831

https://inria.hal.science/tel-01653831v2

Submitted on 20 Apr 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse

UNIVERSITE
BRETAGNE
LOIRE

THESE INSA Rennes
sous le sceau de l'Université Bretagne Loire
pour obtenir le titre de
DOCTEUR DE L'INSA RENNES
Spécialité : Informatique

présentée par
Anna Giannakou
ECOLE DOCTORALE : *Matisse*
LABORATOIRE : *Irisa*

Self-adaptable Security Monitoring for IaaS Cloud Environments

Thèse soutenue le 06.07.2017
devant le jury composé de :

Eric Totel
Professeur, Centrale-Supélec / Président
Sara Bouchenak
Professeur, INSA Lyon / Rapporteur
Hervé Debar
Professeur, Télécom SudParis / Rapporteur
Eddy Caron
Maître de Conférences, HDR, ENS Lyon / Examinateur
Stephen Scott
Professeur, Tennessee Tech University / Examinateur
Christine Morin
Directrice de Recherche, INRIA Rennes / Co-directrice de thèse
Jean-Louis Pazat
Professeur, INSA Rennes / Directeur de thèse
Louis Rilling
Ingénieur-Chercheur, DGA MI / Co-encadrant de thèse

Self-adaptable Security Monitoring for IaaS Cloud Environments

Anna Giannakou

Document protégé par les droits d'auteur

Publications:

- National
  - Workshops :
    - Giannakou, Anna, Louis Rilling, Frédéric Majorczyk, Jean-Louis Pazat, and Christine Morin. “Self Adaptation in Security Monitoring for IaaS clouds”. In: *EIT Digital Future Cloud symposium, Rennes, France, October 19-20, 2015*
- International
  - Conferences :
    - Giannakou,Anna,Louis Rilling,Jean-Louis Pazat,andChristine Morin.“AL-SAFE: A Secure Self-Adaptable Application-Level Firewall for IaaS Clouds”. In: *2016 IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2016,Luxembourg, Luxembourg, December 12-15, 2016*, pp. 383–390
    - Giannakou, Anna, Louis Rilling, Jean-Louis Pazat, Frédéric Majorczyk, and Chris- tine Morin. “Towards Self Adaptable Security Monitoring in IaaS Clouds”. In: *15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CC- Grid 2015, Shenzhen, China, May 4-7, 2015*, pp. 737–740.
  - Workshops :
    - Giannakou, Anna, Louis Rilling, Jean-Louis Pazat, and Christine Morin. “How to Secure Application-Level Firewalls in IaaS Clouds”. In: *International Workshop on Cloud Data and Platforms, CloudDP, London, UK, April 17, 2016*

Acknowledgements

First and foremost I would like to like to thank my advisors for their outstanding guidance and support throughout the duration of this thesis. Christine, thank you for continuously reviewing my work, offering important insights and improvements. Your advice regarding my professional development after the PhD helped me make important decisions about my future. During the last three and a half years you have been a role model for me as a woman in research. Louis, words cannot express how grateful I am for your guidance and support all these years. You have taught me so many things and helped me achieve my goals at so many different levels. Thank you for showing me all these new directions and possibilities and for helping me grow as a researcher. Also, thank you for tirelessly listening me complain about not having enough results :). Jean-Louis, I am grateful for your guidance throughout the whole process.

Furthermore, I would like to thank the members of my committee and especially the reviewers Sara Bouchenak and Herve Debar for evaluating my work. Special thanks goes out to all the members of the Myriads team for creating a warm and welcoming atmosphere at the office. David, Yunbo and Amir, for all of our discussions and for being such wonderful people to interact with. Deb and Sean, thank you for hosting me at Lawrence Berkeley National Lab for my three month internship and for allowing me to explore new research directions.

This thesis would not have been possible without the endless love and support of my friends and family. Genc, I am so grateful that I have met you and I am proud to call you my buddinis. Thank you for listening my complains offering helpful insights every time I invaded your office :). Bogdan and Mada, you have both been so wonderful and special to me. To Tsiort, Magnum, Ziag and Fotis thank you for your honest and deep support throughout these years from thousands of miles away. I love and miss you guys so much. To Irene, you have been nothing less than exceptional, kolitoula. I cannot express how deeply grateful I am for your endless encouragement and advice all this time. To Iakovos, thank you for your stoic comments and for all of our arguments :). To Eri, thank you for all your support and your clear-headed guidance throughout these years. You are admirable and you have given me so much. To my parents, thank you for your love, patience and support that has allowed me to pursue my ambitions. Thank you for raising me as a strong independent person and for showing me the benefits of persistence. To my sister Maria, thank you for being there, always.

Finally, the biggest thank you goes out to a single person that has been by my side for the last five years. Ismael, words cannot describe how grateful I am for all the things that you have given me throughout this period. Thank you for helping me break my personal deadlocks in so many levels and for adding all these new dimensions in my life. I could not have done this without you and I will never forget that. I love you, always, my sun and stars.

Contents

1 Introduction **11**
1.1 Context . . . 11
1.2 Motivation . . . 11
1.3 Objectives . . . 12
1.3.1 Self-Adaptation . . . 12
1.3.2 Tenant-Driven Customization . . . 12
1.3.3 Security and Correctness . . . 13
1.3.4 Cost Minimization . . . 13
1.4 Contributions . . . 13
1.4.1 A Self-Adaptable Security Monitoring Framework . . . 13
1.4.2 SAIDS . . . 13
1.4.3 AL-SAFE . . . 14
1.5 Thesis Outline . . . 14

2 State of the Art **17**
2.1 Autonomic Computing . . . 17
2.1.1 What is Autonomic Computing? . . . 17
2.1.2 Characteristics . . . 17
2.1.3 The Role of the Manager . . . 18
2.2 Cloud Computing . . . 19
2.2.1 What is Cloud Computing? . . . 19
2.2.2 Characteristics . . . 20
2.2.3 Service Models . . . 20
2.2.4 Deployment Models . . . 22
2.2.5 Dynamic Events in Iaas Clouds and Cloud Adaptation . . . 23
2.3 Virtualization . . . 24
2.3.1 Server Virtualization Components . . . 24
2.3.2 Server Virtualization . . . 24
2.3.3 Network Virtualization and Network Management in IaaS Clouds . . . 27
2.4 Security Threats . . . 29
2.4.1 Security Threats in Information Systems . . . 29
2.4.2 Security Threats in Cloud Environments . . . 32
2.4.3 Summary . . . 33
2.5 Security Monitoring . . . 33
2.5.1 What is Security Monitoring? . . . 33
2.5.2 Security Monitoring in Cloud Environments . . . 38
2.6 Summary . . . 50

3 A Self-Adaptable Security Monitoring Framework for IaaS Clouds 53
3.1 Introduction 53
3.2 System Model 54
3.3 Threat Model 54
3.4 Objectives 55
3.4.1 Self Adaptation 55
3.4.2 Tenant-Driven Customization 55
3.4.3 Security and Correctness 56
3.4.4 Cost Minimization 56
3.5 Example Scenario 56
3.6 Adaptation Process 57
3.7 Architecture 58
3.7.1 High-Level Overview 58
3.7.2 Tenant-API 59
3.7.3 Security Devices 62
3.7.4 Adaptation Manager 62
3.7.5 Infrastructure Monitoring Probes 64
3.7.6 Component Dependency Database 64
3.8 Implementation 65
3.8.1 Adaptation Manager 65
3.8.2 Infrastructure Monitoring Probe 67
3.9 Summary 67

4 SAIDS: A Self-Adaptable Intrusion Detection System for IaaS Cloud Environments 69
4.1 Objectives 69
4.2 Models and Architecture 70
4.2.1 Architecture 70
4.3 Security Threats 72
4.3.1 SAIDS Configuration Files 72
4.3.2 LIDS Rules 73
4.3.3 SAIDS Adaptation Sources 73
4.3.4 Connection Between SAIDS Components 73
4.3.5 External Traffic 73
4.4 Adaptation process 74
4.4.1 Events Triggering Adaptation 74
4.4.2 Adaptation Process 74
4.4.3 Topology-Related Change 75
4.4.4 Traffic-Related Change 76
4.4.5 Service-Related Change 76
4.5 Implementation 77
4.6 Evaluation 78
4.6.1 Objectives of the Evaluation 78
4.6.2 Experimentation Methodology 81
4.6.3 Result Analysis 83
4.7 Summary 90

5 AL-SAFE: A Secure Self-Adaptable Application-Level Firewall for IaaS Clouds 93
5.1 Requirements . . . 93
5.1.1 Why Should we Secure an Application-level Firewall . . . 94
5.1.2 Security and Visibility . . . 94
5.1.3 Self-Adaptable Application-Level Firewall . . . 94
5.2 Models and Architecture . . . 95
5.2.1 Events that Trigger Adaptation . . . 95
5.2.2 Component Description . . . 95
5.3 Adaptation Process . . . 97
5.3.1 Security Threats . . . 100
5.4 Implementation . . . 101
5.4.1 Edge Firewall . . . 101
5.4.2 Switch-Level Firewall . . . 101
5.4.3 VMI . . . 101
5.4.4 Information Extraction Agent . . . 104
5.4.5 Rule Generators . . . 105
5.5 Evaluation Methodology . . . 105
5.5.1 Objectives of the Evaluation . . . 105
5.5.2 Experimentation Methodology . . . 107
5.6 Evaluation Results . . . 114
5.6.1 Performance and Cost Analysis . . . 114
5.6.2 Correctness Analysis . . . 122
5.6.3 Limitations . . . 123
5.7 Summary . . . 124

6 Conclusion 125
6.1 Contributions . . . 125
6.2 Future Work . . . 127
6.2.1 Short-Term Goals . . . 127
6.2.2 Mid-Term Goals . . . 128
6.2.3 Long-Term Goals . . . 130

Annexe A Résumé en français 143
A.1 Contexte . . . 143
A.2 Motivation . . . 144
A.3 Objectifs . . . 144
A.3.1 Auto-adaptation . . . 144
A.3.2 Personnalisation . . . 145
A.3.3 Sécurité et correction . . . 145
A.3.4 Minimisation des coûts . . . 145
A.4 Contributions . . . 145
A.4.1 Un système de supervision de sécurité auto-adaptatif . . . 146
A.4.2 SAIDS . . . 146
A.4.3 AL-SAFE . . . 147
A.5 Perspectives . . . 147
A.5.1 Perspectives à court terme . . . 147
A.5.2 Perspectives à moyen terme . . . 147
A.5.3 Perspectives à long terme . . . 148

List of Figures

2.1 The MAPE control loop . . . 19
2.2 The OpenStack architecture . . . 22
2.3 The SDN architecture . . . 28
2.4 Information system with different security devices contributing to security monitoring . . . 34
2.5 A DMZ example . . . 37
2.6 Hypervisor and host OS kernel . . . 39
2.7 The Cloud IDS architecture as in [1] . . . 43
2.8 The Livewire architecture as in [2] . . . 44
2.9 CloudSec architecture as in [3] . . . 46

3.1 An example of a cloud hosted information system . . . 56
3.2 The framework's architecture . . . 59
3.3 The framework's different levels . . . 60

4.1 SAIDS architecture . . . 71
4.2 Migration time with and without SAIDS . . . 84
4.3 Adaptation time breakdown when SAIDS only reconfigures the enforced ruleset inside the LIDS . . . 84
4.4 Adaptation time breakdown when SAIDS has to start a new LIDS, distribute traffic and create a mirroring tunnel . . . 85
4.5 MAD scalability setup . . . 86
4.6 MAD response time . . . 86
4.7 AM scalability setup . . . 87
4.8 AM response time . . . 88

5.1 The AL-SAFE architecture with the Adaptation Manager . . . 96
5.2 Steps of the AL-SAFE adaptation . . . 98
5.3 The migration request arrives between two introspections . . . 99
5.4 The migration request arrives during an introspection . . . 99
5.5 LibVMI stack . . . 102
5.6 Adaptation process flow chart . . . 103
5.7 Snapshot-Introspection relationship . . . 104
5.8 TCP server setup . . . 111
5.9 TCP client setup . . . 112
5.10 UDP setup . . . 113
5.11 Migration time with and without adaptation . . . 114
5.12 Breakdown of each phase in seconds . . . 115
5.13 Impact of the introspection period on kernel compilation time . . . 115

5.14 Impact of the introspection period on server throughput 116
5.15 Request service time for different times in the introspection cycle 117
5.16 Cases of request arrival time with respect to the introspection cycle 117
5.17 Impact of the introspection period on network throughput 118
5.18 Cases of request arrival time with respect to the introspection cycle 119
5.19 Inbound TCP connection establishment time 120
5.20 Outbound TCP connection establishment time 121
5.21 Inbound UDP round trip time 121

List of Tables

3.1 The VM_info table 65
3.2 The Device_info table 65

4.1 Events that trigger adaptation 74
4.2 Resource consumption of the AM component 89

5.1 Events that trigger adaptation 95
5.2 Resource consumption of the introspection component 122

Chapter 1

Introduction

1.1 Context

Server virtualization enables on-demand allocation of computational resources (e.g. CPU and RAM) according to the pay-as-you-go model, a business model where users (referred to as tenants) are charged only for as much as they have used. One of the main cloud models that has gained significant attention over the past few years is the *Infrastructure as a Service* model where compute, storage, and network resources are provided to tenants in the form of *virtual machines (VMs)* and virtual networks. Organizations outsource part of their information systems to virtual infrastructures (composed of VMs and virtual networks) hosted on the physical infrastructure of the cloud provider. The terms that regulate the resource allocation are declared in a contract signed by the tenants and the cloud provider, the *Service Level Agreement (SLA)* [4]. Few of the main benefits of the *IaaS* cloud include: flexibility in resource allocation, illusion of unlimited capacity of computational and network resources and automated administration of complex virtualized information systems.

Although shifting to the cloud might provide significant cost and efficiency gains, security continues to remain one of the main concerns in the adoption of the cloud model [5]. Multi-tenancy, one of the key characteristics of a cloud infrastructure, creates the possibility of legitimate VMs being colocated with malicious, attacker-controlled VMs. Consequently, attacks towards cloud infrastructures may originate from inside as well as the outside of the cloud environment [6]. A successful attack could allow attackers to gain access and manipulate cloud-hosted data including legitimate user's account credentials or even gain complete control of the cloud infrastructure and turn it into a malicious entity [7]. Although traditional security techniques such as traffic filtering or traffic inspection can provide a certain level of protection against attackers, they are not enough to tackle sophisticated threats that target virtual infrastructures. In order to provide a security solution for cloud environments, an automated self-contained security architecture that integrates heterogeneous security and monitoring tools is required.

1.2 Motivation

In a typical IaaS cloud environment, the provider is responsible for the management and maintenance of the physical infrastructure while tenants are only responsible for managing their own virtualized information system. Tenants can make decisions regarding VM life-cycle and deploy different types of applications on their provisioned VMs. Since deployed

applications may have access to sensitive information or perform critical operations, tenants are concerned with the security monitoring of their virtualized infrastructure. These concerns can be expressed in the form of monitoring requirements against specific types of threats. Security monitoring solutions for cloud environments are typically managed by the cloud provider and are composed of heterogeneous tools for which manual configuration is required. In order to provide successful detection results, monitoring solutions need to take into account the profile of tenant-deployed applications as well as specific tenant security requirements.

A cloud environment exhibits a very dynamic behavior with changes that occur at different levels of the cloud infrastructure. Unfortunately, these changes affect the ability of a cloud security monitoring framework to successfully detect attacks and preserve the integrity of the cloud infrastructure [8]. Existing cloud security monitoring solutions fail to address changes and take necessary decisions regarding the reconfiguration of the security devices. As a result, new entry points for malicious attackers are created which may lead to a compromise of the whole cloud infrastructure. To our knowledge, there still does not exist a security monitoring framework that is able to adapt its components based on different changes that occur in a cloud environment.

The goal of this thesis is to design and implement a self-adaptable security monitoring framework that is able to react to dynamic events that occur in a cloud infrastructure and adapt its components in order to guarantee that an adequate level of security monitoring for tenant's virtual infrastructures is achieved.

1.3 Objectives

After presenting the context and motivation for this thesis we now define a set of objectives for a self-adaptable security monitoring framework.

1.3.1 Self-Adaptation

A self-adaptable security monitoring framework should be able to adapt its components based on different types of dynamic events that occur in a cloud infrastructure. The framework should perceive these events as sources of adaptation and take subsequent actions that affect its components. The adaptation process may alter the configuration of existing monitoring devices or instantiate new ones. The framework may decide to alter the computational resources available to a monitoring device (or a subset of monitoring devices) in order to maintain an adequate level of monitoring. Adaptation of the amount of computational resources should also be performed in order to free under-utilized resources. The framework should make adaptation decisions in order to guarantee that a balanced trade-off between security, performance and cost is maintained at any given moment. Adaptation actions can affect different components and the framework should be able to perform these actions in parallel.

1.3.2 Tenant-Driven Customization

Tenant requirements regarding specific monitoring cases should be taken into account from a self-adaptable security monitoring framework. The framework should be able to guarantee that adequate monitoring for specific tenant-requested types of threats will be provided. The monitoring request could refer to a tenant's whole virtual infrastructure or to a specific subset of VMs. The framework should provide the requested type of monitoring until

the tenant requests otherwise or the subset of VMs that the monitoring type is applied to no longer exists. Furthermore, the framework should take into account tenant-defined (through specific SLAs) thresholds that refer to the quality of the monitoring service or to the performance of specific types of monitoring devices.

1.3.3 Security and Correctness

Deploying a self-adaptable security monitoring framework should not add new vulnerabilities in the monitored virtual infrastructure or in the provider's infrastructure. The adaptation process and the input sources required should not create new entry points for an attacker. Furthermore, a self-adaptable security monitoring framework should be able to guarantee that an adequate level of monitoring is maintained throughout the adaptation process. The adaptation process should not intervene with the ability of the framework to correctly detect threats.

1.3.4 Cost Minimization

Deploying a self-adaptable security monitoring framework should not significantly impact the trade-off between security and cost for both tenants and the provider. On the tenant's side a self-adaptable security monitoring framework should not significantly impact performance of the applications that are hosted in the virtual infrastructure regardless of the application profile (compute- or network-intensive). On the provider's side, the ability to generate profit by leasing it's computational resources should not be significantly affected by the framework. Deploying such a framework should not impose a significant penalty in normal cloud operations (e.g. VM migration, creation, etc). Furthermore, the amount of computational resources dedicated to the self-adaptable framework's components should reflect an agreement between tenants and the provider for the distribution of computational resources.

1.4 Contributions

In order to achieve the objectives presented in the previous section, we design a self-adaptable security monitoring that is able to address limitations in existing monitoring frameworks and tackle dynamic events that occur in a cloud infrastructure. In this thesis we detail how we designed, implemented, and evaluated our contributions: a generic self-adaptable security monitoring framework and two instantiations with intrusion detection systems and firewalls.

1.4.1 A Self-Adaptable Security Monitoring Framework

Our first contribution is the design of a self-adaptable security monitoring framework that is able to alter the configuration of its components and adapt the amount of computational resources available to them depending on the type of dynamic event that occurs in a cloud infrastructure. Our framework achieves self-adaptation and tenant-driven customization while providing an adequate level of security monitoring through the adaptation process.

1.4.2 SAIDS

Our second contribution constitutes the first instantiation of our framework focusing on network-based intrusion detection systems (NIDS). NIDSs are key components of a security

monitoring infrastructure. SAIDS achieves the core framework's objectives while providing a scalable solution for serving parallel adaptation requests. Our solution is able to scale depending on the load of monitored traffic and the size of the virtual infrastructure. SAIDS maintains an adequate level of detection while minimizing the cost in terms of resource consumption and deployed application performance.

1.4.3 AL-SAFE

Our third contribution constitutes the second instantiation of our framework focusing on application-level firewalls. AL-SAFE uses virtual machine introspection in order to create a secure application-level firewall that operates outside the monitored VM but retains inside-the-VM visibility. The firewall's enforced rulesets are adapted based on dynamic events that occur in a virtual infrastructure. AL-SAFE offers a balanced trade-off between security, performance and cost.

1.5 Thesis Outline

This thesis is organized as follows:

Chapter 2 reviews the state of the art while making important observations in the area of cloud computing security focusing on both industrial and academic solutions. We start by providing the context in which the contributions of this thesis were developed while describing fundamental concepts of autonomic and cloud computing. Security threats for traditional information systems as well as information systems outsourced in cloud infrastructures are presented. We then present the notion of traditional security monitoring along with key components and their functionality. Finally, security monitoring solutions for cloud environments are presented focusing on two different types of components, intrusion detection systems and firewalls.

Chapter 3 presents the design of our self-adaptable security monitoring framework that is the core of this thesis. The objectives that this framework needs to address are discussed in detail. Fundamental components and their interaction are presented in detail along with a first high-level overview of the adaptation process. This chapter concludes with important implementation aspects of two generic components of our framework.

Chapter 4 presents the first instantiation of our security monitoring framework which addresses network-based intrusion detection systems. This chapter details how the objectives set at the beginning are translated in design principles for a self-adaptable network-based IDS. This first instantiation, named SAIDS, is able to adapt the configuration of a network-based IDS upon the occurrence of different types of dynamic events in the cloud infrastructure. After presenting SAIDS design and main components we describe the adaptation process and how our design choices do not add new security vulnerabilities to the cloud engine. Finally, we evaluate SAIDS performance, scalability and correctness in experimental scenarios that resemble production environments.

Chapter 5 presents the second instantiation of the security monitoring framework, which focuses on a different type of security component, the firewall. This chapter maps the objectives of the security monitoring framework in the area of application-level firewalls proposing a new design for addressing inherent security vulnerabilities of this type of security device. This second instantiation, named AL-SAFE, brings self-adaptation to firewalls. We present in detail the adaptation process for addressing dynamic events and justify the correctness of our design choices. Finally, this chapter concludes with an ex-

perimental evaluation of our prototype that explores the trade-off between performance, cost and security both from the provider and the tenant's perspectives.

Chapter 6 concludes this thesis with a final analysis of the contributions presented and the objectives that were set in the beginning. We demonstrate how our framework's design and the two subsequent instantiations satisfy the objectives presented in this chapter. We then present perspectives to improve performance aspects of our two prototypes, SAIDS and AL-SAFE, along with ideas to expand this work organised in short, mid and long terms goals.

Chapter 2

State of the Art

In this thesis we propose a design for a self-adaptable security monitoring framework for IaaS cloud environments. In order to provide the necessary background for our work, we present the state of the art around several concepts that are involved in our design. We first present the basic notions around autonomic computing along with its main characteristics. Second we give a definition of a cloud environment and an detailed description of dynamic events that occur in a cloud infrastructure. Third we discuss server and network virtualization. Furthermore we provide a description of security threats against traditional information systems and cloud environments. Concepts around security monitoring and security monitoring solutions tailored for cloud environments follow.

2.1 Autonomic Computing

This section presents a brief introduction to autonomic computing. We start with a short historical background while we introduce the basic self-management properties of every autonomous system. Finally, we describe the role of the adaptation manager, a core component that is responsible for the enforcement and realisation of the self-management properties.

2.1.1 What is Autonomic Computing?

The notion of autonomic computing was first introduced by IBM in 2001 [9] in order to describe a system that is able to manage itself based on a set of high-level objectives defined by administrators. Autonomic computing comes as an answer to the increasing complexity of today's large scale distributed systems. As a result the ability of a system's administrator to deploy, configure and maintain such systems is affected. The term autonomic computing carries a biological connotation as it is inspired from the human nervous system and its ability to autonomously control and adapt the human body to its environment without requiring any conscious effort. For example, our nervous system automatically regulates our body temperature and heartbeat rate. Likewise, an autonomic system is able to maintain and adjust it's components to external conditions.

2.1.2 Characteristics

According to [9] the corner stone of each autonomic system is self-management. The system is able to seamlessly monitor its own use and upgrade its components when it

deems necessary requiring no human intervention. The authors identify four main aspects of self-management.

2.1.2.1 Self-configuration

An autonomic system is able to configure its components automatically in accordance with a set of high-level objectives that specify the desired outcome. Seamless integration of new components demands that the system adapts to their presence, similarly to how the human body adapts to the creation of new cells. When a new component is introduced two steps are necessary:

1. Acquiring the necessary knowledge for the system's composition and configuration.
2. Registering itself with the system so that other components can take advantage of its capabilities and modify their behavior accordingly.

2.1.2.2 Self-optimization

One of the main obstacles when deploying complex middleware (e.g. database systems) is the plethora of tunable performance parameters. To this end, self-optimization refers to the ability of the system to continuously monitor and configure its parameters, learn from past experience and take decisions in order to achieve certain high-level objectives.

2.1.2.3 Self-healing

Dealing with components failure in large-scale computer systems often requires devoting a substantial amount of time in debugging and identifying the root cause of a failure. Self-healing refers to the ability of the system to detect, diagnose and repair problems that arise due to software or hardware failures. In the most straightforward example, an autonomous system could detect a failure due to a software bug, download an appropriate patch and then apply it. Another example consists of pro-active measures against externally-caused failures (a redundant power generator in the event of a power outage).

2.1.2.4 Self-protection

Although dedicated technologies that guarantee secure data transfer and network communication (e.g. firewalls, intrusion detection systems) exist, maintenance and configuration of such devices continue to be a demanding error-prone task. Self-protection refers to the ability of the system to defend itself against malicious activities that include external attacks or internal failures.

2.1.3 The Role of the Manager

In every autonomic system the Autonomic Managers (AMs) are software elements responsible for the enforcement of the previously described properties. AMs are responsible for managing hardware or software components that are known as Managed Resources (MRs). An AM can be embedded in a MR or run externally. An AM is able to collect the details it needs from the system, analyze them in order to determine if a change is required, create a sequence of actions (plan) that details the necessary changes and finally, apply those actions. This sequence of automated actions is known as the MAPE [10] control loop. A control loop has four distinct components that continuously share information:

- *Monitor function*: collects, aggregates and filters all information collected from an MR. This information may refer to topology, metrics or configuration properties that can either vary continuously through time or be static.
- *Analyse function*: provides the ability to learn about the environment and determines whether a change is necessary, for example when a policy is being violated.
- *Plan function*: details steps that are required in order to achieve goals and objectives according to defined policies. Once the appropriate plan is generated it is passed to the execute function.
- *Execute function*: schedules and performs the necessary changes to the system.

A representation of the MAPE loop is shown in Figure 2.1.

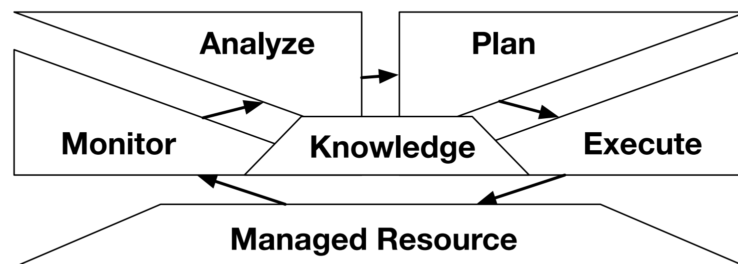


Figure 2.1 – The MAPE control loop

2.2 Cloud Computing

This section briefly introduces the basic notions behind cloud computing, a computing paradigm that extends the ideas of autonomic computing and pairs them with a business model that allows users to provision resources depending on their demands. First the main principles behind cloud computing are outlined. A description of the cloud main characteristics and the available service models follows.

2.2.1 What is Cloud Computing?

Cloud computing emerged as the new paradigm which shifts the location of a computing infrastructure to the network, aiming to reduce hardware and software management costs [11].

The entity that provides users with on-demand resources is known as *service provider*. Many definitions have emerged over the years, however until today no standard definitions exist. In this thesis we rely on the NIST definition presented in [12]:

Definition 1 ***Cloud computing*** *is a model for enabling ubiquitous, 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.*

In order to regulate the terms of providing access to cloud resources, the concept of Service Level Agreement between the provider and the customers was introduced [4]. In the context of cloud computing customers are referred to as tenants.

Definition 2 ***A Service Level Agreement (SLA)*** *is a contract that specifies the service guarantees expected by the tenants, the payment to the provider, and potential penalties when the guarantees are not met.*

2.2.2 Characteristics

According to [12] the main characteristics of cloud computing are: **broad network access**, **on demand self-service**, **resource pooling**, **elasticity** and **measured service**.

- *Broad network access*: Cloud services are usually available through the Internet or a local area network and thus can be accessed from any device with access to the network (e.g. smartphones, tablets, laptops, etc).
- *On-demand self-service*: Tenants can provision resources automatically without the need for a personal negotiation of the terms with the cloud provider. Providers offer dedicated APIs in order to serve this purpose.
- *Resource pooling*: Computing resources can serve multiple tenants simultaneously with different physical and virtual demands adopting a multi-tenant model. In this model, tenants are oblivious about the exact location in which the provisioned resources are located.
- *Elasticity*: Tenants can automatically provision or release new resources depending on computational demand. Theoretically, the resources that a tenant can provision are unlimited.
- *Measured service*: Tenants and the provider can monitor and control resource usage through dedicated mechanisms. The same mechanisms can be used by the tenants in order to check whether the terms defined in the SLA are respected.

2.2.3 Service Models

According to [13] the services that are available in cloud computing are categorized in three models: **Infrastructure as a Service (IaaS)**, **Platform as a Service (PaaS)** and **Software as a Service (SaaS)**. The contributions presented in this thesis were developed on a cloud infrastructure using the IaaS service model.

2.2.3.1 *IaaS*

IaaS offers tenants the capability to provision virtual resources (e.g. processing in the form of virtual machines, storage, networks) without worrying about the underlying physical infrastructure. Although the IaaS cloud model essentially offers the provisioning of a node-based infrastructure, the authors in [14] define two different layers of abstraction in the IaaS cloud model: *Hardware as a Service (HWaaS)* and *Operating System as a Service (OSaaS)*. In HWaaS the tenant is free to install arbitrary software, including the OS, while he is responsible for managing the whole software stack. In HWaaS the provider is only accountable for providing the hardware resources. In OSaaS the tenants are offered a fully managed OS including the underlying hardware resources (essentially the whole environment is perceived as a single compute node). Tenants can deploy their application through the interplay of OS processes. The contributions presented in this thesis target both HWaaS and OSaaS IaaS clouds. Known examples of IaaS HWaaS public clouds include: Amazon Elastic Cloud (EC2) [15], Google Compute Engine [16]

and OVH public cloud [17]. VMware vCloud [18] is a known example of IaaS HWaaS private cloud. Furthermore, a number of open source cloud management systems have been developed over the course of the last few years in order to enable the creation of private clouds (described later in Section 2.2.4). Prominent examples in this category are: Eucalyptus [19], Nimbus [20], OpenNebula [21] and OpenStack [22].

2.2.3.2 *PaaS*

PaaS offers tenants the capability to deploy their own applications as long as they were created using programming languages and libraries supported by the provider. This model allows tenants to focus on application development instead of other time consuming tasks such as managing, deploying and scaling their run-time environment, depending on computational load. Major PaaS systems include Google App Engine [23], Microsoft Azure [24] and Amazon Web Services [15] which are suitable for developing and deploying web applications.

2.2.3.3 *SaaS*

SaaS offers tenants the capability of using the provider's cloud hosted applications through dedicated APIs. The applications are managed and configured by the provider although tenants might have access to limited user-related configuration settings. Prominent examples in this category include: Gmail [25], Google calendar [25] and iCloud [26].

2.2.3.4 Main IaaS Systems

A lot of work in the past was focused on designing and implementing IaaS cloud systems. Tenants are provided with virtualized resources (in the form of Virtual Machines VMs – or containers) and a management system that allows them to manage their resources virtualization technologies like KVM [27], Xen [28] and VMware ESX/ESXi [29] are the building blocks that facilitate server virtualization and efficient resource utilisation. Lately, a trend towards containerization of IaaS cloud systems (e.g. Google Kubernetes [30]) has been observed.

As stated in [31] the core of an IaaS cloud management system is the so called *cloud-OS*. The cloud OS is responsible for managing the provisioning of the virtual resources according to the need of the tenant services that are hosted in the cloud. As an example of a cloud OS, we present OpenStack [32], a mainstream IaaS management system that we used in order to develop our prototype.

OpenStack is an open source cloud management system that allows tenants to provision resources within specific limits set by the cloud administrator. Tenants can view, create and manage their resources either by a dedicated web graphical interface (Horizon) or through command line clients that interact with each one of OpenStack's services. OpenStack operates in a fully centralized manner with one node acting as a controller. The controller accepts user VM life cycle commands and delegates them to a pool of compute nodes. Upon receiving a command from the cloud controller, a compute node enforces it by interacting with the hypervisor. The controller node hosts a plethora of the main services delivered by OpenStack such as: Nova (manager of the VMs lifecycle), Neutron (network connectivity manager), Glance (VM disk image manager) and Keystone (mapping of tenants to services that they can access). Nova and Neutron are also installed on each compute node in order to provide VM interconnectivity and enforce user decision regarding VMs lifecycle. Compute nodes periodically report back to the cloud controller

their available resources (processing, memory, storage) and the state of the deployed VMs (e.g. network connectivity, lifecycle events). OpenStack offers a limited set of integration tools for other public APIs (namely Amazon EC2 and Google Compute Engine). A representation of OpenStack's modular architecture can be found in Figure 2.2.

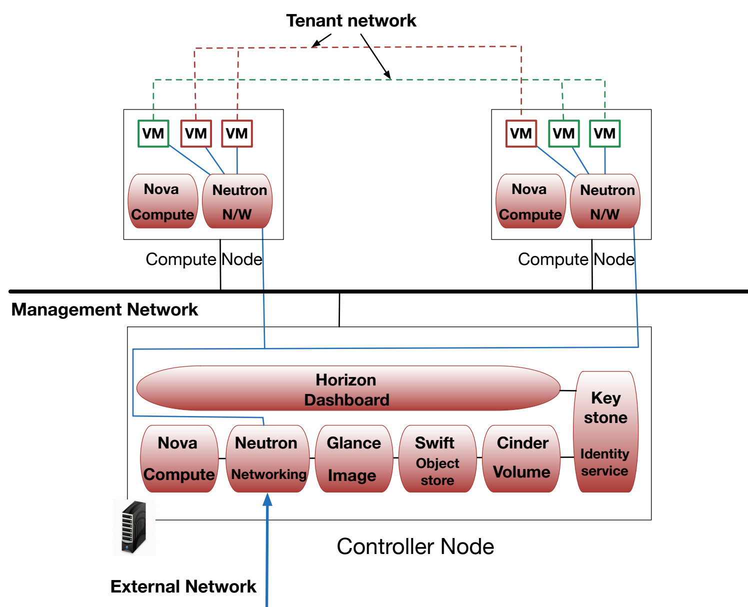


Figure 2.2 – The OpenStack architecture

2.2.4 Deployment Models

There are four distinguishable cloud deployment models: **Private**, **Public**, **Community** and **Hybrid** clouds.

- *Private cloud*: The cloud infrastructure is deployed on compute, storage and network systems that belong to a single organization. A private cloud can be managed either by the organization or a third party entity and its usage does not exceed the scope of the organization.

- *Public cloud*: The cloud infrastructure is available for provisioning for everyone on the Internet. It is typically owned and managed by a cloud provider that allows customers (tenants) to request resources without having to deal with the burden of managing them. As a result tenants are only charged for what they use, in accordance with the pay-as-you-go model.

- *Community cloud*: The cloud infrastructure is dedicated to a specific community or organizations that share a set of policies (i.e. security concerns, mission, and compliance requirements). Community cloud comes as a solution for distributing costs between different organizations in contrast to each organization maintaining its own private cloud (e.g. scientists from different organizations that work on the same project can use the same community cloud). In contrast to public clouds access to community clouds is restricted only to members of the community or organization. They can be managed by one or several organizations of the community. Community clouds can be perceived as a specific category of private clouds.

- *Hybrid cloud*: The cloud infrastructure is a combination of two or more separate cloud infrastructures (private, public or community) that remain individual entities. The entities are bound together by a standardized agreement that allows data and application sharing.

In this thesis we developed a prototype considering a private cloud although the proposed framework can be integrated in both public and community clouds as well.

2.2.5 Dynamic Events in Iaas Clouds and Cloud Adaptation

Cloud environments are based on an elastic, highly scalable model that allows tenants to provision resources (e.g. VMs) with unprecedented ease. Furthermore tenants can choose to deploy different services inside their provisioned VMs and expose them to other users through the Internet, generating network traffic towards and from the cloud infrastructure. As a result, cloud environments become very dynamic, with frequent changes occuring at different levels of the infrastructure. In this section we categorize the observed changes in three categories: service-related, topology-related and traffic-related events.

2.2.5.1 Service-related Events

Service-related dynamic events include all changes in the applications deployed in the VMs of a single tenant. These changes can refer to addition (i.e. installation) of a new application or the removal of an existing one inside an already deployed VM. A reconfiguration of an existing application resulting in additional features is also considered a service-related dynamic event.

2.2.5.2 Topology-related Events

Topology-related events include all changes in the topology of a tenant's virtual infrastructure. The three main commands in a VM life cycle that constitute topology related dynamic events are: *VM creation*, *VM deletion* and *VM migration* (seamlessly moving a VM between two physical nodes over local or wide area network). VM migration can be interpreted as a combination of creation and deletion as when a VM is migrated between two nodes a new copy of the VM is created in the destination node, while the old copy of the VM is deleted from the source node. Public cloud providers offer the possibility of auto-scaling to their tenants in order to automate management of their application's computational load. Scaling decisions generate topology-related changes either by adding new virtual machines (scaling out) or by deleting existing ones when the application's load decreases (scaling in). Network reconfiguration events (e.g. changing a subnet's address range, moving VMs between different subnets or creating/deleting subnets) are also considered topology-related changes.

2.2.5.3 Traffic-related Events

Often tenants deploy network-oriented applications in their cloud infrastructure. Depending on the load of the deployed applications, different levels of network traffic are generated towards and from the virtual infrastructure. Any change in the tenant's virtual infrastructure incoming or outgoing traffic load is considered a traffic-related dynamic event. Public cloud providers offer load-balancing solutions in order to handle the dynamic network load and evenly distribute it to available resources. Load balancing decisions can also lead to topology-related changes when new VMs are started or shutdown.

2.2.5.4 Summary

In this section, we described the three main categories of dynamic events that occur in a cloud infrastructure. The security monitoring framework designed in this thesis addresses the need for reconfiguration of monitoring devices in all three event categories. We now continue with a description of virtualization technologies as the building block that enables cloud-computing.

2.3 Virtualization

This section gives a brief overview of infrastructure virtualization. Infrastructure virtualization can be decomposed in server virtualization and network virtualization. We first present the main server virtualization components followed by the four dominant server virtualization techniques. Finally, this section concludes with a description of network virtualization.

The first ones to define the notion of server virtualization where Popek and Goldberg in their paper "Formal requirements for virtualizable third generation architectures" [33]. According to [33], virtualization is a mechanism permitting the creation of *Virtual Machines* which are essentially *efficient, isolated duplicates of real machines.*

2.3.1 Server Virtualization Components

In an IaaS infrastructure there are three main architectural layers: *physical, hypervisor* and *virtual machine.* We briefly describe each one:

- *Physical*: The physical machine provides the computational resources that are divided between virtual machines (VMs). Computational resources include CPUs, memory and devices (e.g. disk, NIC).
- *Hypervisor*: Originally known as the Virtual Machine Monitor, this component is responsible for mediating the sharing of physical resources (e.g. CPU, memory) between different co-located VMs that operate concurrently. The hypervisor is responsible for ensuring isolation between different VMs providing a dedicated environment for each one without impacting the others.
- *Virtual Machine*: A VM or guest is the workload running on top of the hypervisor. The VM is responsible for executing user applications and virtual appliances. Each VM is under the illusion that it is an autonomous unit with its own dedicated physical resources. The VM is oblivious about the existence of multiple other consolidated VMs on top of the hypervisor of the same physical machine.

The security monitoring framework designed in this thesis targets the virtual machine layer. For extracting key information regarding the services hosted inside the monitored VMs the hypervisor is leveraged.

2.3.2 Server Virtualization

There are different mechanisms that enable the creation of virtual machines each one providing different features. Here we detail the four main ones: *emulation, full virtualization, paravirtualization* and *OS-level virtualization.* The contributions presented in this thesis apply to full virtualization and paravirtualization.

2.3.2.1 Machine-Level Virtualization

2.3.2.1.1 Emulation Emulation is the first proposed technique to allow the system to run a software that mimics a specific set of physical resources. This mechanism was used to enable the usage of console video games on personal desktop machines. In emulation, the assembly code of the guest is translated into host instructions, a technique known as binary translation. A dedicated component, the *emulator* is responsible for performing the translation and providing isolation between different guests. There are two different translation techniques: *static* and *dynamic*. Static binary translation requires translating all of the guest code into host code without executing it. Dynamic binary translation on the other hand offers at runtime emulation where emulators fetch, decode and execute guest instructions in a loop. The main advantage of dynamic binary translation is that since the translation is happening on the fly, it can deal with self-modifying code. Although the performance cost is evident, emulation is very flexible as any hardware can be emulated for a guest's OS. Popular emulators include Bochs [34] and Qemu [35], which support a wide number of guest architectures (x86, x86_64, MIPS, ARM, SPARC).

2.3.2.1.2 Full Virtualization Full system-wide virtualization delivers a virtual machine with dedicated virtual devices, virtual processors and virtual memory. In full virtualization the hypervisor is responsible for providing isolation between VMs as well as multiplexing on the hardware resources. This technique enables running VMs on top of physical hosts without the need to perform any alterations on the VM or the host OS kernel. In [33] the authors formalize the full-virtualization challenge as defining a virtual machine monitor satisfying the following properties:

- Equivalence: The VM should be indistinguishable from the underlying hardware.
- Resource control: The VM should be in complete control of any virtualized resources.
- Efficiency: Most VM instructions should be executed directly on the underlying CPU without involving the hypervisor.

The two methods that make full virtualization possible are: binary translation and hardware acceleration. We discuss both of them.

Binary translation: This technique uses the native OS I/O device support while offering close to native CPU performance by executing as many CPU instructions as possible on bare hardware [36]. When installed, a driver is loaded in the host OS kernel in order to allow it's user space component to gain access to the physical hardware when required. The same driver is responsible for improving network performance for the virtualized guest.Non-virtualized instructions are detected using binary translation and are replaced with new instructions that have the desired effect on the virtualized hardware. The main argument behind virtualization through binary translation is that no modifications of either the guest or the host OS are required. Unfortunately, a non-negligible performance penalty is applied due to the need of performing binary translation and emulation of privileged CPU instructions. Full virtualization with binary translation can be interpreted as a hybrid technique between emulation and virtualization. In contrast to emulation where each CPU instruction is emulated, full virtualization with binary translation allows for some CPU instructions to run directly on the hosts CPU. The most popular fully virtualized solutions using binary translation are: Qemu [35], VirtualBox [37], VMware Fusion and Workstation [38] [39].

Hardware acceleration: In order to cope with the performance overhead introduced by binary translation and enable virtualization of physical hardware, Intel (resp. AMD) came up with the VT-x technology [40] (resp. AMD-V). With VT-x a new root mode of operation is allowed in the CPU. Two new transitions are enabled: from the VMM to a guest a root to non-root transition (called VMEntry) and from the guest to VMM a non-root to root transition (called VMExit). Intel uses a new data structure to store and manage information regarding when these transitions should be triggered, the virtual machine control structure (VMCS). Typically a VMExit occurs when the VM attempts to run a subset of privileged instructions. The VMCS data structure stores all necessary information (instruction name, exit reason). This information is later used by the VMM for executing the privileged instruction. The most popular solutions using hardware assisted virtualization are: KVM [27], VMware ESXi [29], Microsoft Hyper-V [41] and Xen Hardware Virtual Machine [42].

2.3.2.1.3 Paravirtualization In contrast to full virtualization which advocates for no modifications in the guest OS, paravirtualization requires the guest OS kernel to be modified in order to replace non-virtualized instructions with *hypercalls* that communicate directly with the hypervisor. The hypervisor is responsible for exporting *hypercall* interfaces for other sensitive kernel operations such as memory management and interrupt handling. Xen Project [28] has been the most prominent paravirualization solution. In Xen the processor and memory are virtualised using a modified Linux kernel. The modified kernel is actually an administrative VM (called dom0) responsible for providing isolation between VMs, handling network, I/O and memory management for the guest VMs (domU). Dom0 is also in control of the guest VMs lifecycle and bares the responsibility for executing privileged instructions on behalf of the guest OS. The later is done by issuing *hypercalls*. Dom0 traps the latter and executes them either by translating them to native hardware instructions or using emulation. Xen operates based on a *split driver* model where the actual device drivers, called backend drivers, are located inside Dom0 and each DomU implements an emulated device, called frontend driver. Every time a DomU issues a call to a driver the emulated part transfers the call to the actual driver in Dom0 – hence the two drivers complementary operate as one. Although Xen is a promising solution for near native performance, its application is limited to open source OSes like Linux or proprietary solutions which offer a customized Xen-compatible version.

2.3.2.1.4 Hypervisor Practices Emulation, full virtualization and paravirtualization can be combined. Typically, devices are fully emulated (for maintaining the use of legacy drivers) or paravirtulized (for efficient multiplexing access on these devices from different VMs) while the CPU is fully virtualized. Modern hypervisors that adopt this technique are: KVM [27], Xen [28] and VMware Workstation [39].

2.3.2.2 OS-level Virtualization

Another solution, known as lightweight or OS-level virtualization [43], allows the OS kernel to perform virtualization at the system call interface, and create isolated environments that share the same kernel. These flexible, user-oriented isolated environments are known as containers. Containers have their own resources (e.g. file system, network connectivity, firewall, users, applications) that are managed by the shared OS kernel (responsible for providing isolation). Since they all share the same kernel the performance overhead is minimal to none. Furthermore, a container can be migrated in the same way as a VM.

Unfortunately, the main issue behind OS-level virtualization is that all containers in a single physical machine are limited to the kernel of the host OS. This limits the number of OSes to only the ones supported by the host's kernel. LXC [44] and Docker [45] are some of the most prominent solutions in this category.

2.3.3 Network Virtualization and Network Management in IaaS Clouds

Network virtualization is one of the key aspects in an IaaS cloud environment. Assigning IP addresses to VMs, communication between VMs that belong to the same or different tenants and finally communication between VMs and the outside world are some of the issues that need to be addressed from the network virtualization component of the IaaS cloud management system. In this section we first present the mechanisms that materialize network virtualization and we continue with a discussion about network management in IaaS clouds focusing on OpenStack.

2.3.3.1 Network Virtualization

There are different solutions that enable network virtualization. Multi-protocol Label Switching [46] uses a "label" appended to a packet in order to transport data instead of using addresses. MPLS allows switches and other network devices to route packets based on a simplified label (as opposed to a long IP address). *Hard VLANs* allow a single physical network to be broken to multiple segments. By grouping hosts that are likely to communicate with each other to the same VLAN, one can reduce the amount of traffic that needs to be routed. Flat networking relies on the ethernet adapter of each compute node (which is configured as a bridge) in order to communicate with other hosts. With VLAN tagging each packet belonging to a specific VLAN is assigned the same VLAN ID while with GRE encapsulation traffic is encapsulated with a unique tunnel ID per network (the tunnel ID is used in order to differentiate between networks). VLAN tagging and GRE encapsulation both require a virtual switch in order to perform the tagging (respectively encapsulation) while flat networking does not require a virtual switch.

However these solutions lack a single unifying abstraction that can be leveraged to configure the network in a global manner. A solution to this empedement that provides dynamic centrally-controlled network management is software defined networking (SDN) [47]. In this section we mainly focus on SDN.

Software defined networking [47] emerged as a paradigm in an effort to break the vertical integration of the control and the data plane in a network. It separates a network's control logic from the underlying physical routers and switches which are now simple forwarding devices. The control logic is implemented in a centralized controller allowing for a more simplified policy enforcement and network reconfiguration. Although SDNs are logically centralized, the need for a scalable, reliable solution that guarantees adequate performance does not allow for a physically centralized approach. The separation between the control and the data plane is feasible by creating a strictly defined programmable interface (API) between the switches and the SDN controller. The most notable example of such API is OpenFlow [48]. In each OpenFlow switch flow tables of packet-handling rules are stored. Each rule matches a subset of the traffic and performs certain actions (dropping, forwarding, modifying) on the matched subset. The rules are installed on the switches by the controller and depending on their content a switch can behave like a router, switch, firewall or in general a middlebox. A switch can communicate with the controller through a secure channel using the OpenFlow protocol which defines the set of messages that can be exchanged between these two entities. Traffic handling rules can be installed

on the switches either proactively or reactively, when a packet arrives. A representation of the SDN architecture can be found in Figure2.3.

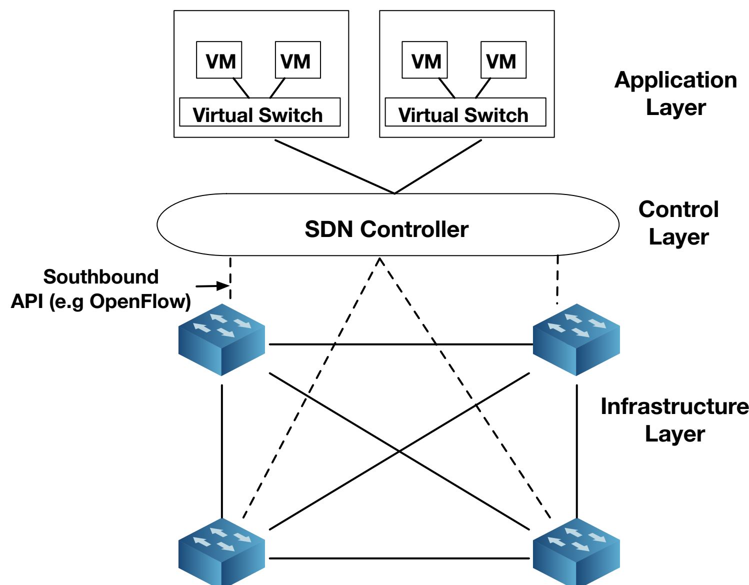


Figure 2.3 – The SDN architecture

Although OpenFlow is the most widely accepted and deployed API for SDNs there are several other solutions such as ForCES [49] and POF [50]. The controller provides a programmatic interface to the network that can be used to execute management tasks as well as offer new functionalities. It essentially enables the SDN model to be applied on a wide range of hardware devices (e.g. wireless, wired). A wide range of available controllers exist such as Nox [51], OpenDaylight [52] and Floodlight [53].

Making network virtualization a consolidated technology requires multiple logical networks to be able to share the same OpenFlow networking infrastructure. FlowVisor [54] was one of the early solutions towards that direction. It enables slicing a data plane based on off-the-shelf OpenFlow compatible switches, making the coexistence of multiple networks possible. The authors propose five slicing dimensions: bandwidth, topology, traffic, forwarding tables and device CPU. Each slice can have its own controller allowing multiple controllers to inhabit the same physical infrastructure. Each controller can only operate on its own slice and gets its own flow tables in the switches.

FlowN [55] offers a solution analogous to container virtualization (i.e. a lightweight virtualization approach). In contrast with FlowVisor [54], it deploys a unique shared controller platform that can be used to manage multiple domains in a cloud environment. A single shared controller platform enables management of different network domains. It offers complete control over a virtual network to each tenant and it allows them to develop any application on top of the shared controller.

Network virtualization platform (NVP) from VMware (as part of the NSX [56] product) provides the necessary abstractions for the creation of independent networks (each one with different service model and topology). No knowledge about the underlying network topology or state of the forwarding devices is required as tenants simply provide their desired network configuration (e.g. addressing architecture). NVP is responsible for translating tenant requirements to low-level instruction sets that are later on installed on the forwarding devices. A cluster of SDN controllers is used in order to modify the

flow tables on the switches. NVP was designed to address challenges in large-scale multi-tenant environments that are not supported by the previously described solutions (e.g. migrating an information system to the cloud without the need of modifying the network configuration). A similar solution is SDN VE [57] from IBM based on OpenDaylight.

2.3.3.2 Network Management in Iaas Clouds

Network virtualization delivers compute-related options (create, delete) to network management. Network objects (networks, subnets, ports, routers, etc) can be created, deleted and reconfigured programmatically without the need of reconfiguring the underlying hardware infrastructure. The underlying hardware infrastructure is treated as a pool of transport resources that can be consumed on demand. Tenants can create private networks (i.e. tenant networks) and choose their own IP address scheme, which can overlap with IP addresses chosen by other tenants. Depending on the type of the tenant network (flat, VLAN, GRE) different communication capabilities are offered to the instances attached to these networks.

The networking component of an IaaS cloud management system is responsible for mapping tenant-defined network concepts to existing physical networks in a data center. Essentially the network component performs the following functionalities: assign IP addresses to VMs, facilitating communication between VMs that belong to the same or different tenants and finally providing VMs with outside-world connectivity.

In OpenStack, Neutron is responsible for managing different tenant networks and offering a full set of networking services (routing, switching, load-balancing, etc) to provisioned VMs. Neutron is composed of agents (e.g. DHCP agent, L3 routing agent, etc) that provide different types of networking services to provisioned VMs. Neutron creates three different networks in a standard cloud deployment:

1. *Management network:* used for communication between the OpenStack components. This network is only reached from within the datacenter.

2. *Tenant networks:* used for communication between VMs in the cloud. The configuration of these networks depends on the networking choices made by the different tenants.

3. *External network:* used to provide internet connectivity to VMs hosted in the cloud.

On each compute node a virtual bridge is created by a dedicated Neutron plugin (called ML2 plugin) which is locally installed on each node. VMs are connected to networks through virtual ports on the ML2-created bridge. The ML2 plugin is also responsible for segregating network traffic between VMs on a per tenant basis. This can be achieved either through VLAN tagging (all VMs that belong to the same network are assigned the same tag) or GRE encapsulation.

2.4 Security Threats

In this section we detail some of the known attacks against information systems and cloud environments.

2.4.1 Security Threats in Information Systems

Although one of the most common ways of executing cyber attacks is through the network (i.e. either the Internet or a local area network), the attackers often target different areas

in an information system. Here we list the most common threats depending on their target level. Before presenting each threat category in detail, we present a high level overview of the vulnerability classes that attackers can exploit. In general, missing validation of inputs in an application can create an entry point for attacks (listed below). Furthermore, lack of access control (i.e. through authentication mechanisms) can allow an attacker to gain unauthorized privileged access.

2.4.1.1 Application Level

Application-level threats are abilities of an attacker to exploit vulnerabilities in the software of one or more applications running in an information system.

One of the most common application-level attack is SQL injection [58] against Database Management Systems (DBMS). An SQL injection attack occurs when a malicious entity on the client side manages to insert an SQL query via input data to the application. This is usually possible due to lacks of input validation. The impact of the injection may vary depending on the skills and imagination of the attacker. Usually, through an SQL exploit the attacker can gain access to sensitive data inside the database, modify them (insert or delete or update) or even retrieve the contents of a file present in the system. He can also shutdown the DBMS by issuing administrative commands and sometimes even execute commands outside the DBMS.

Another type of an injection attack is cross-site scripting (XSS) [59] when the attacker manages to insert malicious code in a trusted website. Cross-site scripting exploits the absence of validation of user input. The malicious code could be in the form of a JavaScript segment or any other code that the browser can execute. When a different user accesses this website she will execute the script thinking that it comes from a trusted source, giving the attacker access to cookies, session tokens or other sensitive information retrieved by the browser on behalf of the infected website. In a more severe scenario the attacker might even redirect the end user to web content under his control. An XSS attack can either be stored (the malicious script permanently resides on the target server) or reflected (the script is reflected off the web server – for example in an error message).

A buffer overflow [60] generally occurs when an application attempts to store data in a buffer and the stored data exceeds the buffer's limits. Buffer overflows are possible because of badly validated input on the application's side. Writing in an unauthorized part of the memory might lead to corrupted data, application crashes or even malicious code execution. Buffer overflows are often used as entry points for the attacker in order to inject malicious code segment into the host's memory and then execute it by jumping to the right memory address. Another alternative for malicious code injection is format string attacks [61]. Format String Attacks (FSA) are used in order to leak information such as pointer addresses. After a successful FSA, normally a return oriented programming exploit is used. Return oriented programming allows the attacker to use short sequences of instructions that already exist in a target program in order to introduce arbitrary behavior.

2.4.1.2 Network Level

In the network-level threat category we describe attacks that target communications of layer 3 and above in an information system.

Network-level impersonation occurs when an attacker masks his true identity or tries to impersonate another computer in network communications. Operating systems use the IP address of a packet to validate its source. An attacker can create an IP packet with a header that contains a false sender's address, a technique known as IP spoofing [62].

This technique, combined with TCP protocol specifications (i.e. the sequence and acknowledgement numbers included in a TCP header) can lead to session hijacking. The attacker predicts the sequence number and creates a false session with the victim who in turn thinks that he is communicating with the legitimate host.

Denial of Service (DoS) attacks aim at exhausting the computing and network resources of an information system. The way these attacks operate is by sending a victim a stream of packets that swamps its network or processing capacity, denying access to its normal clients. One of the methods employed is SYN flooding [63], in which the attacker sends client requests to the victim's server. SYN flooding attacks target hosts that run TCP server processes and exploit the state retention of the TCP protocol after a SYN packet has been received. Their goal is to overload the server with half-established connections and disturb normal operations. Both IP spoofing and SYN flooding are common techniques for launching a denial of service attack and preventing users from accessing a network service. In the event of a server being protected against SYN flood attacks, a denial of service can still be possible if the server in question is too slow in serving flooding requests (the attacker simply overloads the server). Finally, as the name indicates, a man-in-the-middle attack refers to a state where the attacker is able to actively monitor, capture and control the network packets exchanged between two communicating entities. Sophisticated versions of man-in-the-middle attacks include attempts against TLS-based communications where the attackers are able to falsely impersonate legitimate users [64].

Domain Name Servers (DNS) are essential parts of the network infrastructure that map domain names to IP addresses redirecting requests to the appropriate location. Attackers target DNS systems in their effort to redirect legitimate requests to malicious websites under their control. One of the most common techniques to achieve that is DNS cache poisoning. DNS cache poisoning exploits a vulnerability in the DNS protocol [65] in order to replace legitimate resolution results with flawed ones that include the attacker's website.

Depending on the type of services hosted in an information system attackers use different exploitation techniques. Identification of the type of hosted services is a necessary preliminary step in most exploitation attempts. A common way for an attacker to identify network services hosted in an information system or probe a specific server for open ports, is port scanning [66]. The standard way to perform a port scan is to launch a process that sends client requests to a range of ports in a particular server (vertical port scan) or to a specific port on several hosts (horizontal port scan). Depending on the type of the request there are different port scan categories at the TCP level. Application fingerprinting, where an attacker looks for a reply that matches a particular vulnerable version of an application is also a common technique used to identify the type of hosted service.

2.4.1.3 Operating System Level

All user applications in an information system rely on the integrity of the kernel and core system utilities. Therefore, a possible compromise of any of these two parts can result in a complete lack of trust in the system as a whole. One of the most common attacks against a system's kernel is a rootkit installation. Rootkits are pieces of software that allow attackers to modify a host's software, usually causing it to hide their presence from the host's legitimate administrators. A sophisticated rootkit is often able to alter the kernel's functionality so that no user applications that run in the infected system can be trusted to produce accurate results (including rootkit detectors). Rootkits usually come with a dedicated backdoor so the attacker can gain and maintain access to the compromised host. Backdoors usually create secure SSH connections such that the communication between

the attacker and the compromised machine cannot be analysed by Intrusion Detection Systems or other network monitoring tools.

2.4.1.4 Summary

In this section we have described security threats targeting traditional information systems. The described attacks could also target applications running inside virtual machines in an outsourced infrastructure. We continue with a description of cloud specific security threats and a classification based on their target.

2.4.2 Security Threats in Cloud Environments

In a cloud environment security concerns two different actors. First, tenants are concerned with the security of their outsourced assets, especially if they are exposed to the Internet. Second, the provider is also concerned about the security of the underlying infrastructure especially when he has no insight regarding the hosted applications and their workload. In this section we focus on security threats originating from corrupted tenants against other legitimate tenants and their resources, threats against the provider's infrastructure and their origin as well as threats towards the provider's API.

2.4.2.1 Threats against tenants and based on shared resources

One of the key elements of a cloud infrastructure is multi-tenancy (i.e. multiplexing virtual machines that belong to different tenants on the same physical hardware). Although this maximizes efficiency for the cloud provider's resources, it also offers the possibility that a tenant's VM can be located in the same physical machine as a malicious VM. This in turn engenders a new threat: breaking the resource isolation provided by the hypervisor and the hardware and gaining access to unauthorized data or disturbing the operation of legitimate VMs.

One of the most prominent attacks that illustrates this threat is the side channel attack where an adversary with a colocated VM gains access to information belonging to other VMs (e.g. passwords, cryptographic keys). In [67] the attackers used shared CPU caches as side channels in order to extract sensitive infomation from a colocated VM.

Another technique that exploits VM colocation is DoS attacks against shared resources. A malicious VM is excessively consuming shared computing resources (CPU time, memory, I/O bandwidth) disallowing legitimate VMs from completing their tasks.

2.4.2.2 Provider Infrastructure

In an IaaS cloud environment 2.2.3.1 each VM is under the illusion that it runs on its own hardware (i.e. CPU, memory, NIC, storage). This illusion is created by the hypervisor, which is responsible for allocating resources for each VM, handling sensitive instructions issued by VMs and finally managing VM lifecycle 2.3. In this section we discuss security threats targeting the hypervisor, as a core component of the provider's infrastructure.

An attacker targeting the hypervisor might be able to execute malware from different runtime spaces inside the cloud infrastructure. Each runtime space comes with different privileges. We list the runtime spaces in increasing order of privilege level (also the order in difficulty to exploit).

- Guest VM User-Space: This runtime space is the easiest one to obtain especially in an IaaS environment. Although, attempts to run privileged instructions could

lead to an exception, an attacker can run any type of exploit. In [68] the attacker manages to break out from a guest by exploiting a missing check in the QEMU-KVM user-space driver.

- Guest VM Kernel-Space: Since in an IaaS cloud environment tenants can run an OS of their choice, an attacker can provision some VMs, run an already tampered OS and use the malicious guest's kernel to launch an attack to the hypervisor. In [69] an attack to the hypervisor implements a malicious para-virtualized front-end driver and exploits a vulnerability in the back-end driver.
- Hypervisor Host OS: One of the most desirable runtime spaces for an attacker is the one of the host OS as the privileges granted are very high. For example, KVM as a part of the Linux kernel, provides an entry point for attackers that have local user access to the host machine, exploiting a flaw in KVM.

Customers in public clouds manage their resources through dedicated web control interfaces. Moreover, cloud providers also manage the operation of the cloud system through dedicated interfaces that are often accessible through the Internet. A successful attack on a control interface could grant the attacker complete access to a victim's account along with all the data stored in it, or even worse to the whole cloud infrastructure when the provider's interface is compromised. In [70] attacks towards cloud management interfaces are considered extremely high risk and in [71] the authors prove that the web interfaces of two known public and private cloud systems (Amazon's EC2 and Eucalyptus) are susceptible to signature wrapping attacks. In a signature wrapping attack, the attacker can modify a message signed by a legitimate signature, and trick the web service into processing its message as if it was legitimate.

2.4.3 Summary

In summary, traditional information systems as well as cloud environments face multiple security threats originating from different privilege levels in the infrastructure. In an IaaS cloud environment the attack surface is expanded with the addition of the hypervisor, as the building block of a cloud infrastructure, as well as the web-exposed management API. In order to successfully detect attacks a security monitoring framework is needed. We continue our discussion with a detailed description of security monitoring frameworks both for traditional information systems and cloud environments.

2.5 Security Monitoring

Information systems face continuous threats at different levels of their infrastructure. An attacker can gain access to the system by exploiting a software vulnerability and thus be able to modify both the OS kernel and critical system utilities. In order to detect such activities, a security monitoring framework is necessary. A security monitoring framework consists of the appropriate detection mechanisms required to diagnose when an information system has been compromised and inform the administrator in the form of specialised messages (called alerts).

2.5.1 What is Security Monitoring?

According to [72]:

Definition 3 ***Security Monitoring*** *is the collection, analysis, and escalation of indications and warnings to detect and respond to intrusions.*

Due to the diverse nature of applications hosted in an information system, a security monitoring framework requires multiple components that monitor different parts of the system in order to maintain situational awareness of all hosted applications. Figure 2.4 depicts an information system with different security devices: firewalls, antiviruses, network-based IDS.

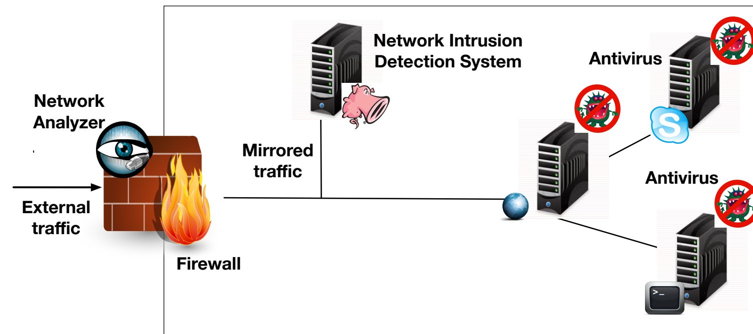


Figure 2.4 – Information system with different security devices contributing to security monitoring

In the following sections we discuss some of the core components embedded in modern security monitoring frameworks focusing primarily on Intrusion Detection Systems (IDS) and Firewalls as the contributions presented in this thesis focus on these two components.

2.5.1.1 Main Components

Several tools are available for mitigating malware threats in an information system. We list the most common ones along with their typical features.

- **Antivirus:** Most antivirus solutions provide capabilities such as scanning critical system components (startup files, boot records), real-time scanning of files that are downloaded, opened or executed, sandboxed dynamic monitoring of running applications and identifying common types of malware (viruses, worms, backdoors, etc). Commercial solutions include Kaspersky Security Scan [73], AVG antivirus [74] and Panda Protection [75].

- **Router:** Typically a router uses a set of traffic management rules that is known as an *access control list*. Routers are normally deployed in front and at the core of an information system's firewall and perform some traffic mitigation such as ingress and egress filtering. Commercial solutions include: Cisco ASR 1000 Series [76] and Juniper MX Series [77].

- **Access Control Systems:** Normally access control systems are concerned with regulating the users attempts to access specific resources in an information system. Information systems apply access controls at different levels of their infrastructure (e.g. an OS regulating access to files, or an authentication server described below).

- **Virtual Private Network (VPN):** VPN allows users to access an organization's private network remotely. It offers traffic encryption between two connection points

through a variety of supported protocols (TLS [78], IPsec [79], DTLS [80]). Examples of VPN service providers include: VPN Express[81] and VyPR VPN [82]

- **Authentication Server:** A server that is used to authenticate users or applications through the network using a set of credentials (e.g. username and password). Authentication servers support a variety of protocols. Notable example of this category is the LDAP protocol [83] and Kerberos [84] protocol.

- **Log Collectors:** In order to facilitate the collection and analysis of events of interest, log collectors are necessary for every information system. Depending on the level of desired system-wide visibility, different time intervals for the collection of logs can be defined. Due to high diversity between event sources (i.e. applications, system, security devices) most software solutions are able to gather and unify information from different sources and different formats. In a cloud environment log collection is of critical importance as it allows tenants to gain insight into resources utilization, application performance, security and operational health. Major public clouds offer customisable logging services such as CloudWatch [85] and Log Analytics [86]. In traditional information systems a variety of log collection solutions exists (e.g. rsyslog [87], LogStash [88]).

2.5.1.2 Intrusion Detection Systems

Intrusion detection systems are usually at the core of a security monitoring framework. Their main purpose is to detect security breaches in an information system before they inflict widespread damage. An IDS is composed of three main stages: data collection, processing and reporting. The core detection feature is implemented in the processing stage.

2.5.1.3 What is an IDS?

According to [89]:

Definition 4 ***Intrusion detection*** *is the process of monitoring the events occurring in a computer system or network and analyzing them for signs of possible incidents, which are violations or imminent threats of violation of computer security policies, acceptable use policies, or standard security practices.*

Consequently, an Intrusion Detection System is software that automates the intrusion detection process.

In the following section we describe the different types of IDSs according to their detection technique and we follow with a classification based on the embedded technology.

2.5.1.4 Types of IDSs

Most of the IDS technologies are based on one of the two following detection techniques [90], [91]. We describe each one along with observed advantages and pitfalls.

- **Signature-based**: Signature-based IDSs compare observed events against a list of a priori known signatures in order to identify possible security breaches. A signature is generally a pattern that corresponds to a registered attack. Signature-based detection is very effective at identifying known threats and is the simplest form of

detection. A signature-based IDS compares the current unit of activity (e.g. network packet, file content) to the list of signatures using string comparison. Unfortunately, signature-based detection is largely ineffective when dealing with previously unknown attacks, attacks that are composed by multiple events or attacks that use evasion techniques [92]. Known examples of signature based IDSs include Snort [93], Suricata [94] and Sagan [95].

- **Anomaly-based:** Anomaly-based IDSs compare a profile of activity that has been established as normal with observed events and attempt to identify significant deviations. Each deviation is considered an anomaly. A normal profile is created by observing the monitored system for a period of time-called *training period* (e.g. for a given network http activity composes 15% of the observed traffic) and can be static (the profile remains unchanged) or dynamic (the profile is updated at specific time intervals). Depending on the methodology used to create the normal profile anomaly-based IDSs are either statistical-, knowledge- or machine-learning-based [92]. Statistical-based IDSs represent the behavior of the analysed system from a random view point, while knowledge-based IDSs try to capture the system's behavior based on system data. Finally machine learning-based IDSs establish a model that allows for pattern categorization.

  One of the main advantages of an anomaly-based IDS is that it can be very effective when dealing with previously unknown attacks. Unfortunately, anomaly-based IDSs suffer from many false positives when benign activity, that deviates significantly from the normal profile, is considered an anomaly. This phenomenon is amplified when the monitored information system is very dynamic. Known examples of anomaly-based IDSs include Bro [96], Stealthwatch [97] and Cisco NGIPS [98].

According to [99] IDS technologies are divided in two categories depending on the type of events that they monitor and the ways in which they are deployed:

- **Network-based (NIDS):** NIDSs monitor network traffic (i.e. packets) for a particular network or segments of a network and analyze network protocol activity or packet payload in order to detect suspicious events or attacks. The most common approach for deploying an NIDS is at a boundary between networks, in proximity to border firewalls or other security devices. A specific category of NIDS is wireless IDSs, which monitor only wireless network traffic and analyze wireless network protocols for identifying suspicious activity. In contrast to other NIDS which focus on packet payload analysis, wireless NIDSs focus on anomalies in wireless protocols.

- **Host-based (HIDS):** HIDSs monitor the events occurring in a single host for suspicious activity. An HIDS can monitor network traffic, system logs, application activity, process list and file access in a particular host. HIDSs are typically deployed on critical hosts that contain sensitive information.

We have described the main IDS categories based on the mechanism used for detection and the way they are deployed. The work done in this thesis focuses on network-based IDSs. We now discuss the second main security component that has been addressed in this thesis: the firewall.

2.5.1.5 Firewalls

This section focuses on a different security component, the firewall.

2.5.1.5.1 What is a Firewall? According to [100]:

Definition 5 *A **firewall** is a collection of components, interposed between two networks, that filters traffic between them according to some security policy.*

Firewalls are devices that provide the most obvious mechanism for enforcing network security policies. When deploying legacy applications and networks, firewalls are excellent in providing a first-level barrier to potential intruders. The most common firewall configuration comprises two packet filtering routers that create a restricted-access network (called Demilitarized Zone or DMZ, see Figure 2.5).

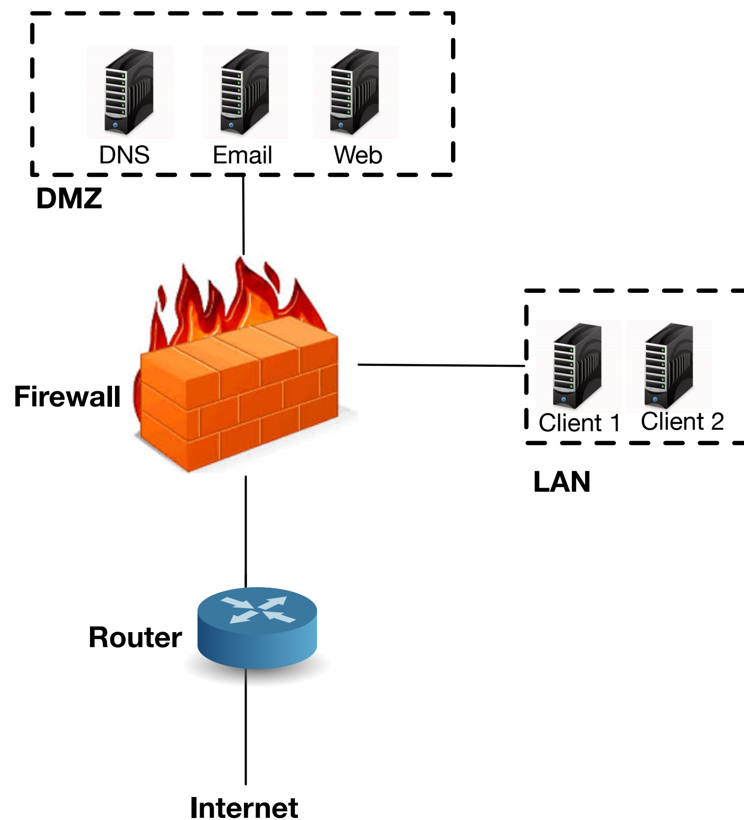


Figure 2.5 – A DMZ example

According to [101] firewalls have three main goals:

1. Protect hosts inside the DMZ from outside attacks,
2. Allow traffic from the outside world to reach hosts inside the DMZ in order to provide network services,
3. Enforce organizational security policies that might include restrictions that are not strictly security related (e.g. access to specific websites).

2.5.1.5.2 Firewall Features We now discuss available firewall features and the capabilities of each one as in [102].

- **Packet filtering:** The most basic feature of a firewall is the filtering of packets. When we refer to packet filtering we are not concerned with the payload of the

network packets but with the information stored in their headers. The mechanism of packet filtering is controlled by a set of directives which is known as *ruleset*. The simplest form of a packet filtering device is a network router equipped with access control lists.

- **Stateful inspection:** This functionality essentially improves packet filtering by maintaining a table of connections state and blocking packets that deviate from the expected state according to a given protocol.
- **Application-level:** In order to extend and improve stateful inspection, *stateful protocol analysis* was created. With this mechanism a basic intrusion detection engine is added in order to analyse protocols at the application layer. The IDS engine compares observed traffic with vendor-created benign profiles and is able to allow or deny access based on how an application is behaving over the network.
- **Application-proxy gateways:** A firewall which acts as an application-proxy gateway contains a proxy agent that acts as an intermediary between different hosts that want to communicate with each other. If the communication is allowed then two separate connections are created (client-to-proxy and proxy-to-server) while the proxy remains transparent to both hosts. Much like an application-level firewall the proxy can inspect and filter the content of traffic.
- **Virtual private networking (VPN):** A common requirement for firewalls is to encrypt and decrypt network traffic between the protected network (DMZ) and the outside world. This is done by adding a VPN functionality to the firewall. As with other advanced firewall functionalities (besides simple header-based packet filtering) a trade-off between the functionality and the cost in terms of computational resources (CPU, memory) depending on the traffic volume and the type of requested encryption, is introduced.
- **Network access control:** Another functionality for modern firewalls is controlling incoming connections based on the result of health checks performed on the computer of a remote user. This requires an agent that is controlled by the firewall to be installed in the user's machine. This mechanism is typically used for authenticating users before granting them access to the network.
- **Unified threat management:** The combination of multiple features into a single firewall is done with the purpose of merging multiple security objectives into a single system. This usually involves offering malware detection and eradication, suspicious probe identification and blocking along with traditional firewall capabilities. Unfortunately, the system's requirements in terms of memory and CPU are significantly increased.

In this thesis, we address application-level firewalls and firewalls that provide stateful traffic inspection capabilities.

2.5.2 Security Monitoring in Cloud Environments

After presenting different components of a security monitoring framework we zoom in security monitoring frameworks tailored for cloud environments. As explained in Section 2.2.5 cloud environments experience dynamic events in different levels of the infrastructure. Naturally, the occurring events engender changes for the security monitoring framework

that requires its components to be automatically adapted to the new state. For example, when a VM is migrated from one compute node to another, the NIDS that is responsible for monitoring the traffic on the destination compute node needs to be reconfigured in order to monitor the traffic for specific attacks against the services hosted in the migrated VM. Without reconfiguration, an attack could pass undetected, creating an entry point for the attacker and allowing him to compromise the cloud-hosted information system.

In this section we discuss cloud security monitoring solutions targeting either to the provider infrastructure and its most critical components (e.g. hypervisor, host OS kernel) or to the tenant information system.

2.5.2.1 Provider Infrastructure Monitoring

This section presents security solutions that target the cloud provider's infrastructure focusing on the hypervisor and host OS kernel. The frameworks described in both categories could be considered as hypervisor or kernel IDS systems. The relationship between the hypervisor and the host OS kernel is depicted in Figure 2.6. In this picture the hypervisor runs as a kernel module while the VMs run in user space. Their virtual address space is mapped through the hypervisor to the host's physical address space.

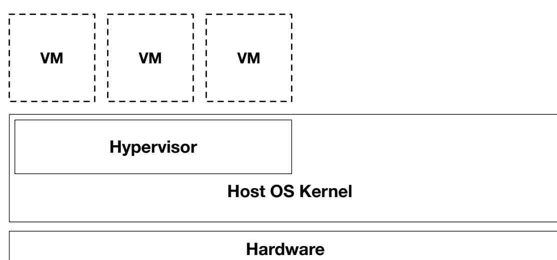


Figure 2.6 – Hypervisor and host OS kernel

2.5.2.1.1 Hypervisor Integrity Checking Frameworks One of the trends in securing the hypervisor involves reducing the Trusted Code Base (TCB) [103] in many of the commercial hypervisors. Although, such a solution would limit the attack surface it would not completely guarantee the integrity of all hypervisor components. To address this challenge, the authors in [104] have created HyperCheck, a hardware assisted intrusion detection framework, that aims at hardening the TCB. Their framework uses the CPU System Management Mode (SMM, a built-in feature in all x86 models) for taking a snapshot of the current state of the CPU and memory and transmits it to a secure remote server for analysis. The remote server is capable of determining whether the analysed hypervisor has been compromised by comparing the newly received snapshot with one taken when the machine was initialized. HyperCheck operates in the BIOS level thus its only requirement is that the attacker does not gain physical access to the machine for altering the SMM during runtime. In order to secure HyperCheck against attacks that simulate hardware resets, a machine with a trusted boot can be used. The authors of HyperCheck implemented a prototype on QEMU which is able to create and send a snapshot of the protected system in approximately 40ms.

In a similar approach the authors of HyperSentry [105] also used a snapshot taken by the SMM to perform integrity checking. The fundamental difference between these two frameworks is that in the case of HyperSentry the request for the snapshot is issued by a stealthy out-of-band channel, typically the Intelligent Platform Management Interface,

that is out of the control of the CPU. Thus an attacker who has obtained the highest level of privileges in the system still cannot call HyperSentry. Performance wise, a periodic invocation of HyperSentry (integrity checks every 16 seconds) would result in an 1.3 % of overhead for the hypervisor, while a full snapshot requires 35ms.

In contrast to the aforementioned hardware assisted solutions, HyperSafe [106] is a software solution that is centered around enforcing hypervisor integrity rather than verifying it. The authors use two software techniques, called *non-bypassable memory lockdown* and *restricted pointer indexing*, to guarantee integrity of the hypervisor's code in addition to control flow integrity. Non bypassable memory lockdown write-protects the memory pages that include the hypervisor code along with their attributes so that a change during runtime is prevented. By leveraging non-bypassable memory lockdown the framework is able to expand write-protection to control data. In order to deal with the dynamic nature of control data (like stack return addresses) the authors compute a control graph and restrict the control data to conform with the results of the graph. The induced overhead by running HyperSafe for tenant applications is less than 5%.

2.5.2.1.2 Kernel Protection Frameworks In contrast to hypervisor integrity frameworks which are only concerned with protecting the code base and data of the hypervisor, kernel protection frameworks aim at securing the code integrity of the kernel. The frameworks described below provide tampering detection for rootkits, a subcategory of malware.

One of the most pivotal works in kernel integrity checking is Copilot [107]. Copilot is able to access a system's memory without relying on the kernel and without modifying the OS of the host. The framework is based on a special PCI add-on card that is able to check the monitored kernel for malicious modifications periodically. The host's memory is retrieved through DMA techniques and sent to a remote admin station for inspection through a dedicated secure communication channel (much like the HyperCheck approach). With an inspection window of 30 seconds Copilot's overhead to system performance is approximately 1%.

HookSafe [108] follows the same philosophy as HyperSafe by write-protecting kernel hooks in order to guarantee control data integrity. The authors base their approach on the observation that kernel hooks rarely change their value once initialised, thus making it possible to relocate them in a page-aligned memory space with regulated access. The performance overhead to real-world applications (e.g. Apache web server) is 6%.

Gibraltal [109], installed in a dedicated machine called the *observer*, obtains a snapshot of the kernel's memory through a PCI card. It observes kernel execution over a certain period of time (training phase) and creates hypothetic invariants about key kernel data structures. An example of an invariant could be that "the values of elements of the system call table are constant". Gibraltal then periodically checks whether the invariants are violated and if so an administrator is notified for the presence of a rootkit. The framework produces a very low false positive rate (0.65%) while maintaining a low performance overhead (less than 0.5%).

A common observation for the frameworks described in both kernel and hypervisor intrusion detection solutions is that the incorporated detection mechanism cannot be adapted depending on changes on the applications hosted in the monitored system (virtualised or not). Furthermore, in the case of hypervisor integrity frameworks, the solutions do not address changes in the virtual topology or the load of network traffic.

After describing the different approaches to secure the hypervisor, one of the most critical parts of the provider's infrastructure, we now shift our focus to security monitoring frameworks for tenant infrastructures.

2.5.2.2 Tenant Information System Monitoring

In this section we focus on tenant security monitoring frameworks with two main components: intrusion detection systems and firewalls. Before that we present an important concept in tenant infrastructure monitoring called virtual machine introspection.

After reviewing threats against cloud hosted information systems it is clear that attacks towards virtual machines often target on-line applications and the underlying OS. Therefore acquiring real-time information about the list of running processes and the OS state in the deployed VMs has become a necessity. Virtual machine introspection is able to provide this information in an agentless manner guaranteeing minimal intrusiveness.

2.5.2.2.1 Virtual Machine Introspection After reviewing threats against cloud hosted information systems it is clear that attacks towards virtual machines often target on-line applications and the underlying OS. Therefore acquiring real-time information about the list of running processes and the OS state in the deployed VMs has become a necessity. Virtual machine introspection is able to provide this information in an agentless manner guaranteeing minimal intrusiveness. Security solutions that employ virtual machine introspection move monitoring and protection below the level of the untrusted OS and as such can detect sophisticated kernel-level malware that runs inside the deployed VMs.

What is Virtual Machine Introspection? The concept of introspection was introduced by Garfinkel et al. in [2]. In general terms

Definition 6 ***virtual machine introspection*** *is inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it.*

The advantages of using VMI as a security solution are two-fold:

1. As the analysis runs underneath the virtual machine (at the hypervisor level) it is able to analyze even the most privileged attacks in the VM kernel

2. As the analysis is performed externally it becomes increasingly difficult for the attacker to subvert the monitoring system and tamper with the results. As such a high confidence barrier is introduced between the monitoring system and the attacker's malicious code.

Unfortunately, as the monitoring system runs in a completely different hardware domain than the untrusted VM it can only access, with the help of the hypervisor, hardware-level events (e.g. interrupts and memory accesses) along with state-related information (i.e. physical memory pages and registers). The system then has to use detailed knowledge of the operating system's algorithms and kernel data structures in order to rebuild higher OS-level information such as the list of running processes, open files, network sockets, etc. The issue of extracting high-level semantic information from low-level hardware data is known as the *semantic gap*.

In order to bridge the semantic gap, the monitoring system must rely on a set of data structures, which can be used as templates in order to translate hypervisor-level observations to OS-level semantics. As such, the monitoring system is required to keep up-to-date detailed information about the internals of different commodity operating systems, thus making the widespread deployment of introspection-based security solutions unfeasible.

Virtuoso [110] attempts to overcome this challenge. Virtuoso is a framework that can automatically extract security-relevant information from outside the virtual machine. Virtuoso analyzes dynamic traces of in-guest programs that compute the introspection-required information. Then it automatically produces programs that retrieve the same information from outside the virtual machine. Although Virtuoso is a first step towards automatic bridging of the semantic gap it is limited only to information that can be extracted via an in-guest API call (such as getpid() in Linux OS). Moreover, Virtuoso does not address the main problem regarding VMI-aware malware: the fact that an attacker might affect the introspection result by altering kernel data structures and algorithms. An example of such malware is DKSM [111].

The de facto standard in performing virtual machine introspection is XenAccess [112]. The authors define a set of requirements for performing efficient memory introspection:

1. *No superfluous modifications of the hypervisor's code*,
2. *No modifications to the target VM*,
3. *Minimal performance impact*,
4. *Fast development of new monitors*,
5. *Ability to have a full view of the target OS* and
6. *Target OS cannot tamper with monitors.*

These requirements are met in XenAccess which provides also low-level disk traffic information in addition to memory introspection. XenAccess utilises Xen's native function *xc_map_foreign_range()* that maps the memory of one VM to another (in this case from DomU to Dom0, see Section 2.3.2.1.3), in order to access the monitored guest's memory which is then treated as local memory, providing fast monitoring results. For gathering the necessary information about the guest's OS a call to the XenStore database is made. XenAccess is a library that allows security monitoring frameworks to perform virtual machine introspection and is not a standalone monitoring framework. As such it does not incorporate any detection or prevention techniques.

LibVmi [113] is the evolution of XenAccess which extends introspection capabilities to other virtualization platforms like KVM. Besides extending XenAcess to other virtualization platforms, LibVmi offers significant performance improvements by utilizing a caching mechanism for requested memory pages. We use LibVmi in the implementation of the contribution of this thesis presented in Chapter 5.

Virtual machine introspection solutions can be classified into two main categories: *passive* and *active* monitoring, depending on whether the security framework performs monitoring activities by external scanning or not.

2.5.2.2.2 Cloud-Tailored IDSs The complexity and heterogeneity of a cloud environment combined with the dynamic nature of its infrastructure (see Section 2.2.5) make the design of a cloud-tailored IDS a challenging task. The problem is augmented when taking into account the security requirements of different tenants whose information systems often require customised security monitoring solutions that do not align with each other (i.e. different types of security threats, level of desired information, etc). The approaches described below detail IDS solutions that aim at addressing those challenges.

Roschke et al. [1] propose an IDS framework (see Figure 2.7) which consists of different IDS sensors deployed at different points of the virtual infrastructure. Each virtual component (e.g. virtual machine) is monitored by a dedicated sensor. All sensors are controlled by a central management unit which is also accountable for unifying and correlating the alerts produced by the different types of sensors.

The central management unit has four core components: *Event Gatherer*, *Event Database*, *Analysis Controller* and *IDS Remote Controller*. The Event Gatherer is responsible for

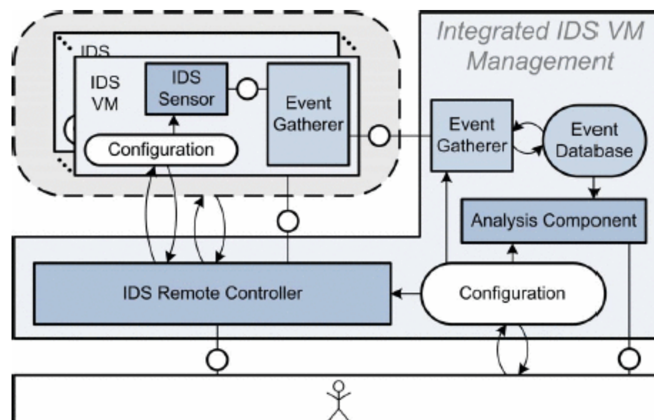


Figure 2.7 – The Cloud IDS architecture as in [1]

receiving and standardising alerts from the deployed sensors which are then stored in the Event Database. Alerts are then accessed by the Analysis component which performs correlation for the detection of multi-event complex attack scenarios. Finally the IDS Remote Controller is responsible for the lifecycle (start, stop, shutdown) and configuration of each IDS sensor.

Although the approach presented in the paper enables the use of different types of IDS sensors (host-based, network based) it does not account for the dynamic nature of the virtual infrastructure. For example, it is not clear whether the dedicated sensor is migrated along with the virtual machine in case of a VM migration. Furthermore, the reconfiguration of the IDS sensors is not automated (e.g. in the case where a new service is added in the deployed VMs). Finally, component sharing is not enabled even within the same virtual infrastructure.

In an attack-specific approach the authors of [114] try to tackle the threat of a Denial-of-Service event by deploying network-based IDS sensors next to each compute node of an IaaS cloud infrastructure. The proposed solution attempts to monitor each compute node by a separate IDS instance and then perform alert correlation in a central point. Although this approach clearly addresses the scalability issue of monitoring the whole traffic at a central point (e.g. one IDS instance attached to the network controller), there are several issues that remain unsolved. For example there is no mention of IDS reconfiguration in case of a changed set of services on the deployed VMs that are hosted in a particular compute node. Although the authors advocate for a distributed approach that will result to a better performing IDS sensor (in terms of packet drop rate) they do not address the case where an unexpected traffic spike occurs. The described framework only includes network-based IDSs, as opposed to [1] which includes different types of IDSs.

In an effort to address security in federated clouds as well as to tackle large-scale distributed attacks that target multiple clouds, the authors of [115] propose a layered

intrusion detection architecture. The framework performs intrusion detection in three different layers: Customer Layer (tenant virtual infrastructure), Provider Layer (provider physical infrastructure) and Cloud Federation Layer. Each layer is equipped with *probes*, which perform the actual detection functionality, *agents* which are responsible for gathering and normalizing the alerts generated by different types of probes, and finally *security engines* that perform the actual decision making by correlating the received alerts. The security engines are responsible for deciding whether different security events represent a potential distributed attack and for forwarding the results to a higher layer. The security engine in the cloud provider layer is able to detect whether parts of its cloud infrastructure have been compromised based on data that it receives from the security engines of different customers (i.e. tenants). Although the authors attempt to combine the results of security monitoring of the tenants and the provider, they do not address cases where reconfiguration of the monitoring probes is required (i.e. when dynamic events occur). Moreover it is not clear whether different security probes can be shared between tenants.

Livewire [2] was the pioneering work in creating an intrusion detection system that applies VMI techniques. Livewire works offline and passively. The authors use three main properties of the hypervisor (isolation, inspection and interposition) in order to create an IDS that retains the visibility of a host-based IDS while providing strong isolation between the IDS and a malicious attacker. A view of Livewire's architecture can be found in Figure2.8. The main components of the VMI-based IDS are:

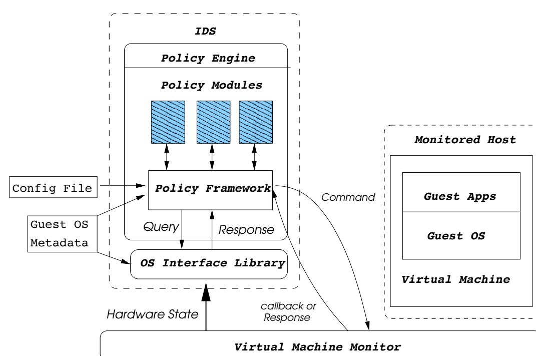


Figure 2.8 – The Livewire architecture as in [2]

- *OS interface library:* responsible for providing an OS-view of the monitored guest by translating hardware events to higher OS level structures. The OS interface library is responsible for bridging the semantic gap (see Section 2.5.2.2.1)

- *Policy Engine:* responsible for deciding if the system has been compromised or not. Different detection techniques (e.g. anomaly detection) can be supported by the policy engine in the form of policy modules.

The authors implemented their prototype on VMware workstation. As the first step towards using introspection in security monitoring, Livewire has some limitations. First, it does not address dynamic events in a cloud infrastructure, as it remains unclear if the dedicated IDS follows the VM in the event of a migration. Second the policy modules do not account for tenant security requirements and cannot be adapted in case a new service is added in the introspected VM. Finally, component sharing is not enabled as the design limits an IDS to a single VM.

HyperSpector [116] secures legacy IDSs by placing them inside isolated virtual machines while allowing them to keep an inside-the-guest view of the monitored system through virtual machine introspection. The authors use three mechanisms to achieve inside-the-guest visibility:

- *Software port mirroring:* the traffic from and to the monitored VM is copied to the isolated VM where the legacy IDS is running.
- *inter-VM disk mounting:* the file system of the monitored VM is mounted in the dedicated VM as a local disk, thus enabling integrity checks.
- *inter-VM process mapping:* the processes running inside the monitored VM are mapped to the isolated VM in the form of shadow processes with local identifiers. A dedicated function called *process mapper* running in the hypervisor is responsible for translating the local identifiers of the shadow processes to actual process identifiers in the monitored VM. The *process mapper* only provides reading access to the registers and memory of the shadow processes thus preventing a subverted IDS from interposing the monitored VMs functionality. Inter-VM process mapping is used for extracting information regarding the list of processes running inside the monitored VM.

Although HyperSpector secures legacy IDSs through virtual machine introspection, it suffers from the same limitations as Livewire.

Lares [117] attempts a hybrid approach in security monitoring through virtual machine introspection by attempting to do active monitoring while still maintaining increased isolation between the untrusted VM and the monitoring framework. The authors propose to install protected hooks in arbitrary locations of the untrusted VM's kernel. The purpose of the hook is to initiate a diversion of the control flow to the monitoring framework. Once a hook is triggered, for example in the event of a new process, then the execution in the untrusted guest is trapped and the control automatically passes in the monitoring software which resides in an isolated VM. The hooks along with the *trampoline* that transfers control to the monitoring software are write protected by a special mechanism in the hypervisor called *write protector*. The *trampoline* is also responsible for executing commands issued by the monitoring software and does not rely on any kernel functions of the untrusted VM.

Although Lares combines the benefits of isolation along with the ability to interpose on events inside the untrusted VM, it has some limitations that prevent its adoption in a cloud environment. First, the security VM cannot monitor more than one guest. This implies that for every VM spawned in a compute node, a corresponding security VM needs to be started as well, reducing the node's capacity for tenant VMs by half. Second, in the event of a VM migration, the tied monitoring VM needs to be moved as well, imposing additional load to the network. Finally, the list of monitored events is static, since the addition of a new event would require the placement of a new hook inside the untrusted VM.

CloudSec [3] attempts to provide active monitoring without placing any sensitive code inside the untrusted VM. The authors use VMI to construct changing guest kernel data structures in order to detect the presence of kernel data rootkits (e.g. kernel object hooking rootkits). The proposed framework is able to provide active concurrent monitoring for multiple colocated VMs. CloudSec does not directly access the memory pages of the untrusted VM. Instead, it interacts with the hypervisor for obtaining the corresponding pages which are stored in a dedicated memory page buffer (MPB). CloudSec uses a dedicated

module (KDS) in order to load information regarding kernel data structures of monitored VM's OS. Using the information from the KDS the Semantic Gap Builder (SGB) attempts to solve the semantic gap and build a profile of the monitored VM. Finally the profile is fed to the Defence Modules which perform the actual detection. An overview of the CloudSec architecture is shown in Figure2.9.

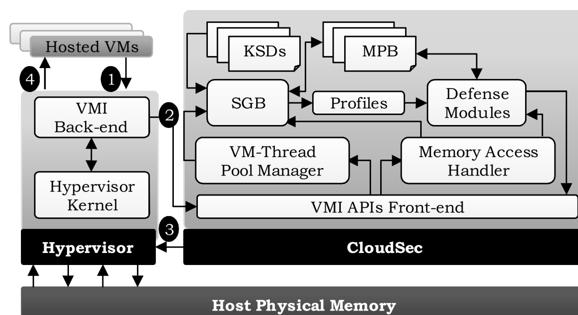


Figure 2.9 – CloudSec architecture as in [3]

Although CloudSec enables active monitoring for multiple VMs concurrently, the performance overhead of the solution in a multi-tenant environment has not been investigated. Furthermore, the active monitoring capabilities are limited to switching off an infected VM. CloudSec does not address dynamic events and is limited to VMware ESXi hypervisor.

KvmSec [118] is a KVM extension that enables active monitoring for untrusted guests from the host machine. While KvmSec is composed of multiple modules that reside in the host and in the untrusted guests, the authors place the core detection modules on the host side in order to provide tamper-resistant monitoring. Communication between the guest and host modules is enabled through a secure channel that enables information exchange. The guest module consists of a kernel daemon that creates and manages the secure communication channel and a second daemon that collects, analyses and acts upon received messages. The secure communication channel is created in shared memory with synchronised access through mutextes. Upon detection of a malicious event KvmSec is able to freeze or shutdown the monitored guest. Currently no other sanitization mechanisms are supported. KvmSec is able to extract the list of running processes inside the untrusted guest but no other detection modules are supported. Although theoretically, KvmSec might be able to monitor multiple consolidated VMs by enabling a shared memory region for each VM, the performance overhead of this approach remains unexplored.

The last five discussed solutions include passive and active monitoring frameworks that incorporate virtual machine introspection. Although passive monitoring is clearly a less invasive approach that favors stealthy monitoring (as there is no need for placing additional code in the untrusted guest), it lacks the ability to interpose on guest events. On the other hand, active monitoring enables the security monitoring framework to act on suspicious events but it requires hooks to be placed inside the untrusted VM, making it a more invasive solution. Passive monitoring solutions can be performed only at specific time intervals (known as introspection frequency), as opposed to active solutions that are triggered only when a suspicious event, like a memory region being accessed, occurs. Furthermore, although the discussed solutions in both categories provide some form of protection mechanisms (e.g. write protected memory regions) there is still a chance that an attacker can disable the hooks and render the result of introspection invalid.

In the contribution of this thesis presented in Chapter 4, we adapt NIDSs to dynamic changes that occur in a cloud environment in order to provide adequate monitoring of the network traffic that flows towards and from the cloud-hosted information system. Our contribution addresses the adaptation issues that are not taken into account in the previously-presented solutions.

2.5.2.2.3 Cloud Firewalls This section presents firewall solutions tailored for cloud environments. We focus on industrial solutions since there substantial effort is put on designing new cloud firewall solutions or adding new features to existing ones. We focus on two firewall categories: next-generation firewalls and application-level firewalls

Next-Generation Firewalls Nowadays large scale distributed attacks generate multiple security events at different levels of a cloud infrastructure and are considered amongst the most impactful cyber threats for a cloud environment. One solution in tackling these types of attacks is embedding a next-generation firewall in the cloud infrastructure. Next-generation firewalls are devices that are able to combine multiple functionalities in one: application-oriented access to the Internet, deep analysis of the network traffic (e.g. deep packet inspection), and finally a user-oriented access policy for on-line applications. In this section we discuss next-generation firewall solutions for cloud environments offered by major industry players.

A joint solution between VMware and PaloAltoNetworks [119] introduces the VM-Series next-generation firewall which is able to provide application-driven access control (in contrast to traditional firewalls that offer a port and IP address control policy). The proposed solution is able to dynamically adapt the enforced security policy when topology events (e.g. VM migration) occur. Their approach introduces a new feature called *tag* for VM identification. Each VM can have multiple tags that represent different features such as IP address, OS, etc. The user is allowed to create security rules based on tags instead of static VM objects. VM-Series is fully integrated in the NSX security suite (see Section 2.5.2.2.3) in order to gain access to the network traffic and topology information of the infrastructure. Unfortunately, VM-Series does not take into account specific tenant-security demands (e.g. protection against specific types of threats) and does not offer component sharing capabilities between different tenants. The VM-Series solution is also integrated in Amazon EC2 [15].

Application-level Firewalls In order to gain insight on which applications are generating network traffic, application-level firewalls rose as a solution. Application-level firewalls filter network packets based on a set of rules which refer to protocols and states of the involved applications. When this solution is applied to web applications hosted in a cloud environment it can offer protection against known application-level attacks (such as SQL injection or cross-site scripting, see Section 2.4.1).

The Amazon Web Application Firewall (WAF) [120] allows tenants to create their own security rules depending on the type of applications that are hosted in their virtual infrastructure. Tenants can gain visibility into specific types of requests by setting a dedicated filter through the WAF API or create access control lists if they require limited access to their applications. Once created, the rules are installed in a front facing load balancer. Although the WAF solutions offer substantial freedom to tenants, by allowing them to fully customize the deployed ruleset, it does not account for dynamic events (topology- or traffic-related events). Furthermore it is unclear if component sharing is enabled by installing rules of different tenants.

A distributed web application firewall is introduced by Brocade (former SteelApp) [121] that offers automatic rule generation based on application behavior. The in-learning capability of the solution is able to observe on-line applications for a period of time and create access control rules. Brocade WAF has three major components that resemble an SDN architecture model: the *Enforcer*, the *Decider* and finally the *Administration Interface*. The Enforcer is responsible for enforcing the security rules and inspecting the network traffic. If a packet that does not match the existing rules arrives then the Enforcer sends it to the Decider which decides whether to allow or block the connection. Then, the Decider generates a rule and sends it back to the Enforcer. As the rule generator, the Decider is the most computing-intensive part of the application and its load depends on the traffic of the application. The Decider is also responsible for auto-scaling in case of increased demand. Finally the Administration Interface is responsible for managing the WAF and inserting high-level policy rules that are then translated by the Decider to firewall rules. Although the solution is capable of autoscaling it is unclear what is the CPU penalty on colocated VMs.

In order to build a tamper-resistant, application-aware firewall that combines in-guest visibility with the isolation of a VMI monitoring framework, the authors of [122] created VMwall. Using XenAccess [112] VMwall is able to correlate processes running inside the untrusted guest with network flows. VMwall maintains a white list of processes that are allowed to make connections and compares the white list to the introspection-generated list. If a match is found a rule allowing the connection is inserted in a dedicated filtering module in Dom0.

Although VMwall is the pioneering work in creating introspection-based firewalls, it faces some limitations. First the white list of processes is statically defined and thus does not take into account the dynamic nature of a VM where services are continuously added or removed by tenants. Second, it does not address dynamic topology related events (e.g. VM migration) that occur in a cloud environment. For example, there is no mention of a prioritisation strategy when a migration event occurring in the middle of an introspection action. Finally, it is unclear whether the kernel filtering module can be shared between multiple VMs thus enabling sharing of the firewall component.

Xfilter [123] is a self-protection mechanism that filters outgoing packets in the hypervisor, based on information obtained through introspection. Xfilter was designed as an active defence mechanism against compromised VMs that are used as stepping stones to target hosts outside the cloud infrastructure. The framework operates in two phases: *Detection* and *Inspection*. During the detection phase Xfilter only inspects the packet header. Once an attack is detected it automatically passes to the *Inspection* phase where additional information for the packet is extracted through introspection (process name and ID that initiated the transfer, port number, destination IP, etc). Then a rule is automatically generated that blocks all packets with that particular set of characteristics. Due to its design, Xfilter is limited in filtering only outgoing connections, thus unable to address all security cases that are covered by a traditional firewall. As such it is an inadequate general-purpose traffic filtering option.

The introspection-based firewall solutions presented are unable to adapt their components based on the dynamic events that occur in a cloud infrastructure. The contribution of this thesis presented in Chapter 5 addresses dynamic events in virtual infrastructure and adapts its components automatically.

In this thesis we focus on application-level firewalls that adapt their components based on the list of services that are hosted in the cloud infrastructure.

2.5.2.2.4 VESPA: A Policy-Based Self-Protection Framework In this section, we present VESPA [124], a self-protection framework that addresses self-adaptation of security devices as a reaction to detected attacks.

VESPA was designed in order to tackle the heterogeneous nature of an IaaS cloud environment and provide lighter administration of the security devices, combined with lower response time (i.e. when a threat is detected) along with lower error rate (e.g. false positives/negatives). The four main design principles of VESPA are:

1. **Policy based self-protection**: The framework's design is based on a set of security policies that address the security objectives of the different stakeholders (i.e. tenants and the provider).

2. **Cross-layer self-defence**: Based on the fact that a cloud environment is composed of different software layers, the framework's response to an attack is not limited to a single layer and can involve protection as well as detection functions (as opposed to [1] where the framework's core functionality is detection).

3. **Multiple self-protection loops**: The framework offers the ability to select among different reaction paths in case of an attack. The security administrator can select between loops that offer different trade-offs between reaction time and accuracy.

4. **Open architecture**: The framework is able to integrate different off-the-self security components.

The authors created a four-layer framework that implements their four design principles. The first layer, called *Resource plane* consists of the cloud resources that need to be monitored (i.e. VMs, tenant networks, etc). The second layer, the *Security plane*, includes all off-the-shelf security components that can be detection devices (e.g. IDSs) or protection devices (e.g. firewalls). The *Agent plane* is used as a mediator between the heterogeneous security devices and the actual decision making process. The agents that are part of the *agent plane* act as collectors and aggregators for the different logs produced by the devices in the *security plane*. Finally, the last layer, called the *Orchestration plane*, is responsible for making the reaction decision when an attack towards the monitored infrastructure occurs.

Although VESPA is a security framework that tries to address self-adaptation of the security monitoring devices, the authors consider only security incidents as potential sources of adaptation. Other types of dynamic events (see Section 2.2.5) are not considered, consequently no reaction mechanisms for these events are implemented. Furthermore, VESPA does not include tenant-related security requirements in the definition of the reaction policies. Finally, although VESPA aims at including commodity security monitoring devices into the *security plane*, modifications on their source code are required in order to enable compatibility with the framework.

The contributions presented in this thesis adapt the security monitoring framework based on environmental changes (topology-, service- and traffic-related) as opposed to VESPA which addresses security incident-oriented adaptation. Furthermore, our contributions are able to respect tenant-defined security requirements in the adaptation process. Finally, our contributions do not require modifications on the detection components.

2.5.2.3 Security as A Service Frameworks (SecaaS)

Most cloud providers follow a shared-responsibility security model when it comes to cloud infrastructures: tenants are responsible for securing anything that they deploy or connect

to the cloud. In order to facilitate the security monitoring of a tenant's virtual infrastructure cloud providers offer complete monitoring frameworks in the form of products. In this section we discuss some of these products along with the list of services that they offer. Each provider offers Identity and Access Management solutions for regulating resource access (Amazon: [125], Google: [126], Microsoft: [127]).

- **Amazon**: Besides the AWS WAF that we discussed in Section 2.5.2.2.3, we focus on security monitoring components such as Shield [128] which is an Intrusion Detection component tailored towards DDoS attacks. Tenants can create their own rules to monitor traffic against specific types of DoS attacks like HTTP or DNS floods. Shield also provides mitigation techniques like rerouting and can be used in combination with WAF for setting proactive filtering against application-level attacks. Other available services include Certificate Manager for deploying SSL certificates. A full list of services can be found in [129].

- **Google**: Google Security scanner [130] is a proactive tool, which automatically scans web applications for known vulnerabilities (Flash or SQL injections, outdated libraries, etc). Tenants can use the results in order to generate traffic filtering rules that proactively block specific types of requests. The Resource Manager [131] regulates access to resources. Interconnected resources are represented hierarchically and users can set access rights to a group of resources simply by configuring a parent node.

- **VMware**: NSX, the network virtualization platform offers a variety of security tools including traditional edge firewalls that are exclusively managed by the tenant [29] or anti-spoofing mechanisms [132] that allow users to restrict access to a set of IP addresses that are determined to be spoofed. VMware also provides integrated third party security solutions like TrustPoint [133] which automatically detects network resources that are not yet configured by performing partial scans of the network. Trustpoint also offers remediation options such as automatically quarantining machines or uninstalling infected applications.

- **Microsoft**: Advanced threat analytics [134] is a specialised tool for detecting distributed attacks that generate seemingly unrelated events. The tool flags incidents that deviate from a previously established normal application behavior. Cloud App Security [135] is another solution for identifying applications that use the network, creating and enforcing customised filtering rules. This product targets SaaS cloud infrastructures.

This thesis proposes a design for a self-adaptable security monitoring framework with two separate instantiations (one for NIDSs and one for firewalls). Our approach borrows elements from Security as a Service frameworks (e.g. integration of tenant security requirements and traffic filtering based on the type of hosted applications) but does not offer a full set of security services like industrial SecaaS solutions.

2.6 Summary

This chapter gave an overview of the state of the art for this thesis. We started with a description of autonomic computing along with its key characteristics. Then the concept of cloud computing was introduced. Together these two complementary computing paradigms form the context in which the contributions of this thesis were developed. We

then focused on describing the IaaS cloud management system that was used in the deployment of our prototype, OpenStack. A description of network virtualization techniques and network management in OpenStack followed. Afterwards, we turned our attention to security threats in traditional information systems and cloud environments. We then presented an overview of the main components of a security monitoring frameworks focusing on two security components: Intrusion Detection Systems and Firewalls. The key observations from this chapter are:

- IaaS cloud environments are very dynamic. We have identified three main change categories: *Topology-related*, *Traffic-related* and *Service-related* changes. Despite the numerous available cloud security monitoring frameworks there are no solutions that address all three types of dynamic events. VESPA, a policy-based self-protection framework, addresses adaptation of the security monitoring framework but focusing on security events as the main source of adaptation (instead of the three types mentioned before).

- Although some of the industrial solutions discussed (e.g. Amazon web application firewall) include the option of integrating tenant-specific security requirements in the form of filtering rules, the rule generation is not automatic, forcing the tenants to write and install the rules themselves.

- Component sharing between tenants is essential in a cloud environment where multiple VMs are deployed in the same physical host. Although the described solutions recognize the necessity of a multi-tenant monitoring framework, it still remains a design requirement that has not been implemented to the best of our knowledge.

Security monitoring for tenant virtualized infrastructures has yet to receive significant attention in the cloud community. Although efforts aimed at including quality of service guarantees for different services in a cloud environment have been made [136], security monitoring requirements are still not included in cloud SLAs. To our knowledge a self-adaptable security monitoring framework that is able to adapt to the dynamic events of a cloud environment, allow tenant-driven reconfiguration of the monitoring devices and enable component sharing in order to minimise costs has yet to be implemented. The goal of this thesis is to design a framework that is able to address the main limitations of current solutions discussed in the state of the art. In the following Chapter 3 we present the high-level design of our framework. Our framework's two instantiations incorporate different concepts presented in the state of the art, namely intrusion detection systems and application-level firewalls. Our first instantiation, presented in 4, is a self-adaptable intrusion detection system tailored for cloud environments. In our second instantiation, presented in Chapter 5, we propose a novel design for securing an application-level firewall using virtual machine introspection. Our firewall is able to automatically reconfigure the enforced ruleset based on the type of services that run in the deployed VMs. To our knowledge none of the firewall solutions discussed are able to achieve this.

Chapter 3

A Self-Adaptable Security Monitoring Framework for IaaS Clouds

3.1 Introduction

In the previous chapter we presented the state of the art in security monitoring for IaaS cloud infrastructures. Our analysis has shown that the existing solutions fail to address all three categories of dynamic events in a cloud infrastructure (topology, monitoring load and service-related changes) while at the same time integrating monitoring requirements from different tenants. To address this limitation we have designed a self-adaptable security monitoring framework for IaaS cloud environments that is able to:

1. Take into account the various kinds of dynamic events in a cloud infrastructure and adapt its components automatically.

2. Take into account tenant-specific security requirements and reconfigure the security devices in such manner that the resulting configuration respects these requirements.

3. Provide accurate security monitoring results without introducing new vulnerabilities to the monitored infrastructure.

4. Minimise costs for both tenants and the provider in terms of resource consumption.

In order to illustrate the practical functionality of our framework, we use a simplified example of a cloud-hosted information system. We use the same example in the whole thesis in order to provide consistency for the reader.

This chapter presents the design and implementation of our framework. It is structured as follows: Sections 3.2 and 3.3 present the system and threat model under which we designed our framework. Section 3.4 details the objectives of our framework while Section 3.5 presents our simplified example. Section 3.6 details the high-level design of the adaptation process when a dynamic event occurs. The main components of our framework along with key implementation details are presented in Sections 3.7 and 3.8 respectively. Finally, Section 3.9 summarises our first contribution.

3.2 System Model

We consider an IaaS cloud system with a cloud controller that has a global overview of the system. Tenants pay for resources that are part of a multi-tenant environment based on a Service Level Agreement (SLA). Each tenant is in control of an interconnected group of VMs that hosts various services. No restrictions about the type of deployed applications are imposed on tenants. The VMs are placed on available physical servers that are shared between multiple VMs that may belong to different tenants. The cloud provider is responsible for the management and reconfiguration of the monitoring framework's components and tenants can express specific monitoring requirements through the SLA or a dedicated API that is part of the monitoring infrastructure. A tenant's monitoring requirements include: 1. Security monitoring for specific types of threats (e.g. SQL injection attempt, worms, etc), at different levels of the virtual infrastructure (application, system, network) and 2. Performance-related specifications in the form of acceptable values (thresholds) for monitoring metrics. An example of a tenant-specified threshold could be: the maximum accepted value for the packet drop rate of a network intrusion detection system. The tenant specifications may lead to the reconfiguration of security monitoring devices that are shared between tenants or between tenants and the provider.

The cloud controller is responsible for providing networking capabilities to the deployed VMs. Two types of networks are constructed: an internal one between VMs that belong to the same tenant and an external one that is accessible from outside the infrastructure. Each deployed VM is assigned two IP addresses and two domain names: an internal private address and domain name and an external IPv4 address and domain name. Within a tenant's virtual infrastructure, both domain names resolve to the private IP address while outside the external domain is mapped to the external IP address.

3.3 Threat Model

We consider software attacks only, that originate from inside or outside the cloud infrastructure. We assume that like any legitimate tenant, an attacker can run and control many VMs in the cloud system. Due to multiplexing of the physical infrastructure, these VMs can reside in the same physical machine as potential target VMs. In our model an attacker can attempt a direct compromise of a victim's infrastructure by launching a remote exploitation of the software running on the deployed VM. This exploitation might target different levels in the victims infrastructure (system, network, applications). We consider all threats described in Section 2.4.1 to be applicable on a victim's VMs. Upon successful exploitation, the attacker can gain full control of the victim's VM and perform actions that require full system privileges such as driver or kernel module installation. Malicious code may be executed at both user and kernel levels. The attacker is also in position of using the network. We consider all attacker-generated traffic to be unencrypted.

In this work we consider the provider and its infrastructure to be trusted. This means that we do not consider attacks that subvert the cloud's administrative functions via vulnerabilities in the cloud management system and its components (i.e. hypervisor, virtual switch, etc). Malicious code cannot be injected in any part in the provider's infrastructure and we consider the provider's infrastructure to be physically secure.

3.4 Objectives

The goal of this thesis is to design a self-adaptable security monitoring framework that detects attacks towards tenant's virtualised information systems. We have defined four key properties that our framework needs to fulfill: self-adaptation, tenant-driven customization, security and correctness and finally, cost minimization. In this section we detail each of them.

3.4.1 Self Adaptation

Our framework should be able to automatically adapt its components based on dynamic events that occur in a cloud infrastructure. Consequently, the framework should be able to alter the existing configuration of its monitoring components, instantiate new ones, scale up or down the computational resources available to monitoring components and finally, shut down monitoring components. We distinguish three adaptation categories depending on their source:

- **Service-based adaptation:** In this category the framework's components need to be adapted due to a change in the list of services that are hosted in the virtual infrastructure. Addition or removal of existing services could impact the monitoring requirements, thus require the instantiation of new monitoring devices or reconfiguration of existing ones.

- **Topology-based adaptation:** In this category, the source of adaptation lies in changes in the virtual infrastructure topology. Sources of these changes include tenant decisions regarding VM lifecycle (i.e. creation, deletion) and provider decisions regarding VM placement (i.e. migration). The security monitoring framework should be able to adapt it's components in order to guarantee an adequate level of monitoring despite the new virtual infrastructure topology.

- **Monitoring load-based adaptation:** In this category, the framework needs to react to changes in the monitoring load. In the case of network traffic monitoring, an increase in the traffic flowing towards and from applications hosted in the virtual infrastructure would trigger an adaptation decision that would guarantee that enough processing power and network bandwidth (if the monitoring device is analyzing network traffic) is provided to the monitoring components. An adaptation decision could also involve the instantiation of a new monitoring probe that will be responsible for a particular traffic segment. In the case of VM activity monitoring, a sudden increase in inside-the-VM activity (i.e. running processes, open files, etc) could lead to altering the computational resources available to the security probe monitoring that particular VM.

3.4.2 Tenant-Driven Customization

Our framework should be able to take into account tenant-specific security requirements. These requirements include application-specific monitoring requests (i.e. requests for detecting specific types of attacks depending on the application profile) and monitoring metrics requests (i.e. detection metrics or performance metrics for the monitoring devices). The framework should be able to consider a given tenant's requirements in reconfiguration decisions and enforce these requirements on the affected monitoring devices.

3.4.3 Security and Correctness

Our framework should be able to guarantee that the adaptation process does not introduce any novel security vulnerabilities in the provider's infrastructure. The reconfiguration decisions should not introduce any flaws in the monitoring devices and should not affect the framework's ability to maintain an adequate level of detection. The monitoring devices should remain fully operational during the reconfiguration process.

3.4.4 Cost Minimization

Our framework should minimise costs in terms of resource consumption for both tenants and the provider. Deploying our framework should minimally affect the provider's capacity to generate profit by multiplexing its physical resources. The distribution of computational resources dedicated to monitoring devices should reflect a tenant-acceptable trade-off between computational resources available for monitoring and computational resources available for VMs. The performance overhead imposed by our framework to tenant applications that are deployed inside the monitored VMs should be kept at a minimal level.

3.5 Example Scenario

Our simplified example of a cloud hosted information system is depicted in Figure 3.1. In

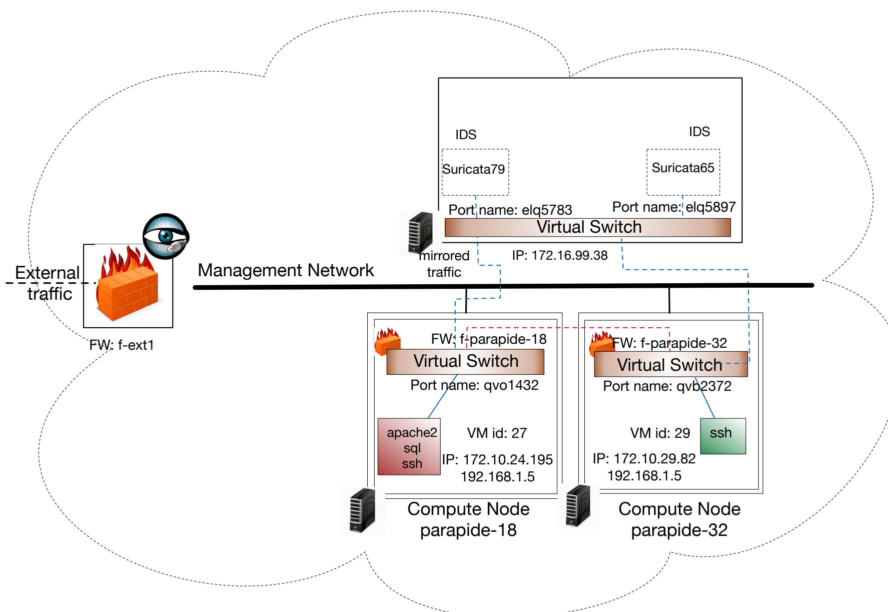


Figure 3.1 – An example of a cloud hosted information system

our example, two different VMs that belong to different tenants (depicted with red and green) are deployed on different compute nodes. The first VM with ID *27* is deployed on node *parapide-18* and hosts two services: an *SQL-backed apache server* and an *ssh server*. The public IP of the VM is *172.10.24.195* and the private IP is *192.168.1.5*. The VM is plugged on the virtual switch of the compute node with a port named *qvo1432*. The second VM with ID *29* is deployed on node *parapide-32* and hosts only one service, an *ssh server*. The public IP of the VM is *172.10.29.82* and the private IP is *192.168.1.5* (the private IPs of VMs that belong to different tenants can overlap).

In this simplified example, we include only two types of monitoring devices: network-based IDSs and firewalls. The traffic flowing towards and from the VM on node *parapide-18* is monitored by a network-based IDS named *suricata79* residing on a separate node with IP *172.16.99.38* while the traffic flowing towards and from the VM on node *parapide-32* is monitored by another network-based IDS named *suricata65* residing on the same node. Each compute node has a firewall at the level of the virtual switch (named *f-parapide18* for the compute node *parapide18* and *f-parapide32* for the compute node *parapide32*). Finally, an edge firewall named *f-ext1* is responsible for filtering the traffic that flows towards and from the cloud infrastructure to the outside world.

3.6 Adaptation Process

After defining the four main objectives of our monitoring framework, we now describe the three levels of the adaptation process. The process begins from the adaptation sources, that can either be dynamic events or changes in the cloud infrastructure (topology-, service- or monitoring load-related) or evolving tenant security requirements. It continues with our framework's decision making. Finally, the adaptation decision is enforced by reconfiguring the affected security devices.

First, the adaptation process is triggered by either a change in the cloud infrastructure (i.e. service, topology or monitoring-load related) or a tenant specific security requirement. All necessary information is extracted and forwarded to the adaptation framework. Depending on the type of change different information is propagated to the framework:

- *Service-related change:* type of service and technical specifications (e.g. port numbers or range, protocol, authorized connections/users, etc).
- *Topology-related change:* ID of the affected VM along with network information (e.g. internal/external IP, port on the virtual switch, etc) and the physical node hosting the VM.
- *Monitoring load-related change:* device-specific metrics that demonstrate the effect of the monitoring load fluctuation on the monitoring functionality (e.g. packet drop rate, memory consumption, etc).

The information extracted from a tenant security requirement includes: specific security events (e.g. attack classes or specific threats) and monitoring metrics (e.g. packet drop rate). The propagated information is extracted from different sources (i.e. the cloud engine, monitoring devices, SLA, etc).

Once the adaptation framework receives the propagated information, it starts making the adaptation decisions. The first step in the decision making process is identifying the monitoring devices affected by the adaptation. The adaptation framework is able to extract the list of the devices based on the VMs involved in the dynamic events. Depending on the monitoring strategy selected, the group of VMs assigned to a specific monitoring device could be determined based on their physical location (e.g. an NIDS monitoring the traffic that flows towards and from all VMs that are deployed on a particular compute node). The framework has full access to topology and networking information for each monitoring device. This information includes: 1. name and IP address of the physical node hosting the

device (e.g. if a device is running in a container), 2. IP address of the device if applicable, 3. list of other co-located devices and finally 4. list of computational resources available on the node hosting the device (e.g. CPU, memory, etc).

After the adaptation framework has identified the list of affected monitoring devices, it makes the adaptation decision. The adaptation decision can imply the reconfiguration of the monitoring devices so that monitoring for specific types of threats is included or removed. It can also imply the instantiation of a new monitoring device. The framework can also decide to assign more computational resources to a group of monitoring devices in order to be able to better manage their computational load. After the decision has been made, it is translated to device specific reconfiguration parameters by dedicated framework components.

The final stage of the adaptation process is executed at the level of the monitoring devices. The device-specific reconfiguration parameters are taken into account and the monitoring devices are adapted accordingly. The adaptation framework is able to maintain an adequate monitoring level even during the reconfiguration phase either by using live reconfiguration capabilities of the devices (when applicable) or by incorporating other strategies, which enable later inspection of activity (e.g. temporary clone of an HIDS, storing traffic for later inspection in the case of an NIDS). After the adaptation process is complete the affected monitoring devices are fully operational.

Although we consider network reconfiguration events such as network migrations part of topology-related changes, our framework does not handle network reconfiguration events at this stage.

3.7 Architecture

This section presents the architecture of our self-adaptable security monitoring framework. First a high-level overview of the system is presented followed by the description and functionality of each component.

3.7.1 High-Level Overview

The high-level overview of our framework's architecture is shown in Figure 3.2. The figure depicts an IaaS cloud with one controller and two compute nodes on which the tenant's virtualised infrastructure is deployed. Different components of our self-adaptable security monitoring framework are included in the figure. A dedicated node is used for hosting different network IDS while an edge firewall filtering the traffic between the outside world and the cloud is deployed on a standalone host. Firewalls are also included at the level of the local switches on the compute nodes. Finally, a log aggregator collects and unifies the events produced by the different types of security devices.

Our framework is composed of three different levels: tenant, adaptation and monitoring devices. The monitoring devices level consists of probes (NIDS and firewalls in Figure 3.2) as well as log collectors and aggregators.

The adaptation level consists of all the framework's components that are responsible for designing and enforcing the adaptation process. A dedicated Adaptation Manager, which can be located in the cloud controller, acts as a decision maker. Dedicated components named Master Adaptation Drivers (MAD), located in the nodes that host the monitoring devices, are responsible for translating the manager's decision to component-specific configuration parameters are also part of this level. A MAD can be responsible

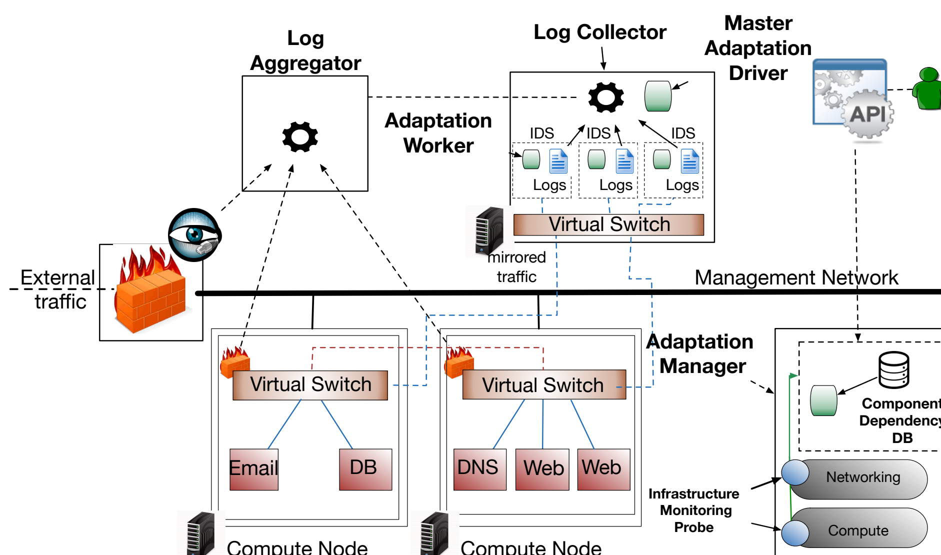


Figure 3.2 – The framework's architecture

for multiple security devices. Depending on the number of security devices it is up to the manager to decide how many MADs are instantiated. A dependency database, also located in the cloud controller, that keeps updated information regarding dependencies between monitoring devices (i.e. different types of monitoring devices that monitor the same VM) is also part of the adaptation level. A lower level agent called Adaptation Workers (AW), is tasked with enforcing the actual reconfiguration parameters on each monitoring device and guaranteeing continuous operation through the adaptation process. Our framework features one Adaptation Worker per monitoring device. Finally, the Infrastructure Monitoring Probes (IMPs), which are located inside core modules of the cloud engine, are responsible for detecting topology-related changes.

The third level of our framework's architecture includes the tenant API. All available monitoring options are accessible through the dedicated API. A representation of the three different levels can be found in Figure 3.3.

After presenting a high-level overview of our framework's architecture we now describe each component in detail.

3.7.2 Tenant-API

One of our framework's core objectives is integration of tenant-specific security monitoring requirements. Tenants can request monitoring against specific attack classes depending on the profile of their deployed applications (e.g. for a DBMS-backed web application a tenant can request monitoring for SQL injection attempts). Furthermore, tenants may have specific requests regarding the quality of monitoring in terms of device-specific metrics (e.g. a tenant can request a lower threshold for the packet drop rate of a NIDS system). In order to facilitate tenant requirement integration, our framework provides a dedicated API that is exposed to the tenants and allows them to express monitoring specifications in a high level manner.

Essentially, the API performs a translation between the tenants monitoring objectives, which are expressed in a high-level language, and our framework-specific input sources. Our monitoring framework then takes into account the outcome of the translation for making an adaptation decision. We now describe the API design.

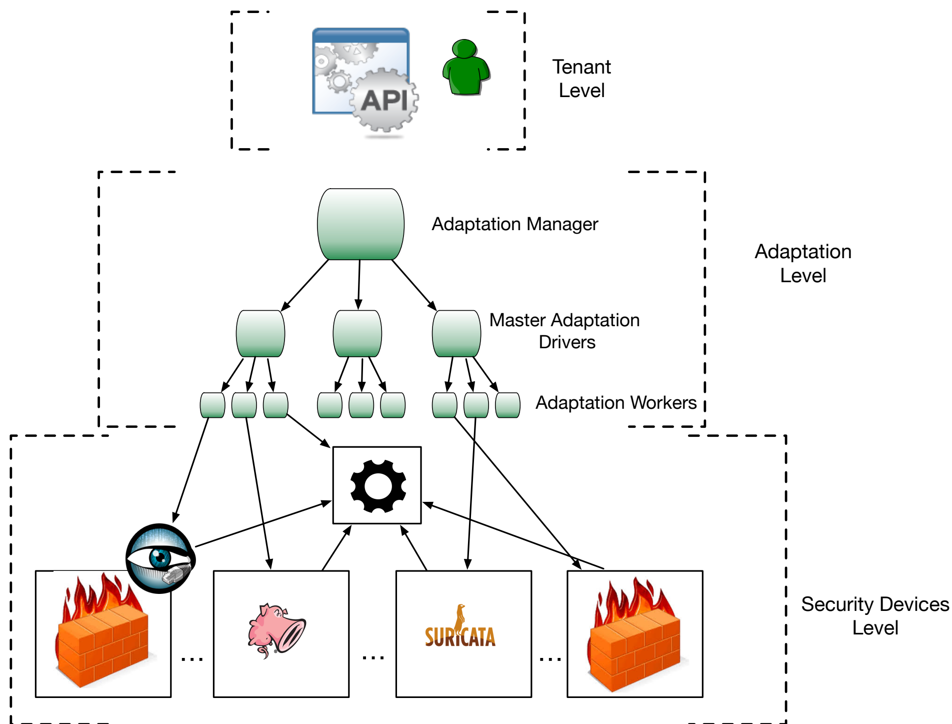


Figure 3.3 – The framework's different levels

3.7.2.1 API Design

The design of our API is organised in three distinct parts. We detail each one. In order to simplify authentication we have made the design choice to integrate our API into the provider's API and make it available through the web.

3.7.2.1.1 Tenant-exposed part: The first part of our API is directly exposed to the tenants. Each tenant uses its unique identifier in order to access the tenant-exposed part through the web. After successful authentication, the tenant has access to the list of monitoring services that are activated in its virtual infrastructure along with detailed record about each service. The information available about each monitoring service are: attack/threat classes (e.g. SQL injection, cross site scripting, etc), list of VMs that are under this monitoring service and finally, a time field that specifies when this option was activated.

A tenant can add a new monitoring service or remove an existing one through a dedicated add/delete option in the API. In the event of a new monitoring service addition, the tenant is given the option to select a monitoring service only amongst the ones that are available/supported by the self-adaptable monitoring framework.

After selecting the monitoring service the tenant adds the IDs of the VMs that it wants this service to be applied on. A second option available for tenants is tuning of SLA-defined monitoring metrics. Each tenant has access to a list of SLA-defined monitoring metrics and can increase or decrease their value.

Finally, a list of the applications that are deployed on its provisioned VMs is provided by each tenant. The information available for each application is:

- its name.

- connectivity record. In the connectivity record the tenant specifies network-related information about the service. This information includes list of ports that the service is expected to use and the list of restricted IPs that are allowed to interact with the application (if applicable).

- VM ID that the service is running on.

3.7.2.1.2 Translation part: The translation part of our API lies one level lower than the tenant-exposed part and is actually performing the translation between the high-level description of tenant requirements to framework-specific information. The translation part parses the tenant-generated input and performs two functionalities for each monitoring service: 1. mapping of the high-level service name to framework-specific service name (if required), and 2. mapping of the instances names to VM cloud-engine IDs. Furthermore the translation part extracts the names of the applications along with the number of ports and the list of allowed IPs (if applicable). The extracted information forms the necessary records required by our framework in order to make an adaptation decision.

Finally, in order to allow our framework to make adaptation decisions on a VM basis, the information is grouped in a VM-based manner (cloud engine ID of the VM, list of running processes and network connectivity, monitoring services). As a last step the translation part generates a framework-readable file with a specific format (e.g. XML format) with the VM-based information and the tenant-defined values of the SLA-specified monitoring metrics. The generated file is unique per tenant. The file depicting the types of services along with specific monitoring requirements for the VM with ID 27 of the example in Section 3.5 can be found in Listing 3.1.

Listing 3.1 – SLA information file

```
1 <Tenant Id="74cf5749-570">
2   <vm id="27">
3         <services authorised_destination_IPs="192.168.1.5"
              authorised_source_IPs="192.168.1.2, 192.168.1.3" dport="22"
              name="ssh" proto="tcp" role="server" sport="0.0.0.0">
4         </services>
5         <services authorised_destination_IPs="172.10.24.195"
              authorised_source_IPs="all" dport="80" name="apache2" proto="
              tcp" role="server" sport="0.0.0.0">
6         </services>
7         <services> name="sql" </services>
8   </vm>
9   <IDS>
10        <additional_monitoring="worm">
11        <drop_rate> accepted=5 </drop_rate>
12  </IDS>
13 </Tenant>
```

In the example of Section 3.5, the tenant with ID 74cf5749-570, has provisioned only one VM on which it deployed an ssh server and an SQL-backed web server. It requested additional monitoring against worms and it accepts a drop rate (for an NIDS) that does not exceed 5%. Each time a tenant expresses a new monitoring requirement the file is regenerated. After describing the different parts of our tenant-exposed API and their functionalities we continue in detailing another type of components of our framework, the security devices.

3.7.3 Security Devices

Security devices include all devices and processes that perform the actual monitoring functionality. The type of devices included are: intrusion detection systems (network or host based), firewalls, vulnerability scanners, antiviruses, etc. The monitoring devices can be installed at any point in the cloud infrastructure and can monitor part of the tenants or the provider infrastructure.

Although the monitoring devices perform different types of monitoring under different configurations, the common denominator between all types of devices is the production of detailed log files. In order to efficiently manage and unify logs originating from the security devices we include log collectors and aggregators in this category (although they do not perform actual monitoring tasks). Log collectors can be co-located with one or multiple monitoring instances and can perform local or remote collection of logs. Aggregators are responsible for looking for specific patterns, defined by the framework's administrator, inside the log files and summarizing events.

3.7.4 Adaptation Manager

The Adaptation Manager (AM) is one of our framework's core components. It is responsible for making the adaptation decisions that affect the monitoring devices of the monitoring framework. The AM is able to handle dynamic events inside the cloud infrastructure and guarantees that an adequate level of monitoring is maintained. The Adaptation Manager has a complete overview of the state of the monitoring framework which is comprised by the following information:

- topological overview: list of monitoring devices and their location (nodes on which they are deployed and IP addresses of the nodes),
- functional overview: a mapping between VMs and monitoring devices. One device can be mapped to multiple VMs and vice versa. The functional overview of the system provides the necessary information regarding which monitoring device is monitoring which subset of the deployed VMs. Depending on the monitoring strategy selected, a monitoring device can be responsible for all the VMs that are hosted in a particular location (e.g. an NIDS monitoring the traffic that flows towards and from all the VMs deployed on a specific compute node).

Upon the occurrence of a dynamic event (e.g. VM migration) the AM performs the actions presented in Algorithm 1 in order to make an adaptation decision:

Algorithm 1 The adaptation decision algorithm

```
1: function ADAPTATION(dynamic_event)
2:     list of services ← MAP(dynamic_event.VM_id, vm_information_file )
3:     affected devices, agents ← MAP(dynamic_event.VM_id)
4:     for i in affected devices do
5:         reconfiguration required ← DECIDE(i, list of services)
6:     PROPAGATE DECISION(agents, reconfiguration required)
```

- Map the ID of the VM affected by the change to the list of services running inside the VM (line 2 in Algorithm 1). This is done by parsing the information provided by the API-generated file (*sla_info.xml* in Listing 3.1).

- Identify the monitoring devices responsible for the affected VM (line 3 in Algorithm 1). These are the monitoring devices that will be adapted. This is done by using information that is provided by the Component Dependency Database (see Section 3.7.6). The information regarding the list of running services and the list of monitoring devices that are going to be adapted are combined in a single file called *vm_information* file. The resulting file for the example information system described in Section 3.5, can be found in Listing 3.2. In the example of Section 3.5, three services are deployed on that particular VM with ID 27 (ssh server, apache web server and an SQL database) while the VM is monitored by a signature-based IDS named suricata65.

Listing 3.2 – VM information file

```
1  <vm id="27">
2      <services authorised_destination_IPs="192.168.1.4"
         authorised_source_IPs="192.168.1.2, 192.168.1.3" dport="
         22" name="ssh" proto="tcp" role="server" sport="all">
3      </services>
4      <services authorised_destination_IPs="172.10.124.195"
         authorised_source_IPs="all" dport="80" name="apache2"
         proto="tcp" role="server" sport="all">
5      </services>
6      <services name="sql">
7      <currentIDS host_ip="172.16.99.38" name="suricata79" type="
         signature_based" additional_monitoring="worm" drop_rate=
         "5" > </newIDS>
8  </vm>
```

- Decide on the type of reconfiguration required (line 5 in Algorithm 1). Depending on the type of monitoring devices and the event category different reconfiguration types might be necessary (e.g. rule addition or removal, module activation, white list creation, new probe instantiation, computational resource redistribution, etc).

- Propagate the reconfiguration parameters to the agents responsible for enforcing the adaptation decision (line 6 in Algorithm 1).

In case of a topology-related dynamic event all steps are performed while in the case of a service- or monitoring load-related change or a tenant-specific changed monitoring requirement only steps 3 to 6 are performed.

The AM is also responsible for handling performance degradation of the monitoring probes. The AM sets predefined thresholds for a set of device specific performance metrics and then allows each monitoring device to raise an alert in case one of the predefined thresholds is violated. The AM then decides if a new probe is necessary. If a new probe is instantiated the AM propagates the necessary information regarding monitoring load redistribution to the lower level agents.

In a cloud environment often dynamic events occur simultaneously. In order to handle the adaptations of the security devices that originate from these events, the AM can handle multiple adaptation events simultaneously. In the event of two different adaptation decisions affecting the same existing monitoring device, we distinguish three outcomes depending on the arguments of the adaptation decisions:

- The adaptation decisions contain different adaptation arguments: In this case there is no conflict between the decisions and the reconfigurations can proceed.

- The adaptation decisions contain the same arguments or there is a partial match between the two argument sets: In this case depending on the nature of the adaptation decisions (activation or deactivation of monitoring parameters) we can foresee two outcomes:

  1. Both decisions lead to activation or deactivation of monitoring parameters: In this case there is no conflict and the reconfigurations can proceed.
  2. One decision leads to activation of monitoring parameters while the other to deactivation: In this case there is a conflict between the reconfigurations. In order to guarantee an adequate level of detection, our framework adopts the strategy of keeping the matching arguments activated.

3.7.5 Infrastructure Monitoring Probes

The Infrastructure Monitoring Probes (IMPs) are located inside different core modules (networking, compute) of the cloud engine and are responsible for detecting topology related changes. The detected changes include VM lifecycle (e.g. start, stop) and placement (i.e. migration) changes. Once a topology-related change occurs an IMP intercepts the dynamic event and extracts all the necessary VM-related information from the cloud engine. The information includes: networking records (external and internal IP address, network port on the virtual switch) and compute records (VM ID, source and destination node – in case of a migration–, tenant ID) of the affected instance. Then the IMP forwards this information to the Adaptation Manager in order to make the adaptation decision. Although located inside the cloud engine IMPs do not preempt normal cloud operations (e.g. VM-lifecycle decisions or network-related reconfigurations) during the reconfiguration of monitoring devices.

3.7.6 Component Dependency Database

In complex security monitoring frameworks that consist of different components, interdependencies between security devices can lead to troublesome security issues. Reconfiguration of a single monitoring component can create the need for reconfiguring a set of secondary monitoring devices. In the case of our framework, an adaptation decision that was triggered by a dynamic event (e.g. a service stop inside a monitored guest) can affect separate security devices: an active monitoring device (e.g. a firewall) and a passive monitoring device (e.g. an IDS). In both devices reconfiguration is necessary in order to reflect a change in the monitoring process that was caused by the dynamic event (e.g. delete rules that filter traffic for the stopped service for the firewall and de-activate the rules that monitor traffic for the stopped service in the IDS). In order to facilitate identification of all affected devices when making an adaptation decision, we introduce the Dependency Database. The Dependency Database is located inside the cloud controller and is responsible for storing security device information for each monitored VM.

Our dependency database consists of two separate tables a *VM_info* table and a *Device_info* table that provide respectively the functional and topological views to the Adaptation Manager. The columns in the *VM_info* consist of the names of all security devices involved in the monitoring of a particular VM (identified with its ID, placing one VM per line). Using the VM ID as a primary key, the Adaptation Manager can extract the list of monitoring devices that are responsible for this VM. These are the devices that are affected by an adaptation decision caused by a dynamic event involving that VM. The *VM_info* table for the VMs of the example of Section 3.5 can be found in Table 3.1. In this

Table 3.1 – The VM_info table

VM ID	Network IDS	Host IDS	External-firewall	Switch-firewall
27	suricata79	ossec1	f-ext1	f-parapide-18
29	suricata65	ossec4	f-ext1	f-parapide-32

example we see that for the VM with ID 27 there is a network IDS named *suricata79*, a host IDS named *ossec1* and two different firewalls, one edge, named *f-ext1*, and one inside the local switch, named *f-parapide-18*. A single VM can be monitored by different types of IDS (host- and network-based).

The *Device_info* table is used to store device specific information. The Adaptation Manager uses each device name in order to extract the following information: location of the device (IP address of the physical node hosting the device) and type of the device The *Device_info* table for *suricata65* IDS can be found in Table 3.2. In this example we see

Table 3.2 – The Device_info table

Device Name	Location	Type of device
suricata65	172.16.99.38	signature_based

that the *suricata65* network IDS is located on a node with IP address 172.16.99.38 and is a signature-based NIDS. When a dynamic event occurs, the AM uses the information available in the Dependency Database to identify the full list of affected devices. Each time a new monitoring device is instantiated a corresponding entry with all the necessary information is added by the AM in the two tables.

3.8 Implementation

We have developed a prototype for our framework from scratch in Python. We used OpenStack (version Mitaka) [32] as the cloud management system. In order to enable network traffic mirroring we used Open vSwitch (OvS) [137] as a multilayer virtual switch. OvS is only compatible with later versions of OpenStack that use Neutron for providing networking services for deployed VMs. Consequently, version Mitaka was selected. We used Libvirt [138] for interacting with the underlying hypervisor. This section presents a few important implementation aspects. Namely, we focus on the details of two of our framework's main components: the Adaptation Manager and the Infrastructure Monitoring Probe.

3.8.1 Adaptation Manager

In order for the manager to be able to handle multiple adaptation events in parallel, a multi-threaded model approach was adopted. A master thread is responsible for receiving notifications regarding topology-related changes from the Infrastructure Monitoring Probes. The notification mechanism currently supports two versions: creating and listening to a dedicated socket, or placing a notification adapter (using the inotify [139] Linux utility) in a specific directory for tracking events (*modify*, *close_write*) on the directory's files. Once the AM receives an event, the AM performs the steps described in Algorithm 2:

1. A worker thread is spawned for handling the considered adaptation event. In order to retrieve the information about the VM involved in the topology change,

Algorithm 2 Adaptation when A VM migration occurs

```
1: function ADAPTATION(VM_network_info)
2:     SPAWN ADAPTATION THREAD
3:     list of services ← INFORMATION PARSER(VM_network_info.VM_id, vm_information_file)
4:     affected devices, locations ← INFORMATION PARSER(VM_network_info.VM_id,
                                      VM_network_info.source_node,
                                      VM_network_info.destination_node,
                                      topology.txt)
5:     for i, j in affected devices, locations do
6:         args.txt ← DECIDE(list of services, i)
7:         IDS_CONN(j, args.txt, +/-)
```

the thread parses the *vm_information_file.xml* (in Listing 3.2) using the *information_parser* function. Using the VM ID as an identifier, the function extracts the list of services running inside the affected guest and the tenant-specific security requirements.

2. The AM makes the adaptation decision and the parameters (e.g. in case an NIDS is involved, which types of rules will be activated/deactivated, what is the tenant acceptable drop rate) are written to a dedicated file named *args.txt*. In order to extract the names, types and location of the affected security probes the worker parses a separate file (*topology.txt*) containing the topological and functional views necessary for the AM. The *topology.txt* file containing the topological and functional views for the information system described in Section 3.5 can be found in Listing 3.3.

Listing 3.3 – topology and functional information file

```
1 Compute-Node    IP    IDS  IDS-Node
2 parapide-18.rennes.grid5000.fr    172.16.98.18    suricata79
      172.16.99.38
3 parapide-32.rennes.grid5000.fr    172.16.98.32    suricata65
      172.16.99.38
```

In the example of Section 3.5, the monitoring strategy described includes one NIDS per compute node. All NIDSs are deployed on the same node. Once a VM migration occurs, for example for the VM with ID 27, the master thread receives the network-related information from the IMP (public IP = 172.10.24.195, private IP = 192.168.1.5, source = parapide-18.rennes.grid5000.fr, destination = parapide-32.rennes.grid5000.fr, port name on the virtual switch of the destination node= qvb1572). Once it receives this information the worker thread parses the *vm_information_file.xml* and the *topology.txt* files and it extracts the list of services running in the migrated VM (sshd, apache2, sqld), the additional tenant-defined monitoring requirements (worm), the tenant specific monitoring metrics (drop rate threshold of 5%) and finally the names of the NIDS that are responsible for monitoring the traffic in the source and destination nodes (*suricata79* and *suricata65* respectively) along with their host IP address (172.16.99.38). These NIDSs are the two devices that need to be adapted. The worker thread then writes the adaptation arguments to *adaptation_args.txt*. The result for the NIDS monitoring the traffic towards and from the destination node (*suricata65* in the example of Section 3.5) is shown in Listing 3.4.

Listing 3.4 – The file containing the adaptation arguments for an NIDS

```
1 signature_based
2 suricata65
3 apache2
4 sql
5 ssh 192.168.1.2, 192.168.1.3
6 worm
7 5
```

3. The worker thread sends the dedicated file through a secure connection (using a dedicated function called *ids_conn*) to a MAD located in the node(s) hosting the affected security devices. The *ids_conn* function uses the IP address of the node hosting the device, and the name of the security device in order to establish the connection

4. A dedicated operator (e.g. + or -), that is decided by the AM is sent together with the file containing the adaptation arguments, indicates whether the adaptation requires an activation or deactivation of monitoring parameters. In our example, the operator sent with the file in Listing 3.4 is a + indicating that the monitoring parameters need to be activated. In case of an adaptation decision that affects multiple security components in different locations, a separate thread per component is created in order to facilitate the parallel transmission of the adaptation file.

3.8.2 Infrastructure Monitoring Probe

3.9 Summary

In this chapter we have described the design of a self-adaptable security monitoring framework. Our framework was designed in order to address the four main objectives: self-adaptation, tenant-driven customization, security and cost minimization. In this chapter we described how the core component of our framework, the Adaptation Manager, orchestrates the adaptation decisions in order to meet the self-adaptation and tenant-driven customization objectives.

A detailed description of the adaptation process, from the dynamic event that triggers the adaptation to the actual reconfiguration of the security probes was presented. During the process, we have demonstrated that the Adaptation Manager respects tenant-defined monitoring metrics by including them in the adaptation parameters. The AM is able to make the adaptation decisions independently from the type of security device. Consequently, our framework is able to integrate different types of security monitoring devices. The Master Adaptation Drivers (described in more detail in the following chapter) are responsible for translating the adaptation decision to device-specific parameters. The remaining two objectives (security and cost minimization) are discussed in the following chapters. Furthermore, we described remaining individual components of our framework and their functionality: the Adaptation Manager, which is the core of our framework, making all the adaptation decisions, the tenant-API, which allows tenants to express their monitoring requirements and translates them to AM-readable information, the Infrastructure Monitoring Probes, which are responsible for detecting dynamic-related events and notifying the AM and the Dependency Database which holds all necessary information regarding interdependent security devices. Each component's functionality contributes to an accurate adaptation decision.

Selected implementation details of two of our framework's components (the Adaptation Manager and the Infrastructure Monitoring Probes) were presented. In order to facilitate multiple adaptation decisions in parallel, the AM was implemented using a multi-threaded approach. Instead of using traditional network-based communication between different components, we opted for a faster file-based approach using the *Inotify* Linux utility. In order to obtain accurate and up-to-date VM-related information we made the design choice of placing the IMPs inside core modules of the cloud engine.

Two separate instantiations of our framework are discussed in the following chapters. The proposed instantiations focus on the adaptation of two different types of security devices. The first instantiation presents a self-adaptable network intrusion detection system called SAIDS, while the second instantiation presents a secure application-level introspection-based firewall called AL-SAFE.

Chapter 4

SAIDS: A Self-Adaptable Intrusion Detection System for IaaS Cloud Environments

In this chapter we present SAIDS the first instantiation of our security monitoring framework. SAIDS is a self-adaptable network intrusion detection system designed for IaaS cloud environments. A preliminary version of this contribution was published in [140]. We begin with a description of SAIDS objectives in Section 4.1, followed by the presentation of individual SAIDS components in Section 4.2. Security threats are discussed in Section 4.3. The adaptation process along with events that trigger the adaptation are featured in Section 4.4. Implementation details and our detailed evaluation plan along with obtained results are described in Sections 4.5 and 4.6 respectively. Finally, Section 4.7 summarises this chapter and presents key observations.

4.1 Objectives

In this section we discuss in detail the objectives that SAIDS should meet.

- **Self-Adaptation**: SAIDS should react to dynamic events that occur in a cloud environment and adapt the network intrusion detection devices accordingly. These events refer to topology-related changes in the virtual or hardware infrastructure and service-related changes. Virtual infrastructure changes are caused by tenant decisions regarding VM-lifecycle (i.e. creation, deletion) or provider decisions regarding VM placement (i.e. migration). Changes in the hardware infrastructure refer to addition or removal of servers. Service related changes refer to the addition or removal of services in the deployed VMs.

- **Customization**: based on the type of services that are hosted on the deployed VMs SAIDS should allow tenants to customise the events that are being detected. Tenants can request monitoring against specific types of threats that refer to different levels of their infrastructure (i.e. application, system or network level). Common threats (e.g. worms, SQL injection attempts) can be detected using generic rules out of public or commercial rule repositories [141]. SAIDS should provide tenants with the ability to use custom rules (i.e. tailored for their deployed systems) for common threats in order to improve detection quality. Furthermore, tenants should be able

to write and include their own customised IDS rules against more specific types of threats that target their deployed services.

- **Scalability**: the number of deployed SAIDS IDSs should adjust to varying conditions: load of the network traffic monitored, number of physical servers in the datacenter, number of VMs in the virtual infrastructure. SAIDS should be able to alter the resources available to its IDSs in the event of a degradation in the quality of detection. Different metrics are used in order to estimate the quality of detection for which SAIDS takes into account tenant-defined thresholds. SAIDS uses the following metrics: packet drop rate (the value of this metric can be improved by altering the computational resources available to the SAIDS IDSs), detection rate (the value of this metric is related to SAIDS IDSs packet drop since it also demonstrates the ability of an IDS to process the input stream without dropping packets, thus can indirectly be improved by altering the computational resources available to the SAIDS IDSs) and false positive rate.

- **Security and Correctness**: SAIDS should guarantee that an adequate level of detection is maintained during the adaptation of the SAIDS IDSs. The adaptation of the SAIDS IDSs should not allow attacks that otherwise would have been detected to remain undetected. Furthermore SAIDS should not create new security vulnerabilities in the provider's infrastructure.

4.2 Models and Architecture

In this section we present the system and threat model used in SAIDS along with a detailed description of SAIDS architecture and individual components.

We adopt the same system and threat model as the ones described in Chapter 3, Sections 3.2 and 3.3.

4.2.1 Architecture

This section describes SAIDS architecture. We first present a high level overview of SAIDS and then we focus on describing the functionality of each individual component.

SAIDS consists of four major components as depicted in Figure 4.1: the Local Intrusion Detection Sensors (LIDS), the Adaptation Worker (AW), the Master Adaptation Driver (MAD) and the Mirror Worker (MW). The LIDSs are deployed on dedicated nodes and our framework features one AW per LIDS (the AW is installed inside the LIDS). SAIDS features one MAD per dedicated node. Finally, we include one Mirror Worker per compute node.

4.2.1.1 Component Description

This section focuses on the description of each individual component's functionality. The components are run by the cloud provider.

4.2.1.1.1 Local Intrusion Detection Sensors: LIDS are used for collecting and analyzing network packets that are flowing through subsets of virtual switches. The detection technique that is used can either be signature- or anomaly-based. A signature-based technique has high true positive rate in detecting known attacks as opposed to an

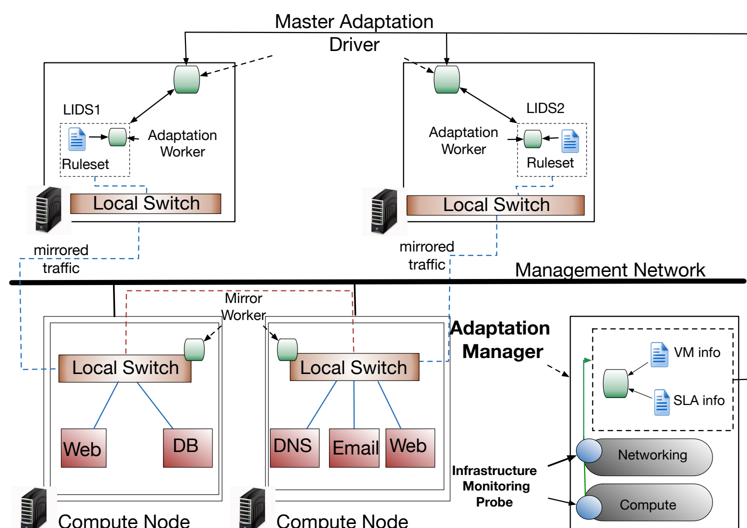


Figure 4.1 – SAIDS architecture

anomaly-based technique which is more effective in detecting unknown attacks. Furthermore, a signature-based LIDS requires zero training time making it a suitable choice for immediate efficiency in contrast with an anomaly-based LIDS which requires a training period. SAIDS supports both types of detection techniques allowing tenants to select their preferred trade-off. In a signature-based LIDS the packets are decoded and preprocessed in order to check their payload for suspicious patterns by comparing it with a preloaded set of rules. If a match is found the packet is logged and an alert is generated. The rules can match either service, network or system level threats. LIDSs organize rules in distinct sets named rule categories. Rules against variants of the same threat are organized in the same rule category. Once the category is included in the configuration file of a signature-based LIDS all the subsequent rules of that category are activated. All the logs from LIDS instances that are located on a given node are collected by a local log collector instance running on the same node.

4.2.1.1.2 Adaptation Worker: The AW is located inside the LIDS and has several roles: First, it is responsible for reconfiguring the enforced ruleset by reloading the new configuration file that was created by the MAD. Second, the AW can detect if the detection process has failed and restart it if necessary. Third, the AW periodically reports LIDS-specific monitoring metrics (e.g. packet drop rate) back to the MAD and ensures that during the reconfiguration process the LIDS continues to operate seamlessly, so an adequate level of detection is maintained. Finally, once the reconfiguration process has been completed successfully, the AW reports back to the MAD.

4.2.1.1.3 Master Adaptation Driver: A MAD is responsible for the reconfiguration and lifecycle of a group of LIDSs on a given node. In order to satisfy the scalability objective of SAIDS the MAD was designed for handling multiple reconfiguration requests in parallel. When a dynamic event occurs, the adaptation parameters are sent by the AM to the MAD. The MAD translates them to LIDS-specific rules and creates a new configuration file that contains the rules that need to be activated in the affected LIDS. In the event of a new LIDS is instantiated the MAD is responsible for creating an endpoint for

that LIDS on the local switch and reconfiguring the traffic distribution between LIDSs on the local switch.

The MAD periodically communicates with different AW instances in order to gain access to LIDS-specific performance metrics. In case of a performance degradation the MAD is responsible for deciding between instantiating a new probe or assigning more computational resources to an existing one. Finally, the MAD can periodically obtain resource utilization information about each LIDS. The time between two consecutive resource utilization queries can be defined by the tenant.

4.2.1.1.4 Mirror Worker: The MW has two different roles: First, it is responsible for checking whether the traffic that flows to and from a group of VMs that are hosted in a particular compute node is correctly mirrored to the corresponding LIDS node(s). Second if a mirroring endpoint does not exist the mirror worker creates it on the underlying local switch.

4.2.1.1.5 Safety Mechanism: SAIDS features a safety mechanism inside each compute node that guarantees that the VM participating in a dynamic event (e.g. a migrated VM) does not enter an active state before the corresponding LIDS has been successfully reconfigured. The AM notifies the safety mechanism that the LIDS reconfiguration has been completed successfully. Although SAIDS has this mechanism enabled by default, in our design we allow tenants to choose whether to disable it or not. The choice between enabling the safety mechanism or not demonstrates a trade-off between security and performance. Consequently, enabling the mechanism could impact the performance of network-critical applications that run inside the affected VM.

After presenting SAIDS individual components we now discuss potential security threats against SAIDS.

4.3 Security Threats

In this section we describe the potential vulnerabilities in SAIDS design and potential vulnerabilities added by SAIDS in the provider's infrastructure. We present our design choices for addressing each one.

4.3.1 SAIDS Configuration Files

The first type of input that is required for the adaptation of the LIDSs is a set of configuration files that are used for translating the adaptation arguments (which include any tenant-defined monitoring requests) to rule category names. The first file contains the adaptation arguments while the second file provides a mapping between specific types of tenant-deployed services and rule category names. An attacker could alter the contents of the files and create false adaptation arguments that would result in the activation of incorrect rule categories or deactivation of correct ones. These files are simple text or XML files for which SAIDS features robust parsers. The input file is pre-processed using a SAIDS-specific filter that verifies that only SAIDS-specific elements and printable ASCII strings without special characters are present in the files. Furthermore the value of each entry (i.e. monitoring request) partially matches the rule name (exact definition in Section 4.4.2), so any complex interpretation is avoided. Following up on the list of deployed services in the example of Section 3.5, the file containing the adaptation arguments after the adaptation decision can be found in Listing 4.1:

Listing 4.1 – The*adaptation_args file*

```
1 signature_based
2 suricata65
3 sshd 192.168.1.2, 192.168.1.3
4 apache2
5 sqld
6 worm
7 5
```

The format of the file is as follows: The first two lines are reserved for the LIDS type and the name of the LIDS while the last line is reserved for comma-separated numeric values of LIDS-specific metrics. In the simplified example of Section 3.5, the tenant has only one VM with three processes running (an ssh daemon and a SQL-backed Apache server) while he requests additional monitoring for worms and accepts a drop rate of 5% from the LIDS.

4.3.2 LIDS Rules

The result of the above translation leads to enabling specific rule categories in the LIDS. Since the rules are LIDS native, they are considered safe.

4.3.3 SAIDS Adaptation Sources

The adaptation process in SAIDS is based on specific arguments that describe dynamic events (e.g. for a VM migration SAIDS needs the VM ID, VM IP public and private addresses, source and destination node, etc). Since the arguments are extracted through the IMPs from inside the cloud engine and we assume that the provider's infrastructure is safe, we consider them safe.

4.3.4 Connection Between SAIDS Components

The Master Adaptation Driver defines the reconfiguration parameters based on adaptation arguments that it receives from the Adaptation Manager in a dedicated file. Interception of this file by an attacker could lead to false reconfiguration decisions. We establish and maintain a secure connection between the AM and the MAD. The secure connection is established through a secure protocol [142] which provides authentication of the AM and guarantees the integrity of the data transferred.

4.3.5 External Traffic

As all network-based intrusion detection systems, LIDSs can be corrupted by malicious production traffic that they analyze. SAIDS introduces a barrier between a potentially corrupted LIDS and the node hosting it by placing the LIDS in an isolated environment (e.g. a Linux container). Communication between LIDS and the local log collector instance is facilitated through shared volumes. Although this communication is not exposed to the network, a potentially corrupted LIDS can still produce malicious logs which could corrupt the local log collector instance and ultimately lead to false logs being transmitted to tenants. To contain the propagation of corruptions of the local log collector, we also place it in an isolated environment. In the event of a corrupted log collector instance, malicious input could be introduced in the log file of the LIDS. However, since the LIDS itself does not need to read log files, this is not a security issue for the LIDS.

4.4 Adaptation process

In this section we describe the events that trigger the adaptation of the LIDSs and the different steps of the adaptation process.

4.4.1 Events Triggering Adaptation

SAIDS adapts its components based on dynamic events that refer to three main categories:

1. **Virtual infrastructure topology-related changes**: this category includes tenant-driven (i.e. VM creation, deletion) or provider-driven (i.e. VM migration) changes.
2. **Hardware infrastructure topology-related changes**: addition or removal of physical servers. The changes in this category are exclusively provider-driven.
3. **Service-related changes**: addition or removal of services on the monitored VMs.
4. **Performance-related changes**: effects in the quality of detection or optimization decisions regarding computational resource utilization. The effects in the detection quality are detected through LIDS-specific detection quality metrics.

In Table 4.1 we classify these events based on their origin and subsequent adaptation action. The adaptation action varies depending on the current state of the monitoring

Table 4.1 – Events that trigger adaptation

Part A			*Part B*
Change category	Event	Origin	Adaptation action
Virtual infrastructure topology	VM creation VM destruction VM migration	Tenant Tenant Provider	{rule update, new LIDS} {rule update} {rule update, new LIDS}
Performance	% Packet drop Latency % unused resources	Traffic load	{new LIDS} {new LIDS} {destroy LIDS}
Service	Service addition Service removal	Tenant	{rule update} {rule update}
Hardware infrastructure topology	Server addition Server removal	Provider	{rule update, new LIDS} {rule update, destroy LIDS}

framework. For example, if a topology related change occurs (e.g. VM migration) SAIDS will check if a LIDS monitoring the traffic flowing towards and from the new VM location exists. If a LIDS exists SAIDS simply reconfigures the enforced ruleset (i.e. rule update action). If a LIDS does not exist then SAIDS instantiates a new LIDS. When a performance degradation occurs, SAIDS opts for a new LIDS instantiation.

4.4.2 Adaptation Process

We now describe the adaptation process for each one of the dynamic events described in the previous section. We focus only on the SAIDS-specific components and we omit the first stage of the adaptation that includes the notification from the Infrastructure Monitoring Probes and the adaptation decision from the Adaptation Manager. The actions performed by SAIDS during the adaptation process were designed in order to satisfy SAIDS self-adaptation and customization objectives. Throughout this section we use the adaptation file presented in Listing 4.1 (the adaptation file resulting from the simplified example scenario presented in Section 3.5).

4.4.3 Topology-Related Change

Once the Master Adaptation Driver (MAD) receives the adaptation parameters from the Adaptation Manager two steps are performed:

1. It checks whether the affected LIDS is running or not. If it is not running then the MAD starts a new LIDS and reconfigures the traffic distribution on the local switch of the node hosting the LIDS in order for the newly instantiated sensor to access the traffic flowing towards and from the affected VM.

2. The MAD translates the adaptation parameters to LIDS-specific configuration parameters and creates a new LIDS-specific configuration file. The configuration file contains the list of rule categories that need to be activated in the LIDS in order to successfully monitor the list of services running inside the affected VM. In our example the MAD partially matches the adaptation argument to the rule category name in order to find the right rule categories that need to be activated. A partial match is found when the adaptation argument is contained in the rule category name (e.g. *worm* in *emerging-worm.rules*). Consequently, for the *worm* adaptation argument the *emerging-worm.rules* category will be activated while for the *sqld* argument the *emerging-sql.rules* will be activated. In case a partial match is not found, MAD uses the second file from SAIDS input set (see Section 4.3), which is a LIDS-specific file, located in the MAD node, to translate the adaptation argument to rule category names. The file only features rule categories that can not be partially matched to the adaptation argument (e.g. *apache2* or *ssh*). A snippet of this file can be found in Listing 4.2:

   Listing 4.2 – The *userservice.conf* file

   ```
   1 mail   emerging-pop3.rules,emerging-smtp.rules
   2 apache2,nginx  http-events.rules,emerging-web_server.rules,emerging-
         web_specific_apps.rules
   3 sshd   emerging-shellcode.rules,emerging-telnet.rules
   ```

   In the newly created suricata configuration file the following rule categories will be activated: (a) *http-events.rules, emerging-web_server.rules, emerging-web_specific_apps.rules* for the web server, (b) *emerging-shellcode.rules, emerging-telnet.rules* for the ssh daemon and finally, (c) *emerging-sql.rules* for the SQL database. A part of the resulting LIDS configuration file can be found in Listing 4.3:

   Listing 4.3 – The *suricata.yaml* file

   ```
   1 #RULE BLOCK
   2 # - decoder-events.rules # available in suricata sources under rules
         dir
   3 # - stream-events.rules  # available in suricata sources under rules
         dir
   4  - http-events.rules    # available in suricata sources under rules
         dir
   5 # - smtp-events.rules    # available in suricata sources under rules
         dir
   6 # - dns-events.rules     # available in suricata sources under rules
         dir
   7 # - tls-events.rules     # available in suricata sources under rules
         dir
   8 # - emerging-user_agents.rules
   ```

```
 9 # -  emerging-voip.rules
10 # -  emerging-web_client.rules
11 # -  tor.rules
12   - emerging-web_server.rules
13   - emerging-web_specific_apps.rules
14   - emerging-worm.rules
15   - emerging-shellcode.rules
16   - emerging-telnet.rules
17   - emerging-sql.rules
```

3. As a last step, the MAD notifies the AW, which is locally installed inside the LIDS, that a new configuration file exists and the IDS needs to be reconfigured. Upon receiving the notification, the AW checks whether the detection process is running and initialises a reload of the newly created configuration file. Once the reload is complete (i.e. the LIDS has been adapted) the AW notifies the MAD that the adaptation process was completed successfully. In case the AW notifies the MAD that the adaptation process failed, for example due to a crash of the detection process or an unsuccessful reload of the enforced ruleset, the MAD propagates the event to the AM which then notifies the safety mechanism that the VM should not yet be resumed in the new location. Depending on the type of failure the following strategy is adopted: first, the AW will try to restart the detection process (or reload the enforced ruleset in the event of a reload failure). If it fails, it propagates the information to the MAD, which in turn instantiates a new LIDS, reconfigures traffic distribution appropriately and destroys the failed LIDS instance. The number of tries that the AW will execute before a new LIDS needs to be instantiated are decided by the MAD. The AW guarantees that during the reconfiguration phase, the LIDS will continue to operate seamlessly, thus no traffic remains uninspected.

4. Finally, the MAD notifies the Adaptation Manager that the adaptation request was served. The AM in turn notifies the safety mechanism that the VM can be safely resumed.

4.4.4 Traffic-Related Change

In order to detect degradation in the performance of an LIDS the MAD periodically queries the AW for LIDS-specific performance metrics (e.g. packet drop rate). Once the performance metric exceeds a predefined threshold, the MAD instantiates a new LIDS, with identical configuration parameters, and reconfigures the traffic distribution on the local switch so that the load is balanced between the two LIDSs. Currently the MAD can redistribute traffic load only on a VM basis (i.e. send all the traffic from and to a particular VM to a specific LIDS).

4.4.5 Service-Related Change

The adaptation process is the same as a topology related change. Since SAIDS does not feature any mechanism for automatic discovery of new services in the deployed VMs, we rely on the tenants in order to notify SAIDS for service-related events (through our framework's dedicated API).

So far the description of the adaptation process focuses on the side of the monitoring probes. Although LIDS reconfiguration is essential for preserving an adequate level of detection in the virtual infrastructure, gaining access to the right portion of the traffic is

also required. Each time a topology-related change occurs (e.g. VM creation or migration), the Mirror Worker is responsible for checking whether a traffic endpoint from the local switch on the compute node to the local switch of the IIDS node exists, and if not creates it. This strategy applies to hardware-related changes as well.

4.5 Implementation

We have implemented a prototype of SAIDS from scratch using the KVM [27] hypervisor on a private cloud. Our cloud was deployed on OpenStack [32] and we used Open vSwitch (OvS) [137] as a multilayer virtual switch. To segregate VMs that belong to different tenant networks we utilised Generic Routing Encapsulation (GRE) tunnels. A span tunnel endpoint was created for mirroring traffic in the virtual switches to the LIDSs node. In this section we discuss the main implementation aspects of each SAIDS component.

Local Intrusion Detection Sensors: we deploy each LIDS inside a dedicated Docker [45] container. Since the LIDS only runs the detection process and does not require a full operating system, we opt for containers in order to achieve minimal start time. Containers are also a suitable lightweight solution for achieving isolation between different detection processes. Currently our prototype features 2 different legacy network IDSs: Snort [93] and Suricata [94]. Each container hosts an IDS process and an Adaptation Worker responsible for managing that process. For providing access to the mirrored traffic for the LIDSs we use the *ovs-docker* utility. *Ovs-docker* allows docker containers to be plugged on OvS-created bridges. It interacts with the virtual switch on the node hosting the LIDSs and creates one network tap per container. We select signature-based LIDSs as they are the ones requiring zero training time. We utilise OpenFlow [48] rules for distributing traffic between LIDSs. Depending on the monitoring strategy selected (e.g. one LIDS monitoring the traffic that flows towards and from a particular compute node), the traffic is distributed based on the tunnel source address of the GRE tunnel transferring the monitored traffic.

Adaptation Worker: We have created a unified version of the AW that is able to handle the signature-based LIDSs that are supported in our prototype (i.e. Suricata and Snort). The AW communicates with the Master Adaptation Driver for receiving reconfiguration requests and reporting back on the reconfiguration status using a shared folder. The AW places the shared folder under surveillance for specific events (file creation and modification) using the *Inotify* Linux utility [139], a tool for detecting changes in filesystems and reporting them back to applications. Once the event is triggered the AW loads this new configuration file (so the new ruleset can be enforced) and calls the *live_rule_swap* functionality available in both Suricata and Snort IDSs in order to live update (i.e. without having to restart the LIDS) the enforced ruleset. The *live_rule_swap* operation allows a user to update the enforced ruleset without stopping the IDS itself (a SIGUSR2 is sent to the detection process). MAD relies on this functionality, consequently the LIDS remains operational even during the actual reconfiguration. The AW ensures that the new ruleset has been loaded by continuously monitoring the log file for a log indicating that the new ruleset has been reloaded. Once the reload is complete the AW notifies the MAD by creating a dedicated file in the shared folder. The AW was implemented in Python.

Master Adaptation Driver: For enabling managing the lifecycle and reconfiguration of multiple LIDSs MAD was implemented using a multithreaded approach. MAD creates a unique folder per LIDS and uses a dedicated thread to watch this folder for changes (again using the *inotify* utility). Once an adaptation request arrives from the AM (i.e.

a file containing the adaptation parameters is created in the watched folder) the thread starts the reconfiguration process. The MAD features IDS specific configuration files for translating the adaptation parameters to rule categories. If the LIDS is not started yet, the thread starts it, creates a port for it on the virtual switch using the *add-port* command from *ovs-docker* and finally redirects the appropriate mirrored traffic to the created port. The last part is done by creating a dedicated OpenFlow rule that redirects the traffic from the GRE tunnel endpoint to the LIDS port.

For tracking the resource consumption of each LIDS sensor the MAD features a special function called *docker_stats_monitor*. First, it obtains the container's ID. Then it periodically queries the *cgroup* of that particular ID for different runtime metrics: CPU, I/O and Resident Set Size memory. The MAD also inspects externally the packet drop rate for a particular LIDS container by collecting interface level packet drop count from inside the container namespace. The MAD was implemented in Python.

Mirror Worker: It checks whether a GRE tunnel for mirroring the traffic flowing towards and from a group of VMs to the corresponding LIDS exists. If not the MW creates it. The IP of the LIDS along with the VMs IDs and the port name of the VM on the destination node are sent by the Adaptation Manager. Once the AW receives the OvS port name, it uses the *list_interface* OvS command giving the port name as input in order to extract the port's id. The MW was implemented in Python.

Safety Mechanism: we implement the safety mechanism by placing a dedicated hook inside the *plug_vifs* Nova function which is executed on compute nodes. The *plug_vifs* function is responsible for creating the virtual interface for the VM on the OvS bridge of the destination node. The hook halts the virtual interface creation until the LIDS reconfiguration has been completed. By placing the hook inside the function we make sure that network connectivity for the VM is not enabled until the adaptation is complete. We select the *plug_vifs* function because it is executed in both VM creation and migration events. The safety mechanism was implemented in Python.

4.6 Evaluation

After presenting the most important implementation aspects of SAIDS we now present the evaluation of our prototype. We first detail the objectives of our evaluation plan along with our experimentation methodology. Finally, we discuss the obtained results along with limitations.

4.6.1 Objectives of the Evaluation

The main goal of SAIDS is to adapt the LIDSs while guaranteeing an adequate level of security, combined with adequate performance (in terms of reaction time for a full adaptation loop) and minimised cost both for tenants and the provider. We now detail the factors that affect each objective.

4.6.1.1 Performance

The performance objective refers to two different aspects: adaptation speed and scalability.

4.6.1.1.1 Adaptation Speed: Here we refer to the time required for SAIDS to perform a full adaptation loop, from the moment a dynamic event occurs until all involved LIDSs are successfully reconfigured. In order to have an exact calculation of the overall time we need to answer the following questions:

1. *What are the different SAIDS components that are involved in each adaptation loop?* Five SAIDS components are mandatorily involved in each adaptation loop: the Adaptation Manager, the Master Adaptation Driver, the Adaptation Worker, the Mirror Worker and the safety mechanism. Obviously, the overall time depends on the different tasks that each component has to complete.

2. *What are the tasks performed by each component?*

   - Adaptation Manager: makes the adaptation decision and sends the adaptation arguments to the Master Adaptation Driver.
   - Master Adaptation Driver: checks if the LIDS container is running and depending on the outcome, directly proceeds in generating the adapted configuration file or first starts a new LIDS container and configures traffic distribution.
   - Adaptation Worker: conducts the live rule update in the LIDS container.
   - Mirror Worker: checks whether a traffic endpoint from the compute node hosting the VM to the node hosting the LIDSs exists and if not creates it.
   - Safety Mechanism: guarantees that in the case of a VM creation or migration the VM does not enter an active state until the reconfiguration of the LIDS has been completed successfully.

   Different factors affect the completion time of each task, which leads us to the next question:

3. *Which factors affect the execution time of each task?*

   - Adaptation Manager: the number of the adaptation arguments affects the size of the file and consequently the time required to send it to the MAD on the LIDS node. The number of the adaptation arguments depends on the number of services running inside the monitored VMs and the number of additional monitoring rules that the tenant has requested.
   - Master Adaptation Driver: the number of rules that need to be activated/deactivated affects the time required to regenerate the LIDS configuration file. The time required for the remaining tasks is not affected by the adaptation arguments.
   - Adaptation Worker: the number of rules that are added affects the overall time required to reload the enforced ruleset.
   - Mirror Worker: since the MW needs to create a single tunnelling endpoint (which translates to executing two OvS commands, one for identifying the port number of the VM's port and one for creating the tunnel itself) the MW execution time is expected to be constant.
   - Safety Mechanism: the waiting time introduced by the safety mechanism in resuming the VM is equal to the time remaining to complete the adaptation process when the Nova function *plug_vifs* is called on the VM destination node. Consequently, the factors that affect the completion time of the four other SAIDS components indirectly affect the execution time of the safety mechanism.

   We now present the second performance objective.

4.6.1.1.2 Scalability: We want to evaluate how many adaptation requests SAIDS can successfully serve in parallel. In order to achieve this we need to answer the two following questions:

1. *How many full adaptation loops can SAIDS handle in parallel?* Each loop is composed of three different levels: The Adaptation Manager, the Master Adaptation Driver and finally the Adaptation Worker with the LIDS (the level of the AW does not scale since the design pairs a single AW with a single LIDS). The evaluation of the overall scalability of SAIDS should be composed of the scalability evaluation of each one of the adaptation levels. Consequently, we need to calculate: (a) *How many MADs can the Adaptation Manager handle in parallel?* This is the scalability result of the first level of adaptation (from the AM to different MADs). To achieve this we calculate the maximum number of MADs that the AM can handle in parallel. For this phase we only vary the number of MADs. (b) *How many LIDSs can a MAD handle in parallel?* This is the scalability result of the second level of the adaptation (from a MAD to the LIDSs). To achieve this we need to consider the case where the number of tasks that a MAD needs to perform per LIDS is maximized. This case essentially requires the MAD to spawn a new LIDS and configure the traffic distribution on the local switch, for each adaptation request. We examine only this case as the one requiring the maximum effort on the MAD side. Since the focus of the experiment is on creating new LIDSs, rather than reconfiguring the enforced ruleset of existing ones, we only activate one rule category per IDS. The number of rule categories that are activated does not change the size of the LIDS configuration file (see example in Listing 4.3) thus the time required for the MAD to generate it is not impacted. Moreover, since the MAD operations are asynchronous, the time required to load the rules in each LIDS does not affect the MAD scalability. For this phase of the experiment we only vary the number of LIDSs.

2. *What is the overhead imposed by the multiple parallel requests in the execution time of each adaptation loop?* We would like to identify the impact of parallelism on the time required to complete each adaptation loop. The reaction time of two SAIDS components (i.e. Adaptation Manager and Master Adaptation Driver) is directly affected by the number of parallel requests. We compute the overhead (in seconds) in the reaction time of the two components.

4.6.1.2 Cost

We examine the associated penalties on deploying SAIDS both from the tenants and the provider's objective. From the provider's perspective we calculate the overhead imposed by SAIDS to normal cloud operations (e.g. VM migration) while for the tenants we examine if SAIDS imposes any overhead in the performance of tenant applications.

- **Provider-side cost**: namely, *What is the overhead (in seconds) introduced by SAIDS to a normal cloud operation like a VM migration?*

- **Tenant-side cost**: since SAIDS monitoring is performed by network based IDSs that work on mirrored traffic, SAIDS deployment does not directly affect tenant applications regardless of their profile (no latency is induced in the production network). The traffic mirroring itself can indirectly affect the applications running on the SAIDS-monitored node due to CPU consumption and physical network bandwidth usage (although this penalty is inherent of the mirroring technique and not

SAIDS itself). The only SAIDS related cost on individual tenant applications is related to the VM downtime when normal cloud operations occur.

4.6.1.3 Security and Correctness

Since one of the main SAIDS objectives is to guarantee an adequate level of detection during the adaptation time, it is clear that we need to examine whether malicious traffic is successfully identified even when the LIDSs are being reconfigured. Furthermore, we need to certify that SAIDS does not affect the detection capabilities of the adapted LIDSs and that the adaptation result is correct. We focus on the following questions:

- *Are the added rules correct and operational?*
- *Are there any packets dropped during the adaptation time?*
- *Can SAIDS detect an attack that occurs during the adaptation time?*
- *Does SAIDS add any security flaw in the adaptation process itself or in the provider's infrastructure?* in Section 4.3 we have already justified why our design choices do not add any flaws in the adaptation process and in the provider's infrastructure.

After presenting the objectives of our evaluation process, we now detail the experimental scenarios used to perform the evaluation of our SAIDS prototype.

4.6.2 Experimentation Methodology

This section describes in detail the experimental scenarios used in order to evaluate SAIDS prototype. The scenarios were designed for addressing multiple evaluation objectives simultaneously. We select VM migration as a representative cloud operation that includes VM creation and deletion. For examining the security and correctness of SAIDS, we select a web server as use case.

4.6.2.1 VM Migration

The VM migration scenario simultaneously addresses the performance and cost objectives (only the provider-associated cost of deploying SAIDS). We aim at calculating the overhead imposed by deploying SAIDS in a VM migration. In this scenario we calculate the migration time of a monitored VM under two different workload cases: 1. an idle VM, no workload running in the migrated VM (idle VM) and 2. a memory-intensive workload running in the migrated VM. The overall migration time depends on two factors: the memory size of the migrated VM and the workload running inside the migrated VM. The workload cases represent two different situations, the first one (i.e. idle VM), with minimum migration time, consequently any overhead imposed by SAIDS is maximised while the second one (i.e. memory intensive workload), with maximum migration time, hence any overhead imposed by SAIDS is minimised. In both cases we examine all possible adaptation options:

- a corresponding LIDS already exists and is running on a dedicated node, thus SAIDS only needs to reconfigure the enforced ruleset.
- SAIDS needs to start the corresponding LIDS, create a port for it on the virtual switch, and redirect the mirrored traffic coming from the destination node of the VM to the LIDS port. Furthermore, SAIDS needs to check whether a tunnel for the

mirrored traffic from the destination node of the VM to the LIDS node exists and if not create it.

In each option we calculate the reaction time of each SAIDS component.

4.6.2.2 Multiple LIDSs and Multiple MADs

This scenario focuses on the scalability objective of our evaluation plan. The multiple LIDSs and multiple MADs scenario examines the ability of SAIDS to handle multiple adaptation requests in parallel. SAIDS's scalability is examined at two different levels: the Master Adaptation Driver and the Adaptation Manager. At the Master Adaptation Driver level, we calculate the total reaction time as well as the reaction time of each phase (ruleset configuration, LIDS creation, traffic distribution). We compare the results with the adaptation of a single LIDS and calculate the scalability overhead. The only varying parameter in this experiment is the number of LIDS.

For the Adaptation Manager level we calculate how many different Master Adaptation Drivers (each one with maximized load) an can AM handle in parallel. Each MAD resides in a different node and requires a dedicated secure connection in order to transmit the adaptation arguments. We calculate the mean reaction time of the AM and we compare it with a single MAD approach in order to calculate the scalability overhead.

For the evaluation, we simulate a large number of nodes using containers and we place each MAD in a separate container with a dedicated IP address. All containers are placed on the same physical node. Since each container is a completely isolated environment, the AM perceives it as a dedicated node and still needs to create a dedicated secure connection per MAD. Due to memory restrictions (our node has 24GB of memory) no LIDS is run inside the containers. Since the MAD operations are asynchronous the fact that no LIDS is run does not affect the result.

In SAIDS, an adaptation request concerning a single LIDS is represented by a file containing the adaptation arguments (one file per LIDS is sent from the AM to the MAD responsible for the adapted LIDS). Consequently, in order to simulate the maximum number of adaptation requests per MAD, we take the results from the first phase of the experiment (i.e. the maximum number of LIDSs that a single MAD can handle) and we send the same number of files containing adaptation arguments to each MAD. The varying parameter in this experiment is the number of MADs.

4.6.2.3 Web Server

In this scenario we examine SAIDS ability to guarantee an adequate level of detection even during the adaptation process. For this purpose we migrate a web server and we launch multiple SQL injection attacks during the migration period. In the set up created for this scenario we have two different LIDSs (one monitoring the traffic in the source node and one monitoring the traffic in the destination node). The first LIDS is already configured to detect SQL injection attacks while the second one is not. We expect that the second LIDS will be able to detect the attacks after SAIDS adapts it. Depending on when in the migration phase the attack packets reach the victim VM we expect different outcomes.

Before presenting the different outcomes we briefly discuss the migration aspect that affects the connectivity of the migrated VM. In each live migration the dirty memory pages of the migrated VM are copied from the source to the destination node until a specific threshold is reached, when the VM is momentarily paused the remaining memory pages are copied and then the VM is resumed at the destination node. Until this threshold

is reached the VM continues to be active on the source node, thus the virtual interface accepting VM-related traffic is the one on the source node (consequently in our case it will be monitored by the first LIDS). In parallel with the memory pages copy, a new virtual interface for the VM is created on the destination node. After the interface is created and the copy of the pages reaches the threshold, the VM is activated on the destination node, thus the traffic is now redirected on the new virtual interface (consequently in our case it is monitored by the new LIDS).

We now list the three different outcomes:

1. Attack packets reach the VM before the virtual interface has been created at the destination node. Consequently, the packets will be inspected by the first LIDS. We expect the attack to be detected since the LIDS is already configured.

2. Attack packets reach the VM after the virtual interface has been created at the destination node and SAIDS has successfully reconfigured the second LIDS. We expect the attack to be detected since the second LIDS is already reconfigured.

3. Attack packets reach the VM after the virtual interface has been created at the destination node and SAIDS reconfiguration is on-going on the second LIDS. Since SAIDS utilises the *live_rule_swap* functionality of a LIDS we expect the second LIDS to analyze the attack packets as soon as the new ruleset has been reloaded (the alert will be generated once the new ruleset is enforced and the attack packets reach the second LIDS).

SAIDS features a safety mechanism that does not allow the VM to enter an active state after migration (i.e. on the destination node) before the LIDS reconfiguration has been completed. The safety mechanism guarantees that no packets will reach the VM before the new LIDS is successfully reconfigured.

Furthermore, for checking whether SAIDS causes the LIDS to drop packets during the adaptation process, we compare the number of packets reaching the virtual interface of the LIDS with the number of packets that the LIDS reports as captured.

4.6.3 Result Analysis

After presenting our evaluation scenarios and the objectives that they serve we now analyze the obtained results.

4.6.3.1 Experimental Setup

To do our experiments, we deployed a data center on the Grid5000 experimentation platform. Our datacenter has 5 physical nodes: one controller, one network node, two compute nodes and one separate node for hosting the LIDSs. Each physical node has 24GB of RAM and features two AMD Opteron processors (1.7Ghz, 4 cores each). The nodes run an Ubuntu Server 14.04 operating system and are interconnected through a 1Gb/s network. The LIDSs gain access to the monitored traffic through mirror ports and GRE tunnels. The LIDSs in all experiments run a Suricata NIDS process. All the VMs deployed on the physical nodes run an Ubuntu server 14.04 Operating System with 2 CPUs and 4 GB of RAM. We perform 10 executions per experiment.

4.6.3.2 VM Migration

To generate the memory-intensive workload we utilised *bw_mem_wr* from the LMBench benchmark suite [143] with a 1024MB working set. The working set is allocated, zeroed and then written as a series of 4 byte integers. In each adaptation we only add two new rule categories that correspond to ssh traffic (*emerging-shellcode.rules, emerging-telnet.rules*). Since the VM is not executing a workload that generates traffic no other rules are necessary. In this scenario we aim at proving that SAIDS imposes negligible overhead in the VM migration. The results are shown in Figure 4.2. The imposed overhead in both cases

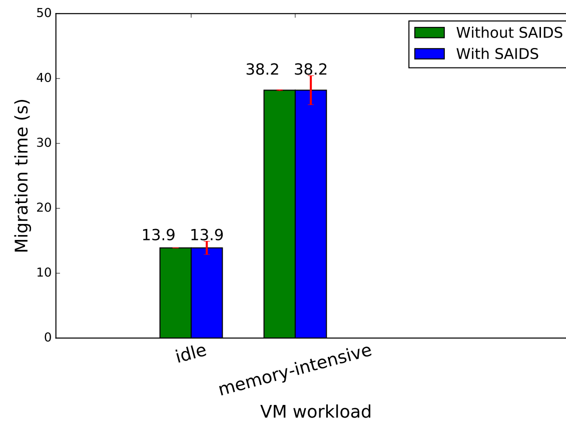


Figure 4.2 – Migration time with and without SAIDS

(idle VM and VM with memory intensive workload) is 0.0s which validates our initial hypothesis that SAIDS imposes negligible overhead on typical cloud operations. A per phase breakdown of the two different adaptation cases (i.e. ruleset reconfiguration only and new LIDS with traffic distribution) is shown in Figures 4.3 and 4.4. In both cases

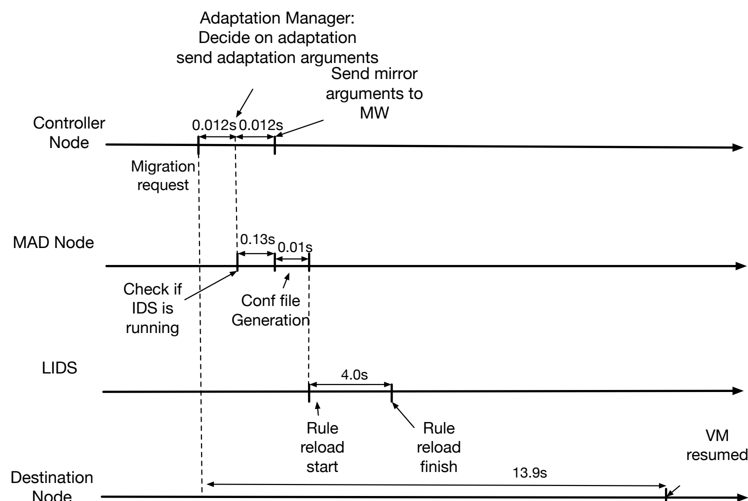


Figure 4.3 – Adaptation time breakdown when SAIDS only reconfigures the enforced ruleset inside the LIDS

the safety mechanism is enabled but the LIDS reconfiguration is completed much earlier than when the *plug_vifs* is called (4.14s and 0.97s respectively while the *plug_vifs* function is called always after the 10th second). Consequently no waiting time for resuming the

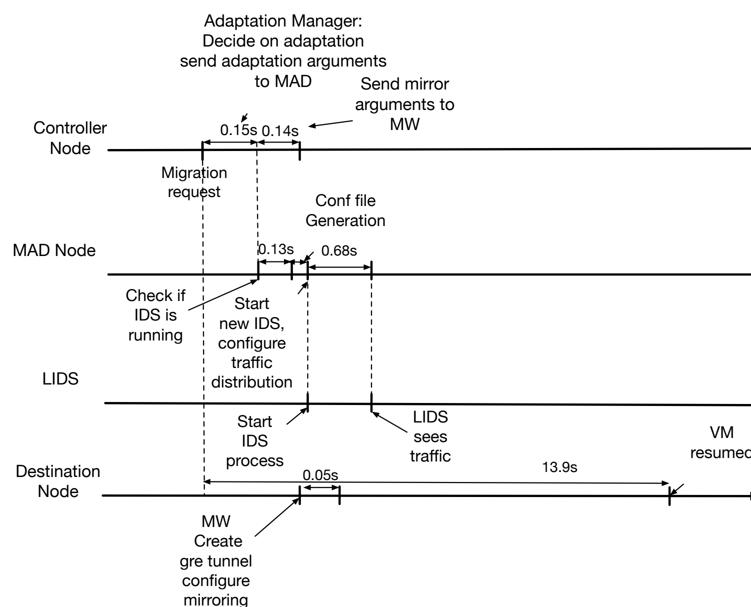


Figure 4.4 – Adaptation time breakdown when SAIDS has to start a new LIDS, distribute traffic and create a mirroring tunnel

VM is introduced. In the first case, where only a reconfiguration of the enforced ruleset is required, the time until the new ruleset is loaded is 4.14s (the MAD starts the reconfiguration process as soon as it receives the adaptation arguments). The AM uses the existing connection in order to send the file with the adaptation arguments thus we include only the time to send the file in the overhead analysis. In the second case, where a new LIDS needs to be instantiated, the time required until it gains access to the traffic is 0.97s (time for the MAD to start the LIDS and reconfigure traffic: 0.82s + time for the AM to send the adaptation arguments: 0.15s – connection establishment + file transmission time). The creation of the tunnel endpoint in the VM destination node takes 0.19s (including the time required for the AM to send the information to the AW which contains connection establishment and file transmission time). The overall time required for SAIDS to perform a full adaptation loop in both cases, is much smaller than the overall migration time (13.9s for an idle VM and 38.2s for a VM with a memory intensive workload). Furthermore, reconfiguring an existing IDS is a much heavier operation than starting a new one. This is due to the fact that during the reconfiguration process the AW needs to wait until the *live_rule_swap* is complete, which, depending on the number of newly added rules and potential LIDS delay in flushing its logs, can be time consuming.

4.6.3.3 Multiple MADs and Multiple LIDSs

In order to create multiple adaptation events in parallel, we wrote a dedicated script that simulates migration events by generating the same arguments that are sent to the Adaptation Manager by the Infrastructure Monitoring Probe in case of a VM migration (VM public IP, VM private IP, source and destination node, port on the virtual switch of the destination node).

4.6.3.3.1 MAD Scalability – Multiple LIDSs: During the first phase of our experiment we focus only on a single Master Adaptation Driver and compute the maximum

number of LIDSs that it can handle in parallel. The setup of a single MAD instance handling multiple LIDS is depicted in Figure 4.5.

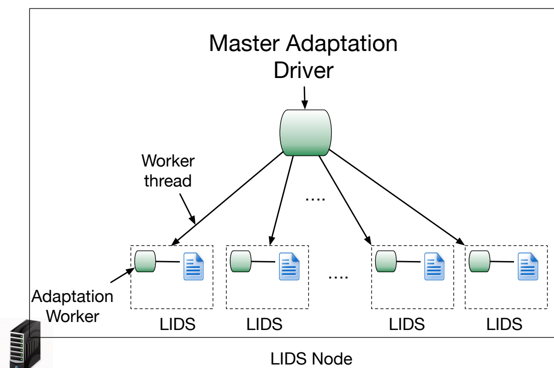


Figure 4.5 – MAD scalability setup

Our results show that a single MAD instance located in a dedicated node with 24GB of RAM can handle up to 50 LIDS (each LIDS requires 460.1MB of RAM consequently 50 is the maximum number of LIDS that the physical node of our testbed can handle before it's memory capacity is reached). The average response time of the MAD agent under different LIDS load is shown in Figure 4.6.

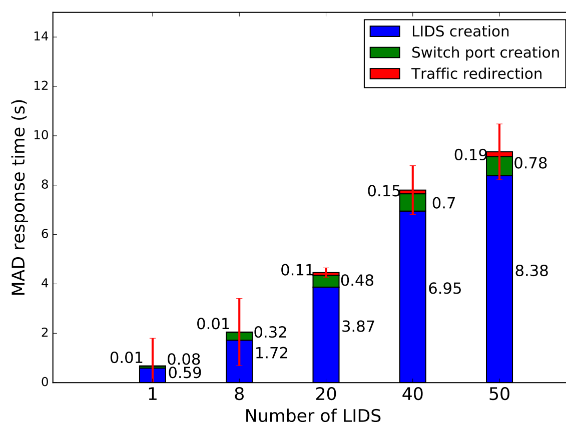


Figure 4.6 – MAD response time

From the obtained results we identify that the task of spawning a new LIDS container, which implies interacting with the Docker daemon, is the most time consuming task. Even with 50 parallel LIDS spawning requests, which represent the maximum number of Suricata containers that our physical node can accommodate, the mean overall reaction time for SAIDS under maximum load is 9.41s, which is still significantly lower than the 13.9s average migration time for an idle VM (see experiment Described in Section 4.6.3.2). Consequently, even if one of the 50 LIDS that are adapted is responsible for monitoring the traffic flowing towards and from the migrated VM, still no overhead will be introduced in the VM migration (the LIDS will be instantiated before the migration is completed). Note that in the breakdown of the MAD phases, we did not include the time required for the MAD to produce the new LIDS configuration file and check whether a new LIDS is running, since their effect in the overall time is negligible (see explanation in Section 4.6.2.2).

In a production environment, a usual deployment scenario includes assigning one core per LIDS in order to maintain an adequate performance level (in terms of packet loss) for the detection process. For simulating a production setup we tested SAIDS with 8 parallel adaptation requests (our machine has 8 cores). The mean overall time for MAD was 2.08s with individual breakdown of: LIDS creation 1.72s, switch port creation 0.32s and traffic redirection 0.01s.

4.6.3.3.2 Scalability of the AM – Multiple MADs: After obtaining the maximum number of LIDSs that a single MAD instance can handle in parallel in our testbed (50 LIDS – 460.1 MB of RAM per LIDS in a node with 24GB of RAM) we now study the scalability in the response time of the Adaptation Manager. In our experiment, each AM worker thread needs to adapt all the LIDS belonging to a single MAD (50 LIDS). We instantiate up to 100 AM worker threads. The setup of a single AM instance handling multiple MADs is depicted in Figure 4.7.

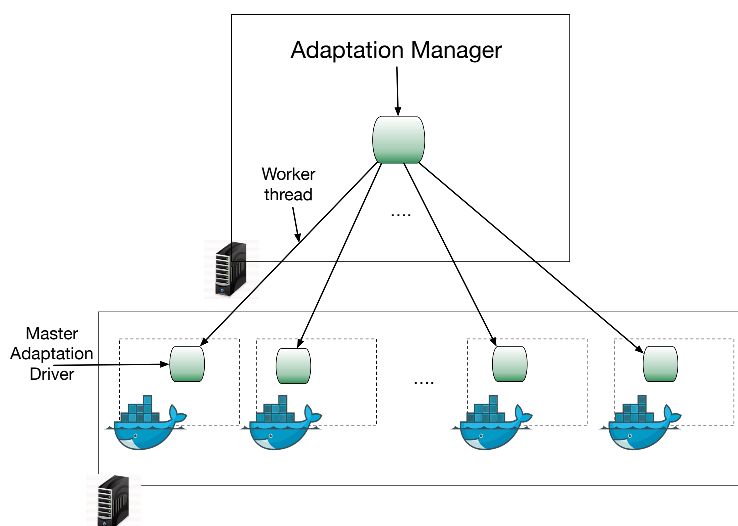


Figure 4.7 – AM scalability setup

In this scenario, the monitoring strategy selected assigns a single LIDS for monitoring the traffic flowing towards and from a single VM (although this strategy is not optimal in terms of provider-side costs we apply it for the scalability study). Consequently, in order to generate the adaptation requests for the 50 LIDS of each thread, we use our script to simulate 50 dynamic events (e.g. VM migrations) for 50 different VMs. In order to target the LIDS that belong to the same MAD instance that a worker thread is handling, all the VMs of a worker thread are migrated to the same destination node. In order to extract the arguments for each one of the 50 VMs that it is handling the worker thread needs to parse the file where all the VM-related information is stored (*vm_info.xml*). For generating enough tasks for the worker threads the minimum number of VM entries in this file is computed as follows: *maximum number of AM worker threads* $\times$ *number of VMs per thread*. In this scenario we instantiate up to 100 AM worker threads consequently the minimum number of entries in the *vm_info.xml*: $100 \times 50 = 5000$. The arguments for the adaptation of each LIDS are written to a separate file (see an example in Listing 4.1, *adaptation_args.txt*). Each file has a size of 219 bytes.

Then, the worker thread opens a single secure connection and sends all 50 files (one per LIDS) to the MAD responsible for the 50 LIDS. Finally, the worker thread opens a

secure connection with the destination node of the migrated VMs and sends the necessary information in a file to the MW. Note that since in our simulation all VMs of a single worker thread are migrated to the same compute node, only one file is needed. Indeed, the target of this experiment is not to evaluate the scalability of the AM with respect to the number of compute nodes. This optimization allows us to gain a better insight in the scalability of the AM with respect to the number of MADs.

The results are presented in Figure 4.8. As the results demonstrate, the phase that

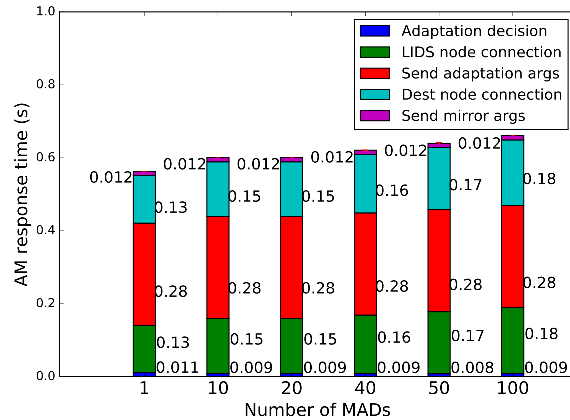


Figure 4.8 – AM response time

is most affected by increasing the load of the MADs for the AM is the establishment of the secure connection. That is due to the fact that each MAD is located in a different container with a different IP address consequently a separate secure connection is necessary (multiplexing is not possible). We measure the time to send the adaptation arguments (i.e. essentially the time required to send the 50 adaptation files) on the AM side. Since we do not wait for confirmation from each MAD instance that it received the files, no delay due to network contention is observed in the result. However, since all MAD instances are essentially run on different containers on the same node, some delay in the ssh connection establishment due to the number of processes running on the node could be observed. The latter makes the result of our experiment a pessimistic outcome compared to a real world scenario where each MAD instance would be run in a separate less loaded node. Since the VM-related information for all the VMs is located in a single file the multi-threading approach does not significantly decrease the adaptation decision time (as opposed to the case of one file per VM, where each worker thread needs to parse a file with only one entry instead of 5000).

Our results demonstrate that a single AM instance can handle up to 5000 LIDS instances while the per-thread response time remains under 1s. The limit in the number of LIDS instances results only from the memory capacity of the testbed used to conduct our experiments. The number of instances could be increased, if SAIDS is deployed in a different setup where the memory capacity of production nodes is significantly larger than 24 GB of RAM per node.

For computing the resource consumption of an AM in terms of CPU and memory handling multiple MADs we used the *pidstat* tool from the *sysstat* suite [144], a tool used for monitoring the resource consumption of a specific task running in an OS. In each experiment we ask the first worker thread to launch *pidstat* immediately after it receives the adaptation arguments and we terminate the monitoring after the last worker thread has completed its tasks. With this strategy we make sure that we only compute

the resource consumption of all the worker threads during the actual adaptation process. Since all the adaptation tasks in each adaptation request are performed by the worker thread responsible for that adaptation request, no other SAIDS-related process consumes resources. We set the monitoring interval at 1s. The results are shown in Table 4.2.

Table 4.2 – Resource consumption of the AM component

Number of MADs	*Usr%*	*Sys%*	*CPU%*	*Memory (MB)*
10	17.19	2.29	19.57	188.88
20	23.20	3.26	26.46	188.81
40	25.0	3.60	29.40	188.69
50	26.93	3.76	30.69	188.31
100	28.4	3.97	32.43	188.93

The increase in the CPU usage when the number of AM worker threads increases is due to the fact that starting a new *ssh* session imposes an one-time CPU penalty (during the connection establishment due to the cryptographic key exchange). Our measurements compute the worst-case scenario for each worker thread which is to establish a new connection. The CPU usage is expected to decrease in average-case scenarios where SAIDS needs to reconfigure an existing LIDS, thus it can use an already established connection for sending the file containing the adaptation arguments.

4.6.3.4 Correctness Analysis

For the web server scenario we installed WordPress on the target VM and we used Metasploit suite [145] for launching SQL injection attacks. We have created our own custom SQL injection rule which is included in the *local.rules* file (this file stores the user-defined rules in both Snort and Suricata LIDS). A snipet of the file can be found in Listing 4.4:

Listing 4.4 – The *local.rules* file

```
1 alert tcp any any -> $HOME_NET any (msg:"WP Sql Injection Attack";content
      :"INSERT INTO wp_users"; sid: 1000017; rev:1;)
```

The first LIDS, which monitors the traffic flowing towards and from the source compute node is configured to detect SQL injection attempts (the custom rule is activated), while the second LIDS, which monitors the traffic that flows towards and from the destination node, is not configured (the custom rule is deactivated). In order to cover all three possibilities for the arrival time of the attack packet (before the virtual interface migration – attack packets are processed by the old LIDS, after the virtual interface migration but before the new LIDS reconfiguration and finally after the virtual interface migration and after the LIDS reconfiguration) we launch 10 consecutive attacks at the beginning of the VM migration.

4.6.3.4.1 Attack packets arrive before the creation of the virtual interface of the target VM on the destination node: In this case the traffic is processed by the first LIDS, so the attack is detected and an alert is generated.

4.6.3.4.2 Attack packets arrive after the creation of the virtual interface of the target VM on the destination node and after the second LIDS has been successfully reconfigured by SAIDS: In this case the enforced ruleset in the second

LIDS is already reconfigured to include the custom SQL injection signature, so the attack is detected and an alert is generated.

4.6.3.4.3 Attack packets arrive after the creation of the virtual interface of the target VM on the destination node but before the second LIDS has been successfully reconfigured by SAIDS: In our strategy, the LIDS reconfiguration starts immediately after the migration command is received by the cloud API and is executed in parallel with the migration. A full adaptation cycle from SAIDS requires either 4.14 (existing LIDS reconfiguration) or 0.97s (new LIDS deployment) while the migration of the target VM requires in the best case scenario (idle VM) 13.9s (see experiment described in Section 4.6.3.2). In this case the migration of the virtual interface of the target VM (executed by the *plug_vifs* function) occurs always after the 10th second in the migration cycle. As a result, the second LIDS reconfiguration has been completed before the migration of the virtual interface of the target VM occurs. Consequently, the SAIDS adaptation cycle has already been completed and the LIDS has already been reconfigured. Indeed, attack packets never reach the new virtual interface on the destination node before SAIDS reconfiguration is complete.

For the two cases that refer to the second LIDS (see Sections 4.6.3.4.2 and 4.6.3.4.3), the number of packets that arrive in the virtual interface of the LIDS container is identical to the number of packets reported by the Suricata process as captured, consequently no packets are dropped during the reconfiguration phase. We chose to compare the number of packets reported by the Suricata process with the number of packets received by the LIDS container as comparison of the number of packets reported in any previous stage (e.g. with the number of packets copied to the mirror interface) may have included non-SAIDS-related packet loss. After analyzing our obtained results we now discuss the limitations of SAIDS.

4.6.3.5 Limitations

SAIDS uses signature-based network IDSs and as such suffers from the inherent limitations of this type of intrusion detection. Therefore, SAIDS cannot detect unknown attacks for which a corresponding signature (i.e. rule) does not exist. Furthermore, since SAIDS works on a copy of the traffic, an additional mirror-induced delay is imposed between the time an attack reaches the target VM and the time when the alert is raised from the LIDS.

Regarding the connection between different SAIDS components, according to our scalability study, a secure connection per MAD is required. This could lead to network contention in a real production environment where thousands of MAD nodes are deployed.

In the scenario described in Section 4.6.3.2 we saw that SAIDS imposes negligible overhead for average-sized VMs (4GB and higher). Since the LIDS reconfiguration is completed before the VM migration is completed in the destination node, the safety mechanism does not have to halt the VM from resuming. However, SAIDS could impose some overhead in migration operations in cases of very light workload where the overall migration time is less than 4.14s (i.e. the time required for SAIDS to reconfigure an existing LIDS).

4.7 Summary

In this chapter we presented SAIDS, the first instantiation of our self-adaptable security monitoring framework. SAIDS is a self-adaptable network intrusion detection system that

satisfies four main objectives: 1. self-adaptation, 2. tenant-driven customization, 3. scalability and 4. security. SAIDS is able to adapt its components based on different types of dynamic events in the cloud infrastructure. Depending on the type of the event SAIDS can alter the configuration parameters of existing security probes or instantiate new ones. A detailed description of the adaptation process along with the role of each SAIDS component was presented.

We evaluated SAIDS under different scenarios in order to calculate the overhead of our approach in normal cloud operations, such as VM migration and we prove that SAIDS imposes negligible overhead in a VM migration. Furthermore, we evaluated the scalability and security/correctness of our approach with dedicated simulation scenarios. Scalability was evaluated in two different levels (from AM to multiple MADs and from a MAD to multiple LIDSs). Due to memory size restrictions imposed by our testbed the maximum number of LIDS that a single MAD can handle in parallel is 50 while the maximum number of MADs that a single AM can handle is 100. Overall SAIDS can handle up to 5000 LIDS in our current testbed, while this number could be increased making our solution suitable for a large scale cloud infrastructure. We have shown that SAIDS is able to detect attacks while handling dynamic events (e.g. VM migration) and is able to remain operational even during the adaptation process.

The contribution presented in this chapter was focused on intrusion detection. The next chapter presents the second instantiation of our security monitoring framework, AL-SAFE and is focused on intrusion prevention.

Chapter 5

AL-SAFE: A Secure Self-Adaptable Application-Level Firewall for IaaS Clouds

In this chapter we present the second instantiation of our framework which focuses on a different type of security component, the firewall. AL-SAFE is a secure application-level introspection-based firewall designed to cope with the dynamic nature of an IaaS cloud infrastructure. This contribution was published in [146]. In Section 5.1 we motivate the need for securing application-level firewalls and we present a justification of our design choices regarding AL-SAFE. The system and threat models that we adopted along with individual component description are presented in Section 5.2. Section 5.3 presents the adaptation process while implementation details are discussed in Section 5.4. Our evaluation strategy along with obtained results are presented in Section 5.5. Finally Section 5.7 concludes this chapter by listing key observations.

5.1 Requirements

Application-level firewalls are an important part of cloud-hosted information systems since they provide traffic filtering based on the type of applications deployed in a virtual infrastructure. However, they are subject to attacks originating both from inside and outside the cloud infrastructure. In this thesis, we aim at designing a secure application-level firewall for cloud-hosted information systems. In a cloud infrastructure, two security domains exists: One is concerned with traffic that flows between VMs inside the virtual infrastructure (that might belong to the same or different tenants) while the other is concerned with traffic that flows between the outside world and the virtual infrastructure. Consequently, an application-level firewall should address both domains.

Furthermore, a cloud-tailored application-level firewall should take into account tenant-specific traffic filtering requirements and self-adapt its ruleset based on dynamic events that occur in a cloud infrastructure. In this section we elaborate on the need for securing a cloud-tailored application-level firewall and we justify how AL-SAFE's design addresses this need. Furthermore, we detail the design principles of AL-SAFE and how they relate to the objectives of our self-adaptable security monitoring framework.

5.1.1 Why Should we Secure an Application-level Firewall

In contrast to typical host- or network-level firewalls which filter network traffic based on a list of rules that use IP addresses and ports, application-level firewalls operate based on a white list of processes that are allowed to access the network. This fine-grained filtering is achievable because application-level firewalls run inside the host operating system, and thus have a complete overview of the running applications and associated processes. Unfortunately, in the conventional design of application-level firewalls, isolation between the firewall and vulnerable applications is provided by the OS kernel, whose large attack surface makes attacks disabling the firewall probable. Hence, we address the following challenge: *Can we keep the same level of visibility while limiting the attack surface between infected applications and a trusted, application-level firewall?* In order to answer this question, we designed AL-SAFE. In the following section we present in detail how AL-SAFE's design addresses this impediment.

5.1.2 Security and Visibility

In order to address the issue of limiting the attack surface between the security device (i.e. the firewall) and a potentially compromised VM, we designed AL-SAFE to operate outside of the virtual machine it is monitoring, in a completely separate domain. Leveraging virtual machine introspection 2.5.2.2.1 we retain the same level of "inside-the-host" visibility while introducing a high-confidence barrier between the firewall and the attacker's malicious code. As we discussed in Section 2.5.2.2.3 firewalls in IaaS clouds are managed by the cloud provider. A successful firewall solution should be able to take into account the type of services deployed in the virtual infrastructure as well as the different dynamic events that occur in a cloud environment. Consequently, a cloud-tailored firewall should be able to allow customization of the filtering rules in a per-tenant basis (service-based customization), and also adaptation of the enforced ruleset upon the occurrence of dynamic events (self-adaptation). In the following section we detail AL-SAFE's design principles.

5.1.3 Self-Adaptable Application-Level Firewall

In AL-SAFE we enabled automatic reconfiguration of the enforced ruleset based on changes in the virtual infrastructure topology (virtual machine migration, creation, deletion) and in the list of services running inside the deployed VMs. To address the need of filtering intra- and inter-cloud attacks, AL-SAFE provides filtering at distinct infrastructure locations: at the edge of the cloud infrastructure (filtering network traffic between the outside world and the cloud infrastructure) and at the level of the local-switch inside each physical host (filtering inter-VM traffic). In this way AL-SAFE prevents attacks that originate both from outside and inside the cloud.

We now present a list of all the design principles of AL-SAFE :

- **Self-adaptation**: AL-SAFE's enforced ruleset should be configured with respect to dynamic changes that occur in a cloud environment, especially changes that refer to the virtual infrastructure topology. The source of these changes can be tenant decisions regarding the VM lifecycle (i.e. creation, deletion) or provider decisions regarding VM placement (i.e. migration).
- **Service-based customization**: the enforced ruleset should be configurable to only allow network traffic that flows towards and from tenant-approved services that are hosted in the deployed VMs. Addition or removal of legitimate tenant-approved services should lead to reconfiguration of AL-SAFE's ruleset.

- **Tamper resistance**: AL-SAFE should continue to operate reliably even if an attacker gains control of a monitored VM. In particular, the reconfiguration of the enforced ruleset should not explicitly rely on information originating from components installed inside the monitored guest.

- **Cost minimization**: the overall cost in terms of resource consumption must be kept at a minimal level both for the tenants and the provider. AL-SAFE should impose a minimal overhead on tenant applications deployed inside the AL-SAFE-protected VMs.

5.2 Models and Architecture

We adopt the same system and threat models as the ones described in Chapter 3 (Sections 3.2, 3.3).

We now present an overview of the events that trigger the adaptation process followed by AL-SAFE's design along with the presentation of key components.

5.2.1 Events that Trigger Adaptation

In order to satisfy the self-adaptation and service-based customization objectives, AL-SAFE is able to automatically configure the enforced rulesets on both filtering levels based on two categories of dynamic events: *topology-* and *service-* related changes. We list the events in each category along with their source in Table 5.1:

Table 5.1 – Events that trigger adaptation

Change category	**Event**	**Origin**	**Adaptation action**
Virtual infrastructure topology	VM creation VM destruction VM migration	Tenant Tenant, Provider Provider	{add rules} {delete rules} {add & delete rules}
Service list	Service addition	Tenant	{add rules}
Service list	Service removal	Tenant	{delete rules}

As listed in the table, virtual infrastructure topology-related changes include VM creation, migration and deletion while service list related changes include addition of new or removal of existing services on the deployed VMs. All dynamic events listed require either addition or removal of existing rules in AL-SAFE.

5.2.2 Component Description

AL-SAFE consists of five main components depicted in Figure 5.1: the edge firewall (EF), that filters network traffic between the outside world and the cloud infrastructure, a local switch-level firewall (SLF), that filters traffic in the local switch of each physical host, the Introspection component (VMI), the Information Extraction Agent (IEA), and the Rule Generators (RG), one for each firewall. All components are run by the cloud provider. AL-SAFE components are integrated in our self-adaptable security monitoring framework by interacting with the Adaptation Manager (located inside the cloud controller) and the Infrastructure Monitoring Probes (located in the cloud controller as well).

The IEA takes as a parameter a tenant-defined white list of processes that are allowed to access the network (*white-list* thereafter). Sharing the white-list with the provider essentially implies disclosing a list of processes that are approved for using the network.

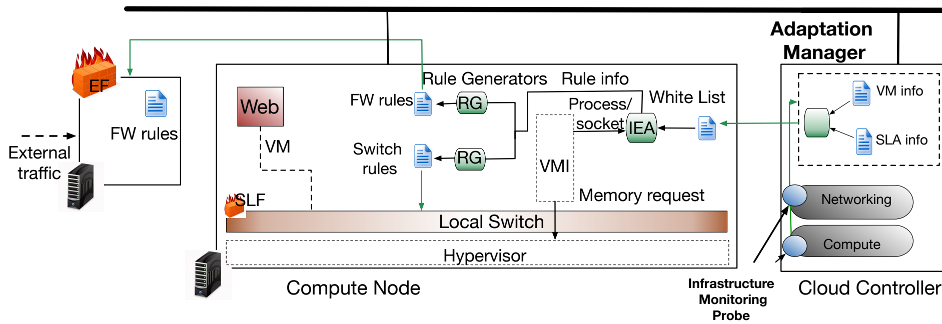


Figure 5.1 – The AL-SAFE architecture with the Adaptation Manager

AL-SAFE, as an application-level firewall requires this list in order to differentiate between connections that originate from tenant-approved services and potentially malicious connections. The white-list is updated each time a tenant adds a new approved process or removes an existing one. The white-list for the VM with ID *27* of the example in Section 3.5 (containing only the services that are expected to use the network) can be found in Listing: 5.1.

Listing 5.1 – White-list example with three tenant-approved services

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <firewallRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="language.xsd">
4   <application name="apache2">
5       <port num="80" proto="tcp">
6         <input action="ACCEPT" conntrack="NEW/ESTABLISHED" >
7         </input>
8         <output action="ACCEPT" conntrack="ESTABLISHED">
9         </output>
10      </port>
11  </application>
12  <application name="sshd">
13      <port num="22" proto="tcp">
14        <input action="ACCEPT" conntrack="NEW/ESTABLISHED">
15          <ip value="192.168.1.2" />
16          <ip value="192.168.1.3" />
17        </input>
18        <output action="ACCEPT" conntrack="ESTABLISHED">
19        </output>
20      </port>
21  </application>
22 </firewallRules>
```

In the white-list each tenant-approved network-oriented process is represented by an *application* entry in the XML file. The *application* entry has different fields: port and protocol that the process is expected to use and a list of IP address (public or private) that are allowed to connect to the process. In our example there are three processes that are allowed to use the network: an *apache server* and *ssh daemon*. Both the *apache server* and the *ssh daemon* have restrictions as to which IP addresses are allowed to interact with.

We now describe the individual AL-SAFE components along with their functionality.

5.2.2.1 VM Introspection

The VMI component is responsible for introspecting the memory of the monitored guest. VMI is able to coherently access the VM's physical memory and uses a profile of the VM's operating system's kernel to interpret its data structures. Thus VMI first extracts the list of running processes, and then iterates over this list to check if a network socket figures in the per-process list of file descriptors. For each network socket found, VMI extracts the process name, the pid as well as source and destination ports, IP address and communication protocol. The VMI-created list is named *connection list*.

5.2.2.2 Information Extraction Agent

The IEA compares the connection list thereafter resulting from the VMI with the tenant-defined white-list of processes. The Adaptation Manager is responsible for sharing the white-list with the Information Extraction Agent through a secure channel. The AM is also responsible for sharing updated versions of the white-list (e.g. when a new tenant-approved service is added). The IEA assigns an *allow* tag on connections from the connection list that figure in the white-list and a *block* tag on all other connections. The IEA propagates the connection information together with an ALLOW or BLOCK instruction to the next component, the Rule Generators. Furthermore the IEA component keeps a record of the rules used for each VM deployed on the compute node on which it runs.

5.2.2.3 Rule Generators

Due to the different types of filtering rules, AL-SAFE features one rule generator per type of firewall (one for the switch-level firewall and one for the edge firewall). Each RG creates the corresponding rules using all propagated information such as source port, source IP address, destination port, destination IP address and protocol. In the case of the switch-level firewall, the rules are grouped by VM with one rule table per VM. Each set of generated rules is then injected in its respective firewall.

5.2.2.4 Edge Firewall

The Edge firewall is located at the edge of the virtual infrastructure in a separate network device and is responsible for external traffic directed towards and from the virtual infrastructure.

5.2.2.5 Switch-Level Firewall

The Switch-level firewall is responsible for filtering network packets in the local switch using a list of ALLOW and BLOCK rules.

5.3 Adaptation Process

AL-SAFE automatically adapts the enforced ruleset based on changes in the topology of the virtual infrastructure and the list of services running in the deployed VMs. We present a high-level overview of the adaptation process in each one of these two cases. The adaptation steps (from introspection of the AL-SAFE-protected VM until the injection of the rules in the two firewalls) are demonstrated in Figure 5.2.

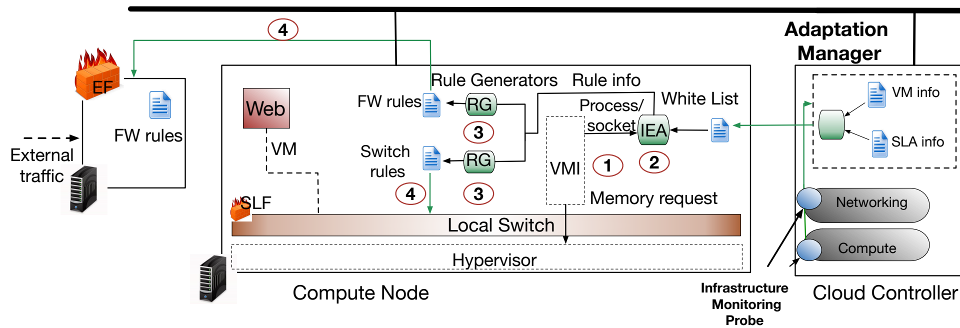


Figure 5.2 – Steps of the AL-SAFE adaptation

5.3.0.1 *Service-Related Changes*

First, the VMI periodically introspects the memory of the monitored guest to obtain the list of processes attempting to access the network. The time between two consecutive introspections is known as the *introspection period* and it defined in the SLA. Second, the IEA extracts the necessary information for generating filtering rules and propagates it to the two Rule Generators. Finally the RGs create the switch-level and edge firewall rules and inject them in the firewalls.

5.3.0.2 *Topology-Related Changes*

Depending on the type of topology-related change (VM creation, deletion or migration) different steps are followed:

***VM deletion*:** In this case no introspection of the deleted VM is required and no new rules are generated, thus the IEA is responsible for deleting the rules that filter the traffic towards and from the deleted VM.

***VM creation*:** In this case once the VM is set to an active state on the host node the process of service-related changes is followed. AL-SAFE currently supports two different security policies that can be applied to VM creation: *proactive* and *reactive* rule generation.

In the case of a *proactive* rule generation, a preliminary phase is executed before the VM enters an active state on the host node: rules that filter the traffic for the white-listed services are generated by the two RGs and inserted in the two firewalls. The proactive policy enables network connectivity for the white-listed services even before the VMI component introspects the memory of the deployed guest, thus preventing any performance degradation of the network-critical tenant applications. Unfortunately, it also generates filtering rules for services that might not yet be activated thus creating a potential entry point for the attacker (i.e. the attacker might identify the list of open ports and start sending malicious traffic towards the monitored guest through these ports).

In the *reactive* security policy, no preliminary phase is executed and all traffic directed towards and from the newly created VM is blocked until introspection finishes and the rules are generated and inserted in the two firewalls.

***VM migration*:** In the case of a VM migration only the rulesets of the switch-level firewalls at the source and destination nodes need to be adapted. Indeed, a VM migration should be transparent for the edge firewall. Since AL-SAFE follows a periodic

introspection strategy, the arrival of the migration request in an introspection period is critical. Let us define t_x as the introspection period and t_y as the time between the last start of an introspection and the moment when the migration command arrives at the source node of the deployed VM. Depending on the arrival time of the migration request we define two different cases:

1. The migration command arrives between two consecutive introspection actions. The remaining time until the next introspection $(t_x - t_y)$ is recorded and is sent as a parameter to the destination node along with the last valid introspection generated ruleset of the source node. The VM is resumed and the next introspection occurs after $t_x - t_y$. Since the VM migration command arrived between two introspections, the only way to respect the introspection period (that is not allow more time than t_x to pass between to consecutive introspections) is to introspect after $t_x - t_y$ time. Our strategy is depicted in Figure 5.3.

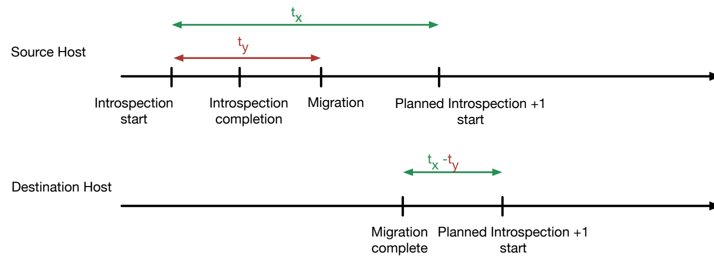


Figure 5.3 – The migration request arrives between two introspections

2. The migration command arrives during an on-going introspection. In this case the current introspection action is terminated and the result from the last valid introspection is sent to the destination node. A new introspection begins as soon as the VM is resumed in the destination node. Since the last introspection was killed it is important to obtain a valid introspection result as soon as possible (in order not to impose any performance penalty in new tenant-approved services that might have started right before the last killed introspection), consequently introspection starts immediately after the VM is resumed. Our strategy is depicted in Figure 5.4.

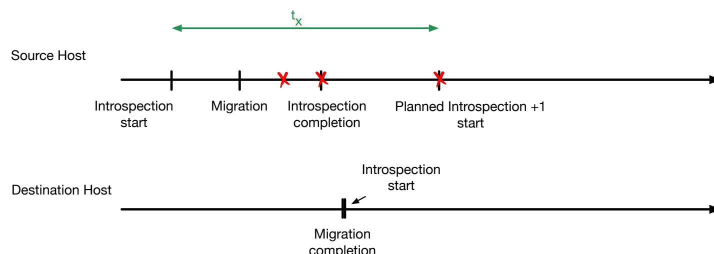


Figure 5.4 – The migration request arrives during an introspection

In a migration event the proactive policy is enforced where the last valid ruleset is injected in the switch-level firewall of the destination node before the VM is resumed.

5.3.1 Security Threats

We now present the security threats against specific AL-SAFE components and how they can be exploited from an attacker. We discuss our design choices for securing AL-SAFE from these attacks.

5.3.1.1 AL-SAFE Input Sources

AL-SAFE operates based on a tenant-defined white-list of processes that are authorized to use the network. An attacker could taint the contents of the white-list and allow illegitimate processes to use the network. The API-generated white-list is expressed in a simple XML format for which the parser is easy to make robust. The input file is pre-processed using a AL-SAFE-specific filter that verifies that only AL-SAFE-specific elements and printable ASCII strings without special characters are present in the file. Moreover, no complex interpretation is required since the values of each entry match fields of the firewall rules.

5.3.1.2 AL-SAFE Adaptation Arguments

AL-SAFE adapts the enforced rulesets in the two level-firewall based on topology or service related changes in the virtual infrastructure. Theoretically, an attacker could bypass the adaptation process or initiate an unnecessary one by tampering with the arguments of existing topology-related changes. AL-SAFE relies on the IMPs, which are located inside the cloud engine, in order to access all VM-related information (i.e. VM id, external/internal IP addresses, tap on the virtual switch, etc). The IMPs are hooks placed inside the cloud engine which copy information from the data structures used by the cloud engine in order to store network-related information regarding the VMs. Since the cloud engine and the information it stores, are considered to be tamper-proof the information extracted from the IMPs is considered accurate.

Regarding service-related changes, an attacker could tamper with the adaptation process in various ways. First, by tainting the arguments of a service (i.e. process name, port, protocol, etc) in order to force AL-SAFE to allow traffic towards and from attacker-preferred ports. AL-SAFE relies on VM introspection in order to detect service-related changes. Introspection parses kernel data structures in the VMs in order to extract the list of active network sockets together with their owner process name. Consequently, the only way for an attacker to tamper with the service arguments is by controlling the VM kernel this is an inherent limitation of all introspection-based solutions and we address it along with possible solutions in Sectionr̃eflim. Second, the attacker could force the introspection component to crash or exploit a software vulnerability in the component itself. The parsing phase relies on commodity tools that may be vulnerable to out-of-bound memory accesses and reference loops in the parsed structures. Out-of-bound accesses are avoided since the commodity tool that we use (i.e. Volatility, presented later in Section 5.4) is in Python which features automatic array boundaries check. To protect against reference loops, as a last option a timeout could be used to stop introspecting. The extracted information is only compared to the white-list of process names or inserted as port numbers (resp. IP addresses) in the filtering rules. It is thus sufficient to check that extracted values are 16 bits integers (resp. valid IP addresses).

5.3.1.3 Transfer of Reconfiguration Parameters

In AL-SAFE the tenant-defined white-list is sent from the AM located inside the cloud controller to the node hosting the monitored VM. An attacker could perform a "Man in the middle" attack during the sending phase and alter the content of the white-list. In our approach, we maintain a secure connection open at all times between the cloud controller and the compute nodes. The authentication protocol used [142] provides authentication of the AM and guarantees the integrity of the data transferred. Hence, an attacker has no way of intercepting or altering any part of the communication between the cloud controller and the compute nodes.

5.3.1.4 Firewall Rules

In AL-SAFE network packets are processed by the OpenFlow tables inserted in the local switch, and by the rules inserted in the edge firewall. Assuming that both filtering engines are robust, the added rules can be considered safe since the only actions allowed are to allow or drop traffic.

5.4 Implementation

We created a prototype of AL-SAFE from scratch using the KVM [27] hypervisor on a private cloud. Our cloud was deployed on OpenStack [32] and we used Open vSwitch (OvS) [137] as a multilayer virtual switch. In this section we present key implementation details of each component.

5.4.1 Edge Firewall

For the edge firewall we rely on the Nftables [147] stateful packet filtering framework which is deployed in a standalone Linux host.

5.4.2 Switch-Level Firewall

For the switch-level firewall our prototype features two versions. The first version uses the stateless filtering capabilities offered by Open vSwitch (i.e. essentially two rules per service are required, one for incoming and one for outgoing traffic). In the second version, AL-SAFE supports stateful filtering. The stateful filtering uses the OvS built-in feature of connection tracking *conn_state* in order to generate rules that keep track of open connections. Each open connection corresponds to an entry in the *conntrack* table. When a packet that is not part of any connection arrives, our prototype creates a new entry in the *conntrack* table and marks the connection as tracked. Mr Fergal Martin Tricot implemented the second version of the switch-level firewall during his 3 month Master 1internship that I co-supervised. The rules are grouped by VM (that is by switch port), with one OpenFlow table for each VM located on the compute node. The evaluation of AL-SAFE was conducted using the first version of the prototype.

5.4.3 VMI

In order to introspect the memory of a running VM we used LibVMI [113] combined with Volatility Memory Forensics Framework [148]. LibVMI [113] as the evolution of XenAccess, is a C library with Python bindings that facilitates the monitoring of low-level details (memory, registers, etc) of a running virtual machine. Since KVM does not

contain APIs that enable the access to the memory of a running VM a custom patch was applied that uses a dedicated Unix socket for memory access. The patch uses libvirt[138] in order to gain control over the running VM (i.e. pause, resume). Although LibVMI is not itself an introspection framework, it provides a useful API for reading from and writing to a VM's memory. LibVMI integration with Volatility [148] is done through a dedicated Python wrapper (PyVMI) that contains a semantic equivalence for each of the LibVMI's API functions. Figure 5.5 shows the full software stack from the patched KVM to Volatility.

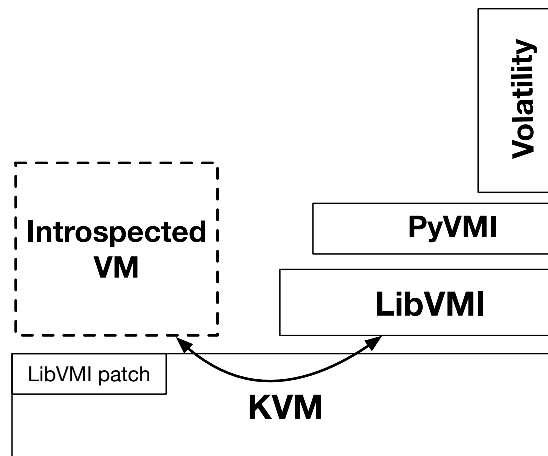


Figure 5.5 – LibVMI stack

Volatility can support any kernel version provided that a profile with the kernel symbols and data structures is created. The cloud provider would have to maintain a profile for each OS version deployed on the monitored VMs. As a modular framework, it provides different functionalities that are implemented by *plugins*. Each *plugin* performs a certain task such as identifying the list of running processes or the list of processes that have opened sockets (like the Linux *netstat* command). Volatility provides support to different processor architectures through the use of *address spaces*. An *address space* facilitates random memory access to a memory image by a plugin. A valid *address space* of a memory image is derived automatically by Volatility and is then used for satisfying memory read requests by each plugin. Unfortunately, Volatility was not designed in order to derive *address spaces* from memory images of running VMs which change constantly. In order to overcome this impediment we take a snapshot of the VMs memory (using LibVMI's built-in function *vmi_snapshot_create*) before each introspection. The overall flow of the VMI component actions is depicted in the chart shown in Figure 5.6.

The technique for obtaining the snapshot of the running VM's memory, called *stop-and-copy*, copies the whole memory of the VM to a temporary file. During this time the VM is paused and cannot make forward progress. Evidently, since snapshotting a VM implies copying a significant amount of memory, the time required is not negligible. Since our VMI component performs periodic introspections, it is necessary, for a successful introspection, that the introspection period (i.e. the time between two consecutive introspections) is larger than the time required to obtain a snapshot. The relation between the time required to obtain a snapshot and the introspection period is demonstrated in the Figure 5.7. Three scenarios are represented: introspection period larger than the time required to obtain the snapshot and introspect the snapshotted file, introspection period larger than the time required to obtain the snapshot but shorter than the time required to snapshot

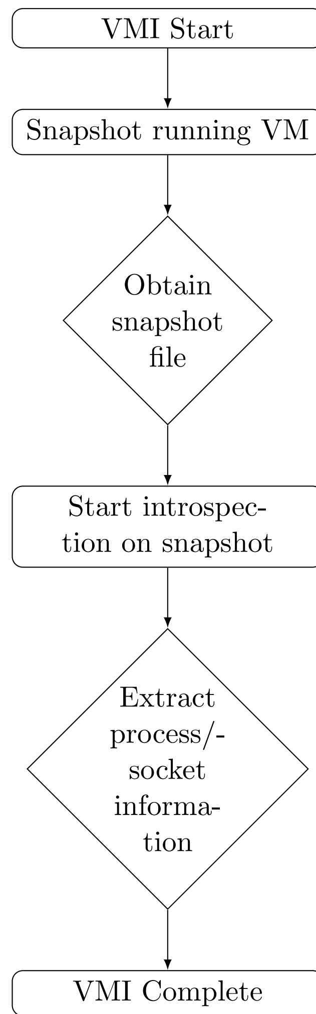


Figure 5.6 – Adaptation process flow chart

and introspect the snapshotted file and finally introspection period shorter than the time required to obtain the snapshot. We observe that defining an introspection period that is shorter than the actual time required to obtain a snapshot will result to a crash of the whole process.

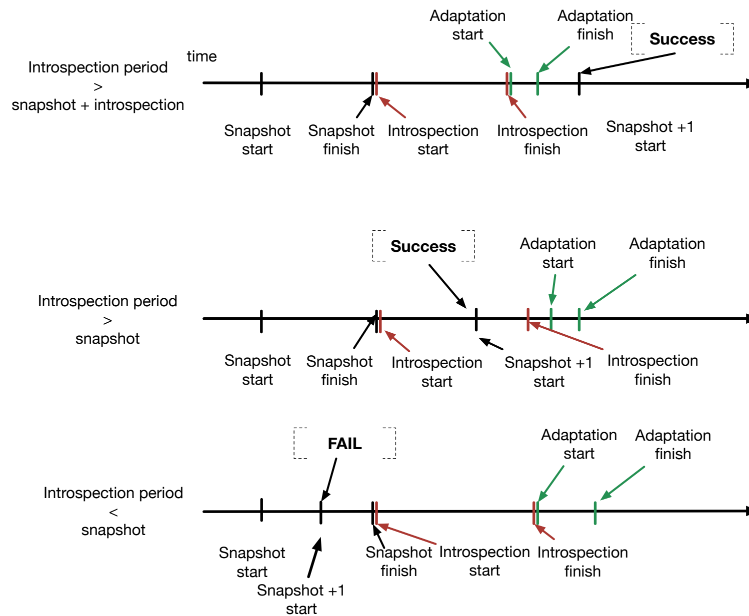


Figure 5.7 – Snapshot-Introspection relationship

For enabling periodic introspection we implemented a Python wrapper that creates a volatility object and performs plugin actions on that object at specific time intervals (i.e. introspection period). Our wrapper is also able to adapt the introspection period on the fly based on instructions received by the Adaptation Manager.

To enable VMI on dynamic infrastructure changes (e.g. VM migration), notifier hooks were placed inside the Nova function *plug_vifs()* that is executed on compute nodes and is responsible for creating the virtual interface(s) for the VM. The hooks pass all necessary information to VMI (VM name, id, version of running OS, etc) and start VMI immediately after the VM is resumed.

5.4.4 Information Extraction Agent

First, the IEA detects the differences between the last two consecutive introspections results and extracts the necessary information for rule generation (source and destination IPs, ports and protocol). Before propagating the information to the two parallel rule generators, a dedicated thread issues commands to the underlying OvS daemon (through the *list_interface* OvS command) and obtains the ID of the OvS port that corresponds to the introspected VM. Then it checks whether an OpenFlow table with filtering rules for that port exists and if not creates it. The IEA stores the table number along with the VM ID in a dedicated file (*table_info.txt*) for later use (e.g. in case the VM is deleted, the IEA extracts the table number from the file and issues a delete command for all the rules in that table to the underlying OvS daemon). The table number along with the port ID and the necessary rule information are passed to the rule generator of the switch-level firewall. An example of the information passed to the switch-level rule generator, for the ssh process belonging in the white-list of Listing 5.1 can be found in Listing 5.2.

Listing 5.2 – Information passed at the switch-level firewall

```
1 Rule_info (table = 28, ovs_port = 4, proto = TCP, port = 22, ips =
      [192.168.1.2, 192.168.1.3], action = ALLOW)
```

In this example, the IEA has cross-checked the introspection result with the white-list (found in Listing 5.1) and has found that the *ssh* process on port 22 is allowed to use the network. Then, it acquires the OvS port number (4) for that particular VM (through the *list_interface* OvS command) and the number for the OpenFlow table (28) where all the rules for that particular VM should be stored (stored in *table_info.txt*). Consequently he propagates the necessary information to the SLF rule generator.

5.4.5 Rule Generators

We implemented a separate rule generator for each firewall. The edge firewall rule generator produces Nftables-compatible rules while the switch-level firewall generator produces OvS compatible rules. To minimize the adaptation time, both rule generators are executed in parallel.

5.5 Evaluation Methodology

In this section we present our evaluation of AL-SAFE. We first present the objectives of our evaluation plan followed by our experimentation methodology. We performed the evaluation on the first version of our prototype, where the switch-level firewall is stateless (i.e. two rules per service are required one for incoming and one for outgoing traffic). The evaluation concludes with the correctness analysis and limitations of AL-SAFE in Section 5.6.3.

5.5.1 Objectives of the Evaluation

The main goal of AL-SAFE is to guarantee an equilibrium of a three-dimensional trade-off between **performance**, *security* and *cost*. In a cloud infrastructure different stakeholders are involved (i.e. tenants and the provider), consequently the trade-offs should be explored from each stakeholder's perspective. We first discuss our approach for evaluating AL-SAFE's performance, followed by the security and cost aspects.

5.5.1.1 Performance

The aspect of performance refers to the time required for AL-SAFE to complete a full adaptation loop (i.e. from the moment a dynamic event occurs until both firewalls are successfully reconfigured). In order to get an estimation of the overall time (i.e. latency) we need to answer the following questions:

1. *What is the overall time (in seconds) needed until both firewalls are successfully reconfigured?* The adaptation process consists of four phases: sharing of the white-list, snapshotting-introspection, rule generation and rule insertion. The overall latency is the sum of each phase's individual latency, which naturally leads us to a second question:

2. *What is the time (in seconds) required to complete each phase?* Depending on the tasks performed by each phase there are different components involved. Different

factors, related to each component's functionality, affect it's individual completion time. Consequently, a third question arises:

3. *What factors affect the execution time of each component?* We discuss the factors per component:

   - **Adaptation Manager**: It is responsible for sharing the tenant-generated white-list with nodes that host the monitored VMs. The number of entries in the list impacts the size of the file thus the time required for sending it to the corresponding nodes.
   - **Rule Generators**: They are responsible for generating the two separate rule categories and inserting them in the firewalls. The overall execution time for these components depends on the number of generated rules and the time required to insert them. Respectively, the number of generated rules is related to the number and type of services running inside a monitored VM and the tenant-defined white-list. Regarding the rule insertion time, for the switch level-firewall the number of rules affects the insertion time, while for the edge firewall the rules are written to a file, the file is then sent to the firewall host and finally the rules are inserted.
   - **Introspection**: The VMI component performs two functionalities: snapshotting and introspecting. Since the technique employed for snapshotting the monitored VM is stop-and-copy the only factor that affects the snapshotting time is the size of the monitored VM's memory. Introspection time depends on different factors as follows:
     - Number of running processes,
     - Number of created sockets,
     - Size of the introspected file (snapshot).

5.5.1.2 Security and Correctness

From a tenant's perspective, AL-SAFE is an application-level introspection-based firewall. AL-SAFE needs to allow only tenant-authorized services to use the network while blocking all other malicious network activity, even when the monitored VM is compromised. From the provider's perspective, AL-SAFE needs to guarantee that no security vulnerabilities are added in the provider's infrastructure by deploying AL-SAFE.

5.5.1.3 Cost

Cost minimization is one of AL-SAFE's core objectives. Thus, the associated overheads both from a tenant's and the provider's perspectives need to be examined. For the provider-associated cost we calculate the performance overhead imposed by AL-SAFE in normal cloud operations (e.g. VM migration) and the system resources consumed by AL-SAFE's components. Respectively, for tenant-associated cost we calculate the performance overhead imposed by AL-SAFE on tenant applications running inside monitored VMs.

- **Provider-associated cost**: *What is the latency (in seconds) introduced by AL-SAFE to a normal cloud operation such as VM migration?* and *What is the cost of deploying AL-SAFE's components in a compute node in terms of CPU and RAM?* All of the resources consumed by AL-SAFE are resources that cannot be assigned to

virtual machines (hence cannot generate profit for the provider). Consequently an exact computation for the CPU percentage and memory consumption is required.

- **Tenant-associated cost**: *What is the cost of deploying AL-SAFE as perceived by tenant applications?* In order to identify the quantitative cost induced by AL-SAFE we examine two different kinds of applications: process-intensive and network-intensive. We select these application profiles for simultaneously examining the main factors affecting each AL-SAFE component under different workloads. For the process-intensive application we identify the associated cost as the additional time required to perform a specific task. For the network-intensive application we identify the cost as the overhead induced in network throughput, application throughput and latency in connection establishment.

5.5.2 Experimentation Methodology

We now present the detailed scenarios that we used in order to perform the actual evaluation. It is worth mentioning that the scenarios were designed in order to address multiple evaluation objectives simultaneously. We select a Linux kernel build as a process intensive application and a web server and Iperf as network intensive applications. For a typical cloud operation we select a VM migration as a super case that includes VM creation (in the destination node) and VM deletion (in the source node). Each application is tested under different workload and introspection period parameters.

5.5.2.1 VM Migration

The VM migration scenario focuses on the provider-associated cost of deploying AL-SAFE. We aim at providing the reader with a fine-grained view of how intrusive a full adaptation loop is to VM migration. Although it can also provide an accurate estimation of the time (latency in seconds) required to perform each phase in the adaptation this is not the focus of this experiment. We compute the overall migration time of a monitored VM in seconds. The scenario has two options: no workload running in the migrated VM (idle) and a memory intensive workload running in the migrated VM. In the first case migration time is minimum (hence adaptation penalty is maximised) while in the second case the migration time is significantly larger (hence adaptation penalty is minimal). In this scenario the adaptation process only affects the switch-level firewall.

5.5.2.2 Linux kernel Build

In the Linux Kernel bulid scenario we compile a Linux kernel inside the untrusted VM and we vary the introspection period. The scenario serves a dual purpose as it addresses both the performance and cost objectives of our evaluation plan. Depending on the objective we compute different metrics:

1. **Performance of AL-SAFE**: We record the time required for each of AL-SAFE components to complete its functionality. The component that dominates the overall latency of a full self-adaptation loop in this particular scenario should be the introspection component. Since the scenario features a process-intensive application with no network activity, no rules are generated or inserted in the two firewalls. As discussed in Section 5.5.1.1, due to the snapshot technique selected, the memory size is the only parameter that influences the time required to obtain the snapshot.

Regarding the introspection time and with respect to the application profile we identify the number of processes and the size of the introspected file (i.e. snapshot) as influencing factors.

2. **Tenant-associated cost**: We measure the elapsed build time in seconds. Whether parallel compilation is enabled or not and the number of virtual CPUs in the virtual machine are expected to influence the result (due to the change in the number of processes). We also consider the time between two consecutive introspections to be an influencing parameter for the overall build time. Each introspection requires a snapshot of the monitored VM which freezes the VM during the snapshot time (due to the stop-and-copy technique), consequently the overhead increases with the introspection frequency.

5.5.2.3 Apache Web Server

In this scenario we install a web server on the monitored VM for serving new client requests. The scenario serves a dual purpose with regards to the evaluation objectives:

1. it quantitatively estimates AL-SAFE's performance
2. it calculates tenant and provider associated cost of deploying AL-SAFE

Depending on the evaluation objective we compute different metrics:

1. **Performance of AL-SAFE**: We record the time required for each AL-SAFE component to complete its functionality. We investigate the individual latency of each component: Introspection (VMI), Information Extraction Agent (IEA), rule generators (RGs) and rule insertion. As discussed in Section 5.5.1.1 the factors that affect the introspection time are: (a) size of the introspected file (i.e. snapshot), (b) number of processes and (c) number of sockets. Two of the factors (process and socket numbers) can be indirectly influenced through variation of the requests/second workload parameter (i.e. more requests/second imply more open sockets). The size of the introspected file can be influenced through assigning different memory values to the monitored VM. Regarding rule creation and insertion we recall that for the edge firewall a secure connection is required in order to inject the rules. Consequently, the influencing factors are the number of rules and the time required to establish a secure connection. In this case a variation in the number of requests can also indirectly influence the rule creation and insertion times. An example would be a scenario where the requests come from different client IP addresses and a list-based tenant security policy (detailed explanation below) is applied.

2. **Tenant-associated cost**: for calculating the tenant-associated cost of deploying AL-SAFE we compute the mean of the following values: latency induced in the response time for each new connection and service throughput. For the new connection response time different setups are examined depending on: the location of the client, the security policy enforced and the time of the request's arrival with respect to the introspection period. We detail each one:

   - *Location of the client*:

     (a) The client is located in a virtual machine belonging to the same tenant: only the switch-level firewall needs to be reconfigured.

   (b) The client is located outside the cloud infrastructure: both the edge firewall and the switch-level firewall need to be reconfigured.
   (c) The client is located inside the cloud infrastructure but in a virtual machine that belongs to a different tenant. In our setup the edge firewall is located on the gateway connecting the cloud infrastructure with the outside world. When a client request from a VM belonging to tenant T1 is issued to tenant's T2 web server VM (public_ip: 80) it first reaches the Neutron controller and then is redirected to the host executing the web server (essentially the request never leaves the cloud). Similarly to the first type of request, only the switch level firewall needs to be reconfigured.

- *Tenant-defined security policy*:
   (a) Policy *allow all*: Allow every request on port 80. This policy requires only one rule in each firewall thus the number of requests does not induce any additional latency.
   (b) Policy *allow only white-listed IPs*: In this case only requests from a tenant-defined address list are allowed.The latency depends on the number of IPs in the list. Since our switch-level firewall is stateless, we generate two rules per IP one for incoming and one for outgoing traffic while for the edge-firewall (stateful) only one is enough (since the conntracking feature allows to use a general rule for established connections).
   (c) Policy *block black-listed IPs*: Reasoning is similar to the *allow only white-listed IPs* policy. Every request is allowed besides the ones originating from blacklisted IPs. Again the latency depends on the length of the list. This policy can be combined with the *allow all* policy (i.e. allow connections from all IPs except the blacklisted ones).

- *Arrival of the request time in the introspection cycle*: Depending on when the connection request arrives and what is the timeout period for the TCP connection, we foresee the following outcomes:
   (a) The request arrives before the introspection has been completed. That is: arrival_of_request + timeout $<$ introspection_complete. In this case the connection fails.
   (b) The request arrives after the introspection has finished but before the adaptation of the two firewalls has been completed. That is: introspection_complete $<$ arrival_of_request + timeout $<$ adaptation_complete. Again the connection fails.
   (c) The request arrives before the adaptation of the two firewalls has been completed but the timeout is enough for the connection to wait until the completion of the adaptation. That is: introspection_complete $<$ arrival_of_request + timeout $\geq$ adaptation_complete and arrival_of_request + timeout $>$ adaptation_complete. In this case the connection succeeds(the port will be open).
   (d) The request arrives after the adaptation of the two firewalls has been completed. In this case the connection succeeds.

After defining the metrics used in this scenario we now focus on the varying parameter in the different workloads that we use, that is the number of requests per second. The web server spawns new sockets in order to serve the requests. In this case we expect an increment in the introspection time. Regarding memory size, we test with

2048MB memory for the VM and two virtual CPUs. The memory size represents average workload use-cases (medium traffic websites, small databases, etc) as stated in [15]. The memory size is expected to be the only factor affecting the snapshotting time.

3. **Provider-associated cost**: we calculate AL-SAFE resource utilization in terms of CPU percentage and memory consumption.

This scenario refers to measuring the performance impact of a full adaptation loop for a server that is accepting incoming connections. For evaluating the impact of a full adaptation on a client located inside the virtual infrastructure (in an AL-SAFE-protected VM) attempting to connect to a web server located outside the virtual infrastructure (hence adaptation of both firewalls is required in order to allow the client traffic to pass unimpeded), we calculate the latency induced in the response time for each new connection. In this case we execute a full adaptation loop only on the client's side. Unfortunately, in contrast to the server side where the connection port is known a priori (port 80 or port 443 for https requests), the client case comes with one impediment: the port number is unknown until the client attempts to make a new connection (hence a rule that allows the connection cannot be inserted proactively in the two firewalls rulesets). In order to overcome this impediment we include two security policies:

1. **Proactive security policy**: allow all traffic directed towards the server's IP and port 80. With this option all traffic towards the web server is allowed regardless of the source port.

2. **Reactive security policy**: wait until introspection detects the source port of the white-listed process and then insert the rule that allows traffic for that particular port only.

The proactive option clearly favors performance as it offers minimal service disturbance but it also introduces security vulnerabilities (since no control is performed on the source process of the connection) as a potentially malicious process executing on the monitored client can gain access to the legitimate web server.

5.5.2.4 Iperf

This scenario is used to evaluate the effect of the introspection phase on the network throughput. We install Iperf [149] in the VM which acts as a server and we use a separate node outside the cloud infrastructure as a client. Iperf measures the maximum available bandwidth on an IP network. The selected scenario focuses on the cost objective of our evaluation plan. The computed metrics are:

1. **Tenant-associated cost**: we measure the network throughput in sending/receiving a *TCP stream*. As in the kernel compilation scenario, each time the VM is introspected a snapshot is taken, which freezes the VM during the snapshot time (due to the stop-and-copy technique), consequently the overhead increases with introspection frequency.

2. **Provider-associated cost**: we calculate AL-SAFE resource utilization in terms of CPU percentage and memory consumption.

We compare the Iperf results with and without introspection.

5.5.2.5 Microbenchmarks

In the previous scenarios we described our methodology for measuring the overhead of a full adaptation loop on network- and process-intensive tenant applications. This section focuses on a fine-grained view of the cost for a full adaptation loop particularly on individual connection establishment.

5.5.2.5.1 TCP connection establishment time: We wrote a client/server TCP program and measure the TCP connection setup time for a single connection to a node outside the virtual infrastructure. We address both cases where either the server or the client are executed inside the monitored VM. We compare the results obtained without adaptation to the ones with the adaptation of the two firewalls.

- *Server inside the AL-SAFE-protected VM:* The setup of this case is depicted in Figure 5.8.

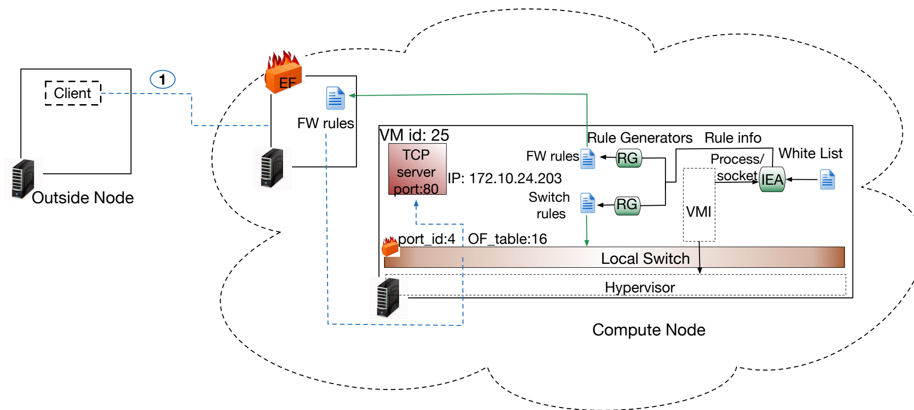


Figure 5.8 – TCP server setup

In this case a client located outside the cloud infrastructure is trying to connect to a tcp server on port 80 running inside an AL-SAFE-protected VM. The VM with ID:25 has *port_id:4* and *OF_table:16* on the virtual switch of the compute node. The white-list of this scenario is shown in Listing 5.3. The white-list states that the process named *tcp_server* is allowed to accept incoming connections.

Listing 5.3 – VM 25 white-list

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <firewallRules xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="language.xsd">
4    <application name="tcp_server">
5        <port num="80" proto="tcp">
6          <input action="ACCEPT" conntrack="NEW/ESTABLISHED" >
7          </input>
8        </port>
9    </application>
10   <application name="tcp_client">
11       <port num="0" proto="tcp">
12         <output action="ACCEPT" conntrack="NEW/ESTABLISHED">
13         </output>
14       </port>
15   </application>
16   <application name="udp_r">
17     <port num="68" proto="udp">
18     <input action="ACCEPT"/>
19     </port>
20   </application>
21 </firewallRules>
```

Consequently the rules that are inserted in the two firewalls after the adaptation loop is complete are:

Switch-level firewall: *table=16, priority =10, tcp, tp_dst=80, in_port=4, actions=ALLOW* for incoming traffic , *table=16, priority =10, tcp, tp_src=80, out_port=4, actions=ALLOW* for outgoing traffic (since the evaluation is conducted on the first version of our prototype two rules are required for the SLF).

Edge firewall : The rule added in the *input* chain is: *tcp dport 80 counter accept.* The firewall already contains a rule for established connections thus the reply from the server will be allowed.

- *Client inside the AL-SAFE-protected VM:* The setup of this case is depicted in Figure 5.9.

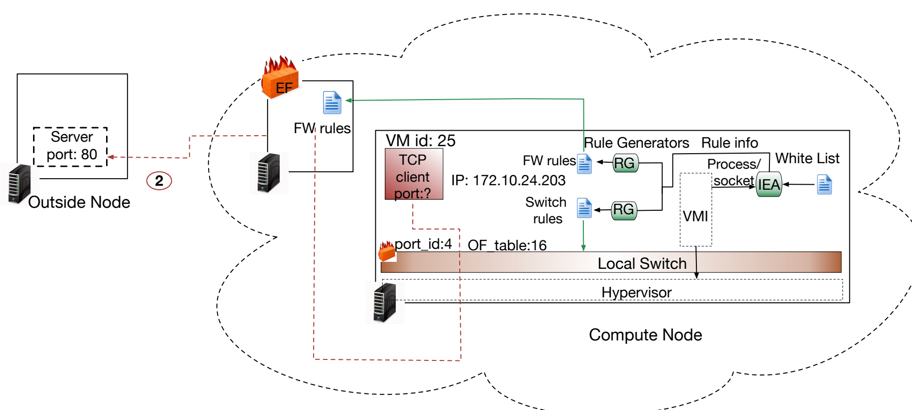


Figure 5.9 – TCP client setup

In this case a client inside the AL-SAFE-protected VM (same VM as in the previous example) is trying to connect to a tcp server located outside the cloud infrastructure. Since the process *tcp_client* is allowed to initiate connections (white-list in Listing 5.3) the rules for the two firewalls after the adaptation loop (and after the port used by the *tcp_client* has been discovered by the introspection) are:

Switch-level firewall: *table=16, priority =10, tcp, tp_src=1451, out_port=4, actions=ALLOW* for outgoing traffic , *table=16, priority =10, tcp, tp_dst=1451, in_port=4, actions=ALLOW* for the incoming reply.

Edge firewall : The rule added in the *output* chain is: *tcp sport 1451 counter accept.* The firewall already contains a rule for established connections thus the reply from the server will be allowed.

5.5.2.5.2 UDP round trip time: for evaluating a UDP stream setup cost we wrote a small client/server program that transmits a block of data and receives an echo reply. We measure the round trip time with and without the adaptation of the two firewalls. The setup of this case is depicted in Figure 5.10. In this case the receiver of the message is

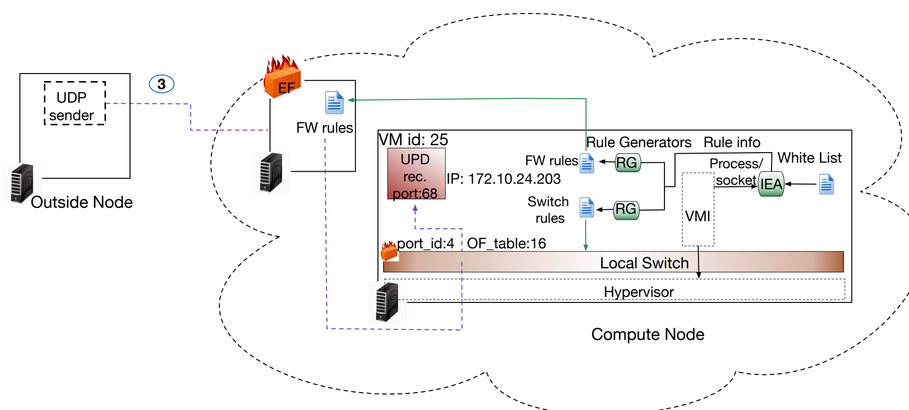


Figure 5.10 – UDP setup

located in the AL-SAFE-protected VM (same VM as in previous examples). The process *udp_r* is allowed to use the network (white-list in Listing 5.3). Consequently the rules added in the two firewalls after the adaptation loop are:

Switch-level firewall: *table=16, priority =10, tcp, tp_dst=68, in_port=4, actions=ALLOW* for the incoming block , *table=16, priority =10, tcp, tp_src=68, out_port=4, actions=ALLOW* for the reply.

Edge firewall: The rule added in the *input* chain is: *udp dport 68 counter accept.* The firewall already contains a rule for established connections thus the reply from the server will be allowed. In both micro-benchmarks the memory of the VM, the number of processes and the number of sockets are constant. The only influencing parameter is the time of the request's arrival in the introspection period (as discussed in the web-server scenario described in Section 5.5.2.3).

5.6 Evaluation Results

After describing our evaluation scenarios and the underlying rationale, we present the results obtained from the different experiments. Section 5.6.1 presents the results from the performance evaluation of AL-SAFE while Section 5.6.2 discusses correctness aspects of our approach. Finally AL-SAFE limitations are detailed in Section 5.6.3.

5.6.1 Performance and Cost Analysis

To do our experiments we deployed a datacenter with three physical hosts: one cloud controller and two compute nodes. Each physical host has 48GB RAM and runs a 64bit Linux Ubuntu 14.04 distribution. The machines are interconnected with a 1Gb/s network. All the VMs deployed on the compute nodes run a 64bit Linux Ubuntu 13.10 distribution with 2 cores and 2GB RAM. We also deployed the Nftables firewall in a fourth physical host with the same hardware as our cloud nodes. All reported results are compared to a baseline value obtained without AL-SAFE.

Before running our experiments we conducted a preliminary set of tests to calculate the time for a full snapshot of a 2GB VM's memory. We calculated the mean snapshot time to 1.5 seconds over 10 repetitions (standard deviation 0.05). Since the technique used copies the whole memory of the VM into a dedicated file the size of the VM is the only factor affecting the snapshot time.

5.6.1.1 VM Migration

To generate the memory-intensive workload we used *bw_mem_wr* from the LMBench benchmark [143] suite with a 1024MB working set. The working set is allocated, zeroed and then written as a series of 4 byte integers. In this scenario we aim at proving that the time required to reconfigure the switch-level firewall is independent from the VM workload. We executed 10 repetitions of each case. The results are presented in Figure 5.11. In the

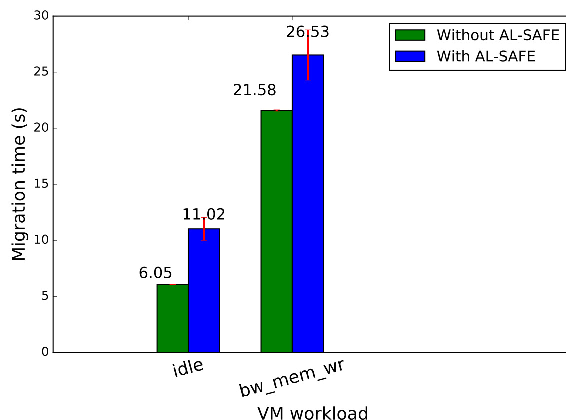


Figure 5.11 – Migration time with and without adaptation

figure, the migration time with and without AL-SAFE for both cases (idle and memory intensive workload) is depicted. The imposed overhead in the migration operation is the same in both cases (4.95s), which validates our hypothesis that the cost of adapting the firewall ruleset is independent from the VM workload. A per-phase breakdown is shown in Figure 5.12. We insert two rules per service in the switch-level firewall (one for ingress and one for egress traffic) and only one rule in the edge firewall. The relatively high amount of

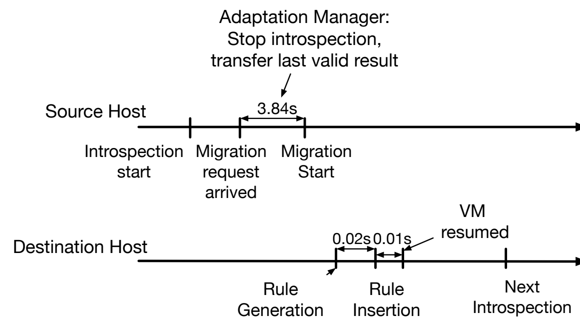


Figure 5.12 – Breakdown of each phase in seconds

time (3.84 seconds) required for the Adaptation Manager to notify the Introspection agent of the source node, in order to interrupt the introspection and send the last valid results, is mostly due to a defect in the DNS configuration resulting in a slow SSH connection establishment.

5.6.1.2 Linux Kernel

We compiled a Linux kernel inside the VM and we varied the introspection period. The kernel was compiled with a configuration including only the modules loaded by the running kernel of the VM, using gcc 4.8.4 with a degree of parallelism of 3. We used the *time* command line utility for measuring the overall execution time. The VM is not expected to start services that use the network during the execution time of the experiment thus no adaptation of the firewalls is required.

Before presenting the results, we discuss a model that estimates the minimum overhead value on the kernel compilation time. Let us define: x the time overhead introduced in the kernel compilation time, α as the mean value of the time required to take a snapshot and n the number of introspections performed during the experiment. Since in each introspection a snapshot of the AL-SAFE-protected VM is taken, a temporary freeze of the VM is performed. Consequently, the minimum overhead should be the result of the number of introspections times the snapshot time. That is $min(x) = n \times \alpha$.

The mean value over five repetitions is shown in Figure 5.13. The results clearly

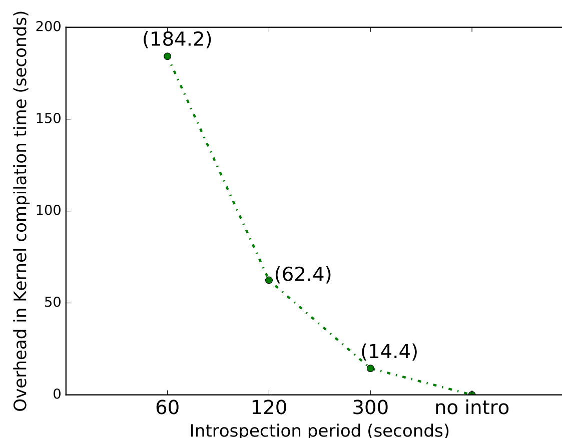


Figure 5.13 – Impact of the introspection period on kernel compilation time

demonstrate a dependency between the period of introspections and execution time. The

highest overhead (12%) is observed when the introspection period is 60 seconds. Indeed the observed overhead (184.2s) conforms with our overhead model as our computed value is 28 $\times$ 1.5 = 42 and 184.2 >> 42. Each introspection requires a snapshot of the running VM which freezes the VM for a short period of time. Obviously, more introspections requires more freezing time for the VM, which translates to a higher execution time. The lowest overhead (14.4s) is observed when the introspection is performed every 5 minutes. Again the result conforms with our overhead model (minimum overhead is computed at 9s). The results suggest that there are additional factors, besides the freezing time resulting from the snapshot, that contribute to the overall overhead value.

5.6.1.3 Apache Web Server

For a network intensive application, we installed the Apache Web server [150] and we used ApacheBench [151] to generate different workloads. In this scenario we examine two aspects of our design: first the dependency between the introspection period and the Web server throughput and second the dependency between the arrival of the connection request in the introspection period and the Web server latency. The second aspect shows the impact of using periodic introspection on the availability of a new Web server instance, like in a cloud scale-up operation. For both aspects the client is located outside the virtual infrastructure. We choose to test with an outside client as in the second aspect, reconfiguration of both firewalls is required and a comprehensive insight into AL-SAFE's overhead is provided.

For the first aspect no adaptation of the firewalls is required (a preliminary phase to allow the connection between the server and the client is executed), while the only varying parameter is the introspection period. We run the experiment for 3 minutes and record the results over five repetitions. The workload consists of 750,000 requests from 1000 concurrent clients. The results shown in Figure 5.14 validate our previous observation

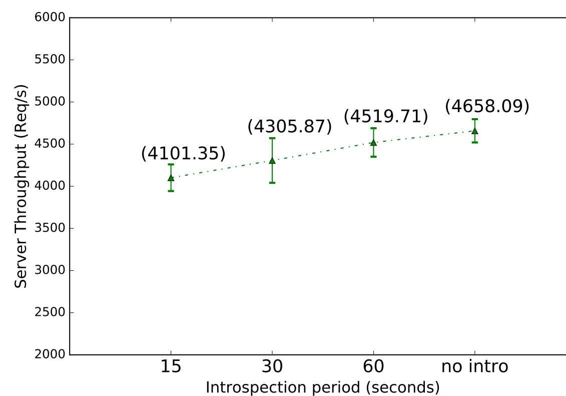


Figure 5.14 – Impact of the introspection period on server throughput

regarding introspection period and performance degradation. In this scenario, the highest number of introspections (20 for the 15 seconds period) imposes the highest cost in the server's throughput (12%).

For the second aspect we fix the introspection period at 30 seconds and we start the Web server at port 80 between two introspections. Thus an adaptation of both firewalls is required in order to allow the connections from the client to pass unimpeded. In this experiment we vary the arrival time of the connection request (right before snapshot, in the middle of introspection, at the end of introspection and after introspection). The

workload consists of 50,000 requests from 1000 concurrent clients. The results over five repetitions are shown on Figure 5.15.

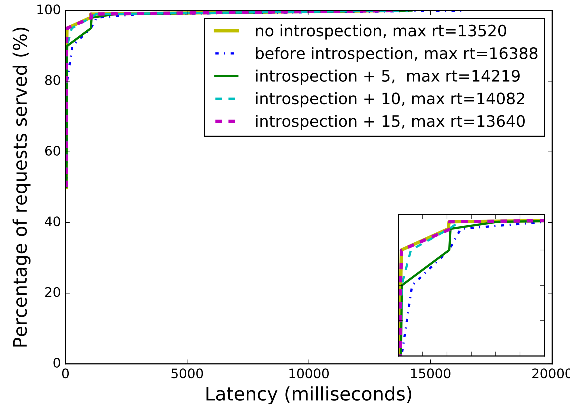


Figure 5.15 – Request service time for different times in the introspection cycle

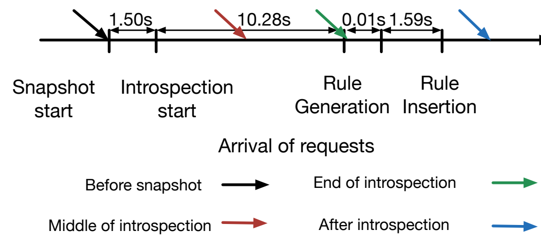


Figure 5.16 – Cases of request arrival time with respect to the introspection cycle

The largest impact on the web server latency (blue dotted curve) is when the client requests are issued right before the introspection takes place. Indeed, in order to establish the connection, the client application has to wait for the introspection to be completed, the rules to be generated by the two separate rule generators and then injected in the two firewalls (two rules per service for the switch-level firewall and one for the edge firewall). This translates to a minimum connection time of 13.38s (1.5s for the snapshot of the AL-SAFE-protected VM + 10.28s for the introspection + 1.60s for rule generation and insertion).

A per-phase breakdown of the produced overhead is shown in Figure 5.16.

When the requests are issued at the end of introspection in the cyan dotted curve, we observe that the curve is much closer to the corner of the graph. This observation holds for all curves (cyan dotted and purple dotted) that represent low latency cases (requests are issued either at the end or after introspection). The produced overhead (in the minimum connection time) results from the time required to reconfigure the two firewalls. The time required to reconfigure the edge firewall is significantly larger than the one for the switch-level firewall due to the establishment of a secure connection between the node that hosts the VM and the firewall node.

5.6.1.4 Iperf

For the Iperf experiment we use the standard TCP_STREAM test with a 64KB socket stream and 8KB message size. We run the experiment for 300 seconds and record the result.

Before the experiment is executed we run a preliminary phase that configures both firewalls to allow the connection, such that no adaptation is taking place during the experiment. The mean results over five repetitions are shown in Figure 5.17. The graph shows the

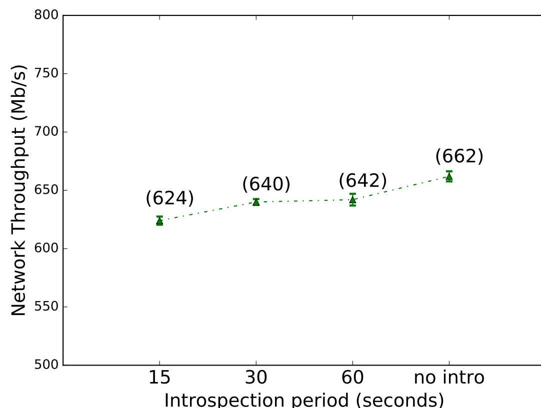


Figure 5.17 – Impact of the introspection period on network throughput

network throughput value for increasing introspection periods. The results confirm our previous observation regarding introspection period and performance overhead. Indeed a shorter introspection period results in more snapshots that obviously result to more downtime for the VM. In this case the highest overhead (5.75%) is observed in the 15 seconds case (20 snapshots).

5.6.1.5 Micro-Benchmarks

Before presenting the individual results of each micro-benchmark we present a model for estimating the overhead of the adaptation loop on individual connection establishment. Let us define: x the overhead in terms of seconds for a full adaptation loop, α the time required to obtain a snapshot of the monitored VM, β the time required to perform the actual introspection process, γ as the time required for the firewall reconfiguration and π as the introspection period (i.e. the time between two consecutive introspections). Depending on when in the adaptation loop the connection request is issued and whether it is a client or a server which is hosted in the AL-SAFE-protected VM, we define different values for x.

5.6.1.5.1 Adaptation on the Server Side – TCP: In this case we install a server in the AL-SAFE-protected VM and we issue a connection request from a client located outside the cloud infrastructure. Consequently both firewalls need to be reconfigured in order for the connection to be established.

- Request issued right before introspection: $x = \alpha + \beta + \gamma$. That is the request has to wait for each phase to be completed before it reaches its destination.

- Request issued in the middle of introspection: $x = \frac{\beta}{2} + \gamma$. The request has to wait until the introspection finishes and the two firewalls are successfully reconfigured before it reaches its destination.

- Request issued at the end of introspection: $x = \gamma$. The request has to wait only for the two firewalls to be reconfigured in order to reach its destination.

5.6.1.5.2 Adaptation on the Client Side – TCP: In this case we install a client inside the AL-SAFE-protected VM and we issue connection request to a server located outside the cloud infrastructure. Consequently both firewalls need to be reconfigured in order to establish the connection.

- Request issued right before introspection: $x = \alpha + \beta + \gamma$. That is the request has to wait for each phase to be completed before it can leave the cloud infrastructure.
- Request issued in the middle of introspection: $x = (\pi - \frac{\beta}{2}) + \alpha + \beta + \gamma$. Since the introspection is performed on a snapshot of the AL-SAFE-protected VM, which was taken before the client was started, the client process does not appear in the introspection result (i.e. because the client process was not started at the moment the snapshot was taken). Consequently, the connection request needs to wait until the next snapshot, introspection and adaptation in order to leave the cloud infrastructure.
- Request issued at the end of introspection: $x = (\pi - \beta) + \alpha + \beta + \gamma \rightarrow x = \pi + \alpha + \beta$. The request needs to wait until the next introspection and subsequent firewall reconfiguration.

5.6.1.5.3 Adaptation on the Receiver Side – UDP: This case follows the same overhead estimation as the adaptation on the server side for a TCP connection.

5.6.1.5.4 Adaptation Latencies: The different arrival times with respect to the adaptation phase are presented in Figure 5.18. The figure also shows the mean values for the different phases of the adaptation, snapshot, introspection, rule creation, rule insertion. These are the values used for computing the estimated overhead in each case according to our overhead model.

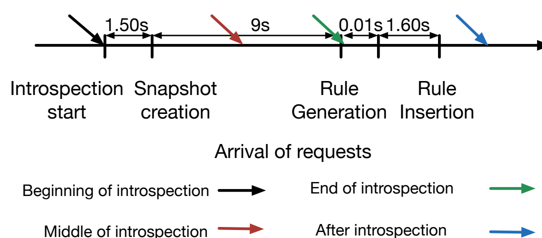


Figure 5.18 – Cases of request arrival time with respect to the introspection cycle

Since the introspection time only depends on the number of running processes (and open sockets) and in the micro-benchmark experiments we create only one new process in order to handle the connection, we can assume that the same value for the mean introspection time can be applied to both TCP and UDP scenarios. Furthermore, in both cases 2 firewall rules are inserted in the switch-level firewall (because we perform the evaluation on the first, stateless, version of our prototype) and only one rule in the edge firewall. Consequently, the same mean value for rule creation and insertion can also be applied to both scenarios.

5.6.1.5.5 Inbound TCP Connection: This experiment computes the overhead of the adaptation when the server is located inside the cloud infrastructure and the connection request comes from a machine outside the cloud infrastructure (i.e. both firewalls need

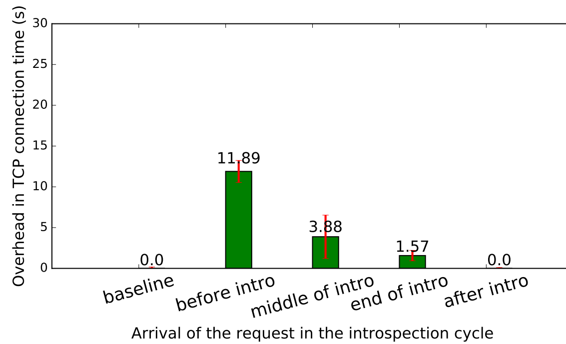


Figure 5.19 – Inbound TCP connection establishment time

to be reconfigured). Figure 5.19 shows the connection establishment times when the connection requests are issued at different times during the introspection process(beginning, middle, end, after). The case with the smallest overhead (1.57s) is when the request is issued at the end of introspection. Indeed the request only has to wait for the firewall reconfiguration in order to reach the server. According to our model, the observed overhead should be 1.61s which is indeed the case. We observe again a relatively high time required for the secure connection establishment (1.60s), which is due to the already discussed DNS defect when establishing a secure SSH connection.

The case that demonstrates the highest overhead is the one when the request is issued right before the snapshot. That is because, the request has to wait until the snapshot is taken, the introspection is complete and the rules are generated and injected in the two firewalls. According to our model the estimated overhead in this case is : $\alpha + \beta + \gamma$ =1.5s + 9.0s + 1.61s = 12.11s. The observed overhead is 11.89s. The 0.22s (1.68%) deviation between the estimated overhead value and the observed overhead is attributed to the fact that the estimated overhead was computed based on mean values for each phase.

The results demonstrate that the arrival of requests in the introspection cycle plays a major role in the connection establishment time. For a client attempting to connect to an AL-SAFE-protected server the best case scenario is issuing a request at the end of the introspection cycle.

5.6.1.5.6 Outbound TCP Connection: In this experiment the TCP client is installed in the AL-SAFE-protected VM inside the cloud infrastructure issuing connection requests to a server located outside the cloud infrastructure. Consequently, both firewalls need to be reconfigured in order for the client request to pass unimpeded. In contrast with an inbound TCP connection where the connection port is known a priori (e.g. port 80), an outbound TCP connection faces the limitation of an unknown port number. The overhead in connection establishment times, when issuing the request at different times during the introspection process, is shown in Figure 5.20. Initiating a request right before introspection is now the best case scenario with the smallest overhead. Indeed the open socket will be included in the new introspection result (since the client process is not started in the AL-SAFE-protected VM when the snapshot of the first introspection was taken detailed presentation in Section 5.6.1.5). According to our model the estimated overhead is: $\alpha + \beta + \gamma$ = 1.5s + 9s + 1.6s = 12.11s. The observed overhead is : 12.03s which validates our initial hypothesis about the cost of issuing the request right before the introspection of the AL-SAFE-protected VM. In all other cases the time period between the time of the request and the next introspection has to be added to the connection establishment

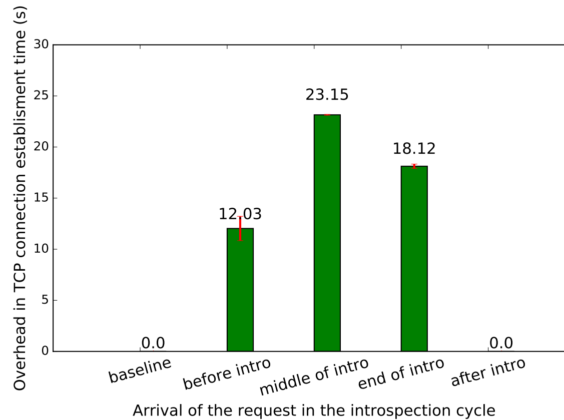


Figure 5.20 – Outbound TCP connection establishment time

time. For example, in the case where the request is issued at the middle of introspection the added time is 10.5s (the introspection period was defined at $\pi = 15$s and the request was issued at the middle of the introspection process at $\frac{\beta}{2} = 4.5$s consequently the waiting time amounts at 10.5s). In this case the estimated overhead should be: $\pi - \frac{\beta}{2} + \alpha + \beta + \gamma = 10.5\text{s} + 1.5\text{s} + 9\text{s} + 1.61\text{s} = 22.61\text{s}$. The actual overhead is 23.15s (we again observe a deviation of 0.54s or 2.37% between estimated and observed overhead due to the mean values used for calculating the estimated overhead).

The case of a client located inside the AL-SAFE-protected VM is the exact opposite of the server case (presented in Section 5.6.1.5.5). The best case scenario now is when the connection request is issued at the beginning of the introspection.

5.6.1.5.7 UDP Round Trip Time: In the UDP round trip time experiment we install the process receiving the ECHO request inside a monitored VM located inside the cloud infrastructure. Consequently, both firewalls need to be reconfigured in order for the message to complete its roundtrip. Figure 5.21 shows the overhead in connection establishment times when the message is send at different times in the introspection period. The observed overhead follows a similar pattern with the one imposed on the inbound

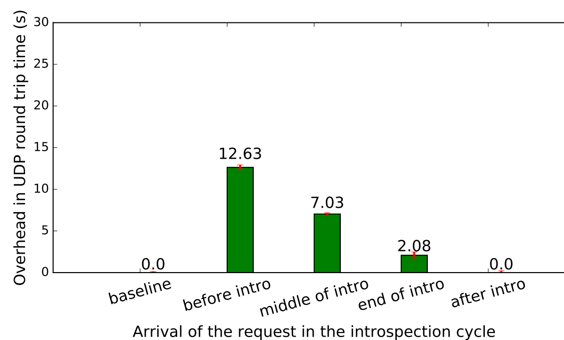


Figure 5.21 – Inbound UDP round trip time

TCP connections (see Section 5.6.1.5.5). The best case scenario is when the message is sent at the end of the introspection process (i.e. it has to wait only for rule creation an reconfiguration). Indeed the observed overhead conforms with our estimation: 2.08s and 1.61s respectively the deviation is once more attributed to the mean values used for

calculating the estimated overhead. The worst case scenario is when the message is sent right before introspection. Indeed the message has to wait until the introspection process finishes and the two firewalls are successfully reconfigured. According to our model the overhead is estimated as: $\alpha + \beta + \gamma = 1.5s + 9.0s + 1.61s = 12.11s$. Again the observed overhead conforms with our model. In UDP communications, much like TCP connections, when the request arrives in the introspection cycle (beginning, middle, end) highly affects the produced overhead. The case where the request is issued at the end of the introspection is the one with the smallest overhead.

5.6.1.6 Resource Consumption

In this section we discuss the cost of AL-SAFE in terms of CPU consumption and RAM. We focus our analysis on the Introspection component (VMI) as it is the one expected to consume the most resources. Since the introspection mechanism extracts the necessary information about network sockets by iterating on the process list of the running VM it is obvious that the number of processes affects both the execution time of the VMI and the required resources.

We calculate the CPU and RAM utilization of the introspection process in our Web server scenario (Section 5.6.1.3), with a generated workload of 750,000 requests from 1000 concurrent clients, over ten executions. Since our Web server is configured with an event-based module, it is expected to generate many child processes, each one handling a pre-specified number of threads. We compare the result with the resources consumed by the VMI in the Iperf scenario (Section 5.6.1.4) where only a single process is created to handle the connection socket. The results are shown in Table 5.2. The table lists average CPU usage and memory consumption along with the overall execution time of the VMI component (*real*) and the times spent in user (*usr*) and kernel (*sys*) modes.

The high cost of introspection in terms of memory is because Volatility loads the whole snapshot file (in both cases 2 GB) into memory. The number of generated processes inside the monitored VM increases the CPU consumption of the VMI component.

Table 5.2 – Resource consumption of the introspection component

Application	*Real (s)*	*Usr (s)*	*Sys (s)*	*CPU%*	*Memory (MB)*
Apache	13.6	5.04	2.21	53.6	2193
Iperf	11.9	3.75	1.60	45	2193

5.6.2 Correctness Analysis

In this section we justify the security and correctness aspects of AL-SAFE. We focus on the functionality of AL-SAFE as a firewall as well as the contribution of the AL-SAFE approach in addressing inherent design limitations of application-level firewalls.

Since AL-SAFE is an application-level firewall one of its main security goals is to successfully block unauthorized connections. We have validated the correctness of our generated rules both for inbound and outbound connections. For intra-cloud connection attempts the switch-level component of AL-SAFE successfully intercepted all packets from processes that were not in the white-list. For extra-cloud inbound connections the packets were stopped by the edge firewall. In both cases no unauthorized packets reached the VM or left the compute node.

In a typical system, software exploits can directly affect the execution of an application-level firewall. Exploits combine network activity from a user-level component along with a

kernel-level module that hides the user-level component from the view of the application-level firewall. The malicious exploit likely obtains full-system privileges and can thus halt the execution of the firewall. The malicious kernel-level module can alter the hooks used by the in-kernel module of the application-level firewall so that the firewall is simply never invoked as data passes through the network. Conventional application-level firewalls fail under these types of attacks. AL-SAFE withstands attacks from these types of exploits. However, AL-SAFE can still retrieve a maliciously crafted connection list and allow connections for malicious applications that impersonate legitimate white-listed applications. Compared to a traditional application-level firewall which operates inside the host and if compromised can open any port regardless if it is in the white-list or not, AL-SAFE does not open any not white-listed port.

AL-SAFE denies all unknown connections by default. In a production system where services have sufficiently long life-times, this tackles the case of an attacker timing the introspection period and attempting to use the network between two consecutive introspections. The performance overhead of this choice on each connection is outlined in Section 5.6.1.5.

Finally we analyze the potential vulnerabilities added by AL-SAFE to the provider infrastructure. AL-SAFE's components are exposed to three kinds of potentially malicious input. First, the white-list of processes, second the added rules and third the introspection results. The design choices for the three items (as presented in Section 5.3.1) address the issue of malicious input.

We now discuss AL-SAFE's limitations and our suggested approach for handling them.

5.6.3 Limitations

AL-SAFE as an application-level firewall located outside the monitored VM, is able to provide, through virtual machine introspection, the same degree of visibility as an inside the host solution. However, AL-SAFE suffers from some limitations. We detail these limitations based on their category:

- **Performance**: AL-SAFE performs introspection periodically, which delays the network connectivity of newly started services and clients. To reduce this overhead, AL-SAFE could introspect on watchpoints, e.g. on *listen()* and *connect()* syscalls on TCP sockets.

- **Security**: As all introspection-based security solutions AL-SAFE is vulnerable to kernel data structure manipulation. An attacker who fully controls the VM can also tamper with kernel data structures to control introspection results. To counter such attacks we could use approaches to check the VM's kernel integrity [109]. Furthermore, an additional security impediment would be a previous legitimate process that has turned malicious. An attacker can hijack a connection after it has been established and verified by AL-SAFE as legitimate. It can use a software exploit to take control of a particular process bound to the port or use a kernel module to alter packets before they are sent out to the local switch network interface. To counter this issue we could place dedicated Intrusion Detection Systems in the infrastructure, using the approach of SAIDS.

5.7 Summary

In this chapter we presented AL-SAFE, the second instantiation of our security monitoring framework which focuses on firewalls. AL-SAFE is a secure application-level introspection-based firewall that is able to adapt the enforced ruleset based on changes in the virtual infrastructure topology and the list of services running in the monitored VMs. AL-SAFE's design addresses the inherent design limitation of application level-firewalls that run inside the monitored VMs. Hence, they can be compromised by malicious kernel-level code that is executed inside the monitored host. Using virtual machine introspection AL-SAFE pulls the firewall outside the untrusted VM while maintaining an inside-the-VM visibility. AL-SAFE filters traffic at two distinct points in the virtual infrastructure regulating the load imposed on other security devices, which are part of our framework such as intrusion detection systems.

We have conducted a thorough evaluation of our approach examining both performance and correctness aspects. We have shown that the overhead in cloud operations such as VM migration is independent from the VM workload. This overhead is lower than the migration time. Our results show a dependency between the introspection period and the generated overhead for tenant applications running inside the untrusted VM. Increasing the introspection period depending on the type of activity inside the VM (fewer introspections for compute-intensive applications that do not use the network) could significantly decrease the overhead. Our prototype already features a dedicated mechanism for adapting the introspection period on the fly.

Finally, we have shown that AL-SAFE correctly blocks unauthorized connections while allowing all tenant-approved connections to pass unimpeded. The design choices made for AL-SAFE's components do not add any security vulnerabilities in the provider's infrastructure. AL-SAFE's limitations both from a security (kernel data structure manipulation) and performance aspects (delay of network connectivity) were presented along with suggestions on how they can be addressed.

Chapter 6

Conclusion

This chapter summarizes our contributions and details how these contributions fulfil the objectives presented in Section 1.3. The contributions along with their assessment are listed in Section 6.1 while suggestions for future research work are presented in Section 6.2.

6.1 Contributions

In this thesis we designed a self-adaptable security monitoring framework that is able to adapt its components based on dynamic events that occur in a cloud infrastructure. Four main objectives were defined: *self-adaptation*, *tenant-driven customization*, *security* and *cost minimization*. Our framework achieves these objectives and constitutes a flexible monitoring solution for virtualized infrastructures that is able to integrate different types of monitoring devices. Two different instantiations of our framework, SAIDS and AL-SAFE were presented in detail.

SAIDS, a self-adaptable network intrusion detection system uses Local Intrusion Detection Sensors (LIDS) in order to monitor traffic towards and from the cloud infrastructure. SAIDS reaction to different types of dynamic events was presented and justified in order to provide the reader with a clear overview of the adaptation process. The first instantiation of our framework is a scalable solution that can alter the existing configuration and the computational resources available to a set of LIDSs depending on the load of monitored traffic while maintaining an adequate level of detection. SAIDS prototype was developed using different cloud technologies (e.g. OpenStack [32], Open vSwitch [137]). Our evaluation under different scenarios that resemble production environments allowed us to assess SAIDS performance, scalability, and correctness. Our results showed that SAIDS is able to handle 5000 LIDS (evaluation performed on 8 core machines with 24GB of RAM each – our testbed's machines memory capacity imposed a limitation on the number of LIDSs that our prototype can handle in parallel) while imposing negligible overhead to cloud operations.

AL-SAFE is the second instantiation of our security monitoring framework which focuses on an active monitoring component, the firewall. AL-SAFE is executed outside the monitored VMs and filters traffic at distinct points of the virtual infrastructure combining an edge firewall, located at the interface between the cloud network and the external network, with a switch-level firewall. We proved that our design is able to address the inherent design limitation of application-level firewalls (malicious code exposure due to inside-the-host execution) and at the same time maintain an inside-the-host level of visibility through virtual machine introspection. The adaptation of AL-SAFE's enforced ruleset for different types of dynamic events was thoroughly detailed, followed by a jus-

tification of subsequent actions. Finally, our evaluation presented a comprehensive study of the trade-offs between the security, adaptation benefits of deploying AL-SAFEand the performance overhead imposed on cloud operations and tenant applications hosted in the virtual infrastructure. Our results have shown that the overhead imposed by AL-SAFE on new sockets of network-oriented tenant applications highly depends on the arrival time of the connection request in the introspection period.

We now discuss how our work addresses the four objectives that were defined in the introduction of this thesis.

Self-adaptation: Our framework is able to adapt its components based on three types of dynamic events that occur in a cloud infrastructure: *topology-related*, *service-related* and *monitoring load-related* events. Our framework's core component, the Adaptation Manager, is able to make adaptation decisions based on the type of event and act as a coordinator of the adaptation process synchronizing the different components involved. The AM as a high-level component guarantees that the adaptation decision remains abstracted from the type of the monitoring device, providing our framework with another level of genericness. Both SAIDS and AL-SAFE adapt their enforced rulesets upon receiving the adaptation arguments from the AM. In order to guarantee the accurate translation of the adaptation arguments to device-specific configuration parameters both SAIDS and AL-SAFE feature dedicated components that interact with the actual monitoring devices. In both SAIDS and AL-SAFE prototypes we integrate different off-the-self components (2 NIDSs and 2 firewalls) with no modifications in their code.

Tenant-driven customization: Our framework takes into account tenant-defined monitoring requirements as they are expressed through a dedicated API. These requirements may refer to monitoring tailored for specific services that are deployed in the virtual infrastructure or to performance-related metrics. The Adaptation Manager guarantees that the tenant requests will be taken into account in the adaptation decision and will be propagated to lower level agents, the Master Adaptation Drivers (MADs), that will translate them to device-specific configuration parameters. Our framework supports dedicated actions in case a tenant-defined requirement is violated (e.g. altering the computational resources available to a monitoring device in case a performance-related metric exceeds a tenant-defined threshold).

Security: Our framework is able to guarantee that the adaptation process will not add any security flaw in the monitoring device itself or in the provider's infrastructure. Our design choices have proven that the different elements that participate in the adaptation of a monitoring device (i.e adaptation sources, input files, etc) do not add any new security flaws and do not create any potential entry point for the attacker. Furthermore, in both of our frameworks instantiations we have experimentally validated the correctness of the adaptation result. The monitoring devices continue to remain operational during the adaptation process, guaranteeing that an adequate level of detection is maintained.

Cost minimization: Our framework is able to guarantee that the cost for both tenants and the provider in terms of application performance and computational resources is kept at a minimal level. SAIDS evaluation results showed that our framework's instantiation imposes negligible overhead in normal cloud operations. Regarding computational resources (CPU and RAM) deploying SAIDS bears minimal cost. As a passive monitoring solution, SAIDS does not directly affect the performance of network-oriented cloud applications. AL-SAFE's overhead in normal cloud operations does not depend on the VM workload while the CPU and memory cost is tolerable. AL-SAFE follows a time-based introspection model and as such the overhead for new sockets of network-oriented

tenant applications highly depends on the arrival time of the connection request in the introspection period.

The work presented in this thesis was able to address the gap in existing cloud security monitoring frameworks regarding reaction to dynamic events. Existing solutions only partially address the defined objectives (as described in Section 1.3) while our framework is able to combine self-adaptation based on dynamic events with accurate security monitoring results.

This thesis presented the design of a self-adaptable security monitoring framework that is able to adapt its components based on different types of dynamic events that occur in a cloud infrastructure. SAIDS and AL-SAFE the framework's two instantiations, addressed self-adaptation for two different types of security devices, intrusion detection systems and firewalls. Naturally, the work done in this thesis can be extended. We have identified several directions of improvement that would lead to a complete self-contained monitoring infrastructure. We discuss these directions in the next section.

6.2 Future Work

Our future work is organized in three categories depending on feasibility and time required to complete the described improvements. In Section 6.2.1 we present a few short term goals that constitute performance and design improvements of our existing instantiations. Section 6.2.2 focuses on other components of our framework while Section 6.2.3 concludes this chapter with our vision regarding a self-contained security monitoring framework.

6.2.1 Short-Term Goals

Different design and performance improvements regarding SAIDS and AL-SAFE prototypes could be realised.

SAIDS: Currently SAIDS does not feature a mechanism for automatic discovery of new services that are deployed in the monitored VMs. The only way for SAIDS to become aware of a change in the list of running services (and subsequently reconfigure the involved LIDSs) is through our framework's dedicated API needing the tenant to declare that a service was started or stopped. A solution for automatic service discovery would be for SAIDS to use AL-SAFE's periodic introspection results. Each time a new legitimate service is detected in a VM by introspection, the Adaptation Manager could trigger an adaptation of the enforced ruleset of the LIDS responsible for monitoring the traffic that flows towards and from that particular VM. The addition of automatic service discovery does not require a significant change in the existing SAIDS design since the Adaptation Manager is currently shared between the two instantiations.

AL-SAFE: As demonstrated by the performance evaluation of AL-SAFE, periodic introspection imposes unnecessary overhead to applications that are not network-intensive (see the kernel-build results). A solution would be to correlate the type of application activity with the introspection period for example computation-intensive applications can have a larger introspection period than network-intensive applications. Furthermore, instead of a periodic introspection period AL-SAFE could adopt a watchpoint-based introspection model in which the VMI component could introspect every time a specific event occurs (e.g. a *listen* syscall on a TCP socket). Finally, our results have shown that the response time of the introspection component is not negligible. In order to improve the response time of this component and subsequently decrease the overhead imposed by the adaptation loop on new connections, introspection could be optimized by introspecting directly on LibVMI

rather than a combination of LibVMI and Volatility. This change implies implementing a version of the *netstat* command using the VM's memory pages exported by LibVMI and necessary information regarding kernel data structures. Introspecting directly on LibVMI holds an additional advantage, the removal of the snapshotting phase, that was necessary for providing a coherent address space to volatility. This improvement will significantly reduce the memory consumption of the VMI component.

Dependency Database: Currently all the necessary information for the monitoring devices involved in the monitoring of a particular VM is contained in simple text files. Although in Chapter 3 we defined that this information should be stored in the *Dependency Database* we did not have time to implement this component. Including a relational database (e.g. MySQL [152]) for storing this information in the existing framework implementation should require minimal changes to the the Adaptation Manager component. These changes are necessary in order to facilitate connection with the database as well as exchange of information for the latter a message protocol could be used like for example RabbitMQ [153].

Tenant API: Currently, the tenant API as defined in Chapter 3 has not yet been implemented. A simple Restful interface can be used in order to provide the translation between high-level tenant monitoring requirements and Adaptation Manager-readable adaptation arguments.

6.2.2 Mid-Term Goals

This section focuses on expansion of our framework to other types of security devices as well as addressing aspects like multitenancy and combining security monitoring for tenants and the provider.

Other types of security devices: In order to extend the monitoring capabilities of our framework to other types of monitoring (e.g. inside-the-host activity monitoring, network flow analysis) other types of monitoring devices need to be included. Currently, our self-adaptable security monitoring framework includes only network-based intrusion detection systems and firewalls. A possible improvement would be to include host-based IDSs or network analyzers like Bro [154]. Since we plan to include other types of IDSs the changes required would primarily refer to SAIDS. Many host-based IDSs operate based on agents that are installed locally inside the monitored VM and perform different types of monitoring (e.g file integrity checking, rootkit detection, etc). These agents communicate with a central manager and periodically report findings. In order to support this model, the design changes required for SAIDS are two fold. First, at the level of the Master Adaptation Driver. Instead of being responsible for regenerating the actual configuration file for the IDS (like in the LIDS case) the MAD could simply forward relevant monitoring parameters to the appropriate Adaptation Worker (AW). Depending on the number of agents reporting to each AW the MAD could also adapt the portion of computational resources available to each AW in order to perform load balancing. Second, at the level of the Adaptation Worker, instead of having one AW per detection process, a single AW instance could be responsible for all detection agents running inside a group of VMs (e.g. all the VMs residing in the same compute node). Since most detection agents support remote configuration through a secure connection, the AW could be located in a separate domain, introducing another security layer between a potentially compromised detection agent and the SAIDS component.

Other types of security devices that could be included are log collectors and aggregators. In order to satisfy the cost minimization objective, a log collector instance would be

responsible for gathering and unifying the logs produced by a subset of monitoring devices (e.g. all the devices that monitor VMs that reside on the same compute node). Regarding the tenant-driven customization objective, the collector would apply special filters to the collected logs (e.g. if a specific attack for which tenants have requested additional monitoring has been detected or if the number of attacks in a specific time window has exceeded a certain threshold) and propagate the results to an aggregator instance. Tenants could access the aggregated logs through a dedicated mechanism that guarantees authentication and data integrity, satisfying the correctness objective. Different policies, designed to cope with the scale of the system and adapt the number of collectors and aggregators, could be defined in order to address the self-adaptation objective.

Multi-tenancy: The current version of our security monitoring framework does not address implications that arise in multi-tenant environments. In order to enable security monitoring for different tenants, we need to consider the sharing of the monitoring devices between tenants. Component sharing between tenants can also be perceived as an additional aspect of cost minimization. We now discuss the necessary changes in our two instantiations SAIDS and AL-SAFE in order to enable component sharing.

Since each tenant has its own network, and legacy network intrusion detection systems do not support monitoring two different networks with the same NIDS instance, SAIDS will have to assign separate LIDSs to different tenants. However, the remaining components (Adaptation Manager and Master Adaptation Driver) can still be shared between tenants. In order to differentiate between LIDS that belong to different tenants, an extra field indicating the ID of the tenant that this device is assigned to can be added in the set of information stored for each LIDS probe. Each MAD could maintain a per-tenant list with all the LIDS names that are under its control. Our evaluation results have shown that both the MAD and the AM can handle multiple adaptation requests in parallel, thus enabling parallel adaptation of LIDS that belong to different tenants.

For AL-SAFE device sharing implies using the same firewall (either switch-level or edge) for filtering traffic towards and from VMs that belong to different tenants. Since the filtering is performed by dedicated rules, installing rules for different VMs in the same firewall device is straightforward. In order to address simultaneous dynamic events that concern different tenants, parallel generation of filtering rules that concern different VMs is necessary. Unfortunately, in the current version of AL-SAFE parallel rule generation is only supported for VMs that reside in different compute nodes. For enabling parallel generation of rules for VMs that reside in the same compute node, parallel introspection of those VMs is needed. Unfortunately, in the current implementation the VMI component does not support parallel introspection of collocated VMs (because it is single-threaded). Consequently, the core change for AL-SAFE supporting multi-tenancy requires making VMI multithreaded. A multi-threaded VMI component that introspects directly on LibVMI will also impose a significantly lower memory overhead.

Combining the security monitoring of tenants and the provider: In a cloud environment the provider could assume the role of a super-tenant. This is essentially translated to a tenant with increased privileges who also requires adequate security monitoring of its infrastructure and adaptation of the security devices in case of dynamic events. The existence of a super-tenant raises two research questions: first, the necessary design changes for our security monitoring framework in order to support the different roles. In the case of SAIDS this would imply a number of dedicated LIDS instances that monitor the provider's traffic and are possibly located in an isolated node without any other tenant-LIDS. For AL-SAFE this would imply that provider-related rules are injected and enforced in the two types of firewalls. Second, an agreement for a fair sharing of moni-

toring resources between the tenants and the super-tenant (i.e. the provider) needs to be defined in order to guarantee that the monitoring devices dedicated to tenants will always have access to the necessary computational resources. An adaptable threshold regarding the percentage of monitoring resources dedicated to the provider should be agreed between tenants and the provider and included in the SLA. Furthermore, a framework for translating the threshold value to specific monitoring parameters (e.g. how many rules in a shared firewall can the provider install) needs to be realized. This research question is closely related to another PhD thesis in Myriads team entitled *Definition and enforcement of service level agreements for cloud security Monitoring.*

Integration of SAIDS in a large-scale system: Qirinus [155], a start-up that specializes in automatic deployment of security monitoring infrastructures for cloud environments, plans to integrate SAIDS in their system. The integration would allow tenants to use the Qirinus API in order to provide a high-level description of their system along with specific security requirements which will then be translated to SAIDS-specific arguments. The Qirinus system will also be responsible for automating the deployment of SAIDS individual components in the virtual infrastructure in such way that the tenant-defined requirements are respected. Integrating SAIDS with Qirinus will enable the transfer of SAIDS technology to real world large-scale scenarios.

Handling network reconfiguration events: Currently our self-adaptable security monitoring framework does not handle network reconfiguration events although they are considered topology-related changes. Indeed, these types of events, for example migrating a VM between networks, bare some resemblances with events that refer to the placement of VMs (in this case with a VM migration). These resemblances allow us to consider that significant similarities will occur between the adaptation process that follows a VM-placement event and the adaptation process that follows a network-reconfiguration event. For example, in SAIDS and AL-SAFE, the difference between the necessary reconfigurations in a VM-placement dynamic event and a network reconfiguration dynamic event would consist in changing the IP addresses (internal and external) in the rules related to the VMs.

6.2.3 Long-Term Goals

As a long-term research direction, we consider the design of a fully autonomous self-adaptable security monitoring framework. A fully autonomous monitoring framework should be able to react to security events and take subsequent actions in order to isolate potentially infected VMs and stop attackers from gaining control of the virtual infrastructure. Reaction is essentially based on the ability of the framework to translate security monitoring findings (e.g. IDS alerts) to adaptation decisions that affect the configuration of the monitoring devices. In the context of this thesis, such an ability is linked to including security events to the set of possible adaptation sources. Currently our self-adaptable security monitoring framework supports adaptation of the security devices based on three different types of dynamic events: topology-, service- and monitoring load-related events. Security events (i.e. attacks) as a potential adaptation source were not considered.

In our framework a reaction mechanism could operate by transferring SAIDS-generated alerts to AL-SAFE and translating them to filtering rules. The primary functionality of this mechanism would be to extract all related information from the alert (IP address, protocol, port, etc) and propagate it through a secure channel to AL-SAFE's Information Extraction Agent. The main challenge behind this mechanism is the determination of the correct Information Extraction Agent (since in our cloud environment one IEA is installed

in each compute node). In order to determine the right IEA host, the mechanism needs to obtain a partial topological and functional overview of the monitoring framework. We define as partial overview the topological and functional information that refers only to a subset of security devices, for example all the monitoring devices that are under the control of a specific Master Adaptation Driver instance (as opposed to a complete overview where the functional and topological overview refer to system-wide information).

Adding security to the set of possible adaptation sources opens a convergence area with the VESPA architecture [124] and will allow us to create a fully autonomous self-adaptable security monitoring framework that considers security- as well as infrastructure-related dynamic events.

Bibliography

[1] S. Roschke, F. Cheng, and C. Meinel, "Intrusion Detection in the Cloud," in *Dependable, Autonomic and Secure Computing, 2009. DASC '09. Eighth IEEE International Conference on*, pp. 729–734, Dec 2009.

[2] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *In Proceedings Network and Distributed Systems Security Symposium*, pp. 191–206, 2003.

[3] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "CloudSec: A security monitoring appliance for Virtual Machines in the IaaS cloud model," in *Network and System Security (NSS), 2011 5th International Conference on*, pp. 113–120, Sept 2011.

[4] D. Dib, N. Parlavantzas, and C. Morin, "SLA-Based Profit Optimization in Cloud Bursting PaaS," in *2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pp. 141–150, May 2014.

[5] T. Mather, S. Kumaraswamy, and S. Latif, *Cloud Security and Privacy: An Enterprise Perspective on Risks and Compliance.* O'Reilly Media, Inc., 2009.

[6] "Top 12 cloud computing threats." `https://downloads.cloudsecurityalliance.org/assets/research/top-threats/Treacherous-12_Cloud-Computing_Top-Threats.pdf`. Accessed: 2016.

[7] "Amazon Web Services as a DDoS Launch Hub ." `https://vpncreative.net/2014/07/29/hackers-sneak-back-aws-ddos-launch-hub/`. Accessed: 2017.

[8] N. u. h. Shirazi, S. Simpson, A. K. Marnerides, M. Watson, A. Mauthe, and D. Hutchison, "Assessing the Impact of Intra-Cloud Live Migration on Anomaly Detection," in *Proceedings CloudNet*, 2014.

[9] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, Jan 2003.

[10] M. C. Huebscher and J. A. McCann, "A Survey of Autonomic Computing: Degrees, Models, and Applications," *ACM Comput. Surv.*, vol. 40, pp. 7:1–7:28, Aug. 2008.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Tech. Rep. UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[12] P. M. Mell and T. Grance, "SP 800-145. The NIST Definition of Cloud Computing," tech. rep., Gaithersburg, MD, United States, 2011.

[13] F. Liu, J. Tong, J. Mao, R. Bohn, J. Messina, L. Badger, and D. Leaf, *NIST Cloud Computing Reference Architecture: Recommendations of the National Institute of Standards and Technology (Special Publication 500-292).* USA: CreateSpace Independent Publishing Platform, 2012.

[14] S. Kächele, C. Spann, F. J. Hauck, and J. Domaschka, "Beyond IaaS and PaaS: An Extended Cloud Taxonomy for Computation, Storage and Networking," in *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, UCC '13, (Washington, DC, USA), pp. 75–82, IEEE Computer Society, 2013.

[15] "Amazon Web Services." `https://aws.amazon.com/ec2/`. Accessed: 2015.

[16] "Google Compute Engine." `https://cloud.google.com/compute/`. Accessed: 2015.

[17] "OVH Public CLoud." `https://www.ovh.com/fr/cloud/`. Accessed: 2015.

[18] "VMware vCloud Suite." `http://www.vmware.com/products/vcloud-suite.html`. Accessed: 2015.

[19] "HPE Helion Eucalyptus." `http://www8.hp.com/us/en/cloud/helion-eucalyptus.html`. Accessed: 2017.

[20] "Nimbus Infrastructure." `http://www.nimbusproject.org/`. Accessed: 2017.

[21] D. Miloji, I. M. Llorente, and R. S. Montero, "OpenNebula: A Cloud Management Tool," *IEEE Internet Computing*, vol. 15, pp. 11–14, March 2011.

[22] O. Sefraoui, M. Aissaoui, and M. Eleuldj, "Article: OpenStack: Toward an Open-source Solution for Cloud Computing," *International Journal of Computer Applications*, vol. 55, pp. 38–42, October 2012. Full text available.

[23] "Google App Engine." `https://cloud.google.com/appengine/`. Accessed: 2017.

[24] "Microsoft Azure." `https://azure.microsoft.com/en-us/`. Accessed: 2017.

[25] "Google Apps." `https://gsuite.google.com`. Accessed: 2017.

[26] "iCLoud." `https://www.icloud.com`. Accessed: 2017.

[27] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, "KVM: the Linux Virtual Machine Monitor," in *In Proceedings of the 2007 Ottawa Linux Symposium (OLS-07*, 2007.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SIGOPS Oper. Syst. Rev.*, vol. 37, pp. 164–177, Oct. 2003.

[29] C. A. Waldspurger, "Memory Resource Management in VMware ESX Server," in *Proceedings of the 5th Symposium on Operating Systems Design and implementation-Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, (Berkeley, CA, USA), pp. 181–194, USENIX Association, 2002.

[30] "Production-Grade Container Orchestration." `https://kubernetes.io/`. Accessed: 2017.

[31] R. Moreno-Vozmediano, R. Montero, and I. Llorente, "IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures," *Computer*, vol. 45, pp. 65–72, Dec. 2012.

[32] "OpenStack." `http://www.openstack.org/`. Accessed: 2017.

[33] G. J. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable Third Generation Architectures," *Commun. ACM*, vol. 17, pp. 412–421, July 1974.

[34] "Bochs IA-32 Emulator Project." `http://bochs.sourceforge.net`. Accessed: 2017.

[35] "QEMU Open Source Processor Emulator." `http://wiki.qemu.org/Main_Page`. Accessed: 2017.

[36] K. Adams and O. Agesen, "A Comparison of Software and Hardware Techniques for x86 Virtualization," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, (New York, NY, USA), pp. 2–13, ACM, 2006.

[37] "VirtualBox." `https://www.virtualbox.org`. Accessed: 2017.

[38] "VMware Fusion." `http://www.vmware.com/products/fusion.html`. Accessed: 2017.

[39] "VMware Workstation." `http://www.vmware.com/products/workstation.html`. Accessed: 2017.

[40] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith, "Intel virtualization technology," *Computer*, vol. 38, pp. 48–56, May 2005.

[41] "Microsoft Hyper-V." `https://technet.microsoft.com/library/hh831531.aspx`. Accessed: 2017.

[42] "Xen Hardware Virtual Machine." `https://wiki.xen.org/wiki/Xen_Project_Software_Overview#HVM`. Accessed: 2017.

[43] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, pp. 81–84, Sept 2014.

[44] "LXC linux containers." `https://linuxcontainers.org/`. Accessed: 2017.

[45] "Docker containers." `https://www.docker.com/what-docker`. Accessed: 2017.

[46] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," 2001.

[47] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker, "Ethane: Taking Control of the Enterprise," *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 1–12, Aug. 2007.

[48] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 69–74, Mar. 2008.

[49] A. Doria, J. H. Salim, R. Haas, W. Wang, L. Dong, and R. Gopal, "Forwarding and Control Element Separation (ForCES) Protocol Specification," 2010.

[50] H. Song, "Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane," in *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, (New York, NY, USA), pp. 127–132, ACM, 2013.

[51] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: Towards an Operating System for Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 105–110, July 2008.

[52] "OpenDaylight: Open Source SDN Platform." `https://www.opendaylight.org`. Accessed: 2017.

[53] "Project Floodlight." `http://www.projectfloodlight.org/floodlight/`. Accessed: 2017.

[54] R. Sherwood, G. Gibb, K. kiong Yap, M. Casado, N. Mckeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," tech. rep., 2009.

[55] D. Drutskoy, E. Keller, and J. Rexford, "Scalable Network Virtualization in Software-Defined Networks," *IEEE Internet Computing*, vol. 17, pp. 20–27, March 2013.

[56] "VMware NSX." `http://www.vmware.com/products/nsx`. Accessed: 2015.

[57] C. S. Li, B. L. Brech, S. Crowder, D. M. Dias, H. Franke, M. Hogstrom, D. Lindquist, G. Pacifici, S. Pappe, B. Rajaraman, J. Rao, R. P. Ratnaparkhi, R. A. Smith, and M. D. Williams, "Software defined environments: An introduction," *IBM Journal of Research and Development*, vol. 58, pp. 1:1–1:11, March 2014.

[58] "SQL Injection." `https://www.owasp.org/index.php/SQL_Injection`. Accessed: 2017.

[59] "Cross Site Scripting ." `https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)`. Accessed: 2017.

[60] "Aleph One. Smashing The Stack For Fun And Profit ." `http://insecure.org/stf/smashstack.html`. Accessed: 2017.

[61] "Tim Newsham. Format String Attacks." `http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf`. Accessed: 2017.

[62] L. T. Heberlein and M. Bishop, "Attack Class: Address Spoofing," in *In Proceedings of the 19th National Information Systems Security Conference*, pp. 371–377, 1996.

[63] W. M. Eddy, "TCP SYN flooding attacks and common mitigations," 2007.

[64] N. Karapanos and S. Capkun, "On the Effective Prevention of TLS Man-in-the-middle Attacks in Web Applications," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, (Berkeley, CA, USA), pp. 671–686, USENIX Association, 2014.

[65] "DNS flaw for cache poisoning attacks." http://www.kb.cert.org/vuls/id/800113. Accessed: 2017.

[66] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the Slammer Worm," *IEEE Security and Privacy*, vol. 1, pp. 33–39, July 2003.

[67] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 199–212, ACM, 2009.

[68] " N. Elhage. Virtunoid: Breaking out of KVM ." nelhage.com/talks/kvm-defcon-2011.pdf. Accessed: 2017.

[69] "Buffer Overflow in the Backend of XenSource ." https://web.nvd.nist.gov/view/vuln/search-results?query=CVE-2008-1943&search_type=all&cves=on. Accessed: 2017.

[70] "ENISA cloud risk assessment ." https://www.enisa.europa.eu/publications/cloud-computing-risk-assessment. Accessed: 2017.

[71] J. Somorovsky, M. Heiderich, M. Jensen, J. Schwenk, N. Gruschka, and L. Lo Iacono, "All Your Clouds Are Belong to Us: Security Analysis of Cloud Management Interfaces," in *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, (New York, NY, USA), pp. 3–14, ACM, 2011.

[72] R. Bejtlich, *The Tao of Network Security Monitoring: Beyond Intrusion Detection*. Pearson Education, 2004.

[73] "Kaspersky Security Scanner." https://www.kaspersky.com/free-virus-scan. Accessed: 2015.

[74] "AVG Antivirus." http://www.avg.com/ww-en/free-antivirus-download. Accessed: 2015.

[75] "Panda Protection." http://www.pandasecurity.com/france/homeusers/solutions/activescan/?ref=activescan. Accessed: 2017.

[76] "Cisco ASR 1000 Series." http://www.cisco.com/c/en/us/products/routers/asr-1000-series-aggregation-services-routers/index.html. Accessed: 2015-09-29.

[77] "Juniper MX Series." http://www.juniper.net/us/en/products-services/routing/mx-series/. Accessed: 2016.

[78] T. Dierks and C. Allen, "The TLS Protocol Version 1.0," 1999.

[79] S. Kent and R. Atkinson, "Security Architecture for the Internet Protocol," 1998.

[80] E. Rescorla and N. Modadugu, "Datagram Transport Layer Security Version 1.2." RFC 6347, Jan. 2012.

[81] "VPN Express." https://www.expressvpn.com/. Accessed: 2017.

[82] "VyPR VPN." https://www.goldenfrog.com/vyprvpn. Accessed: 2017.

[83] M. Wahl, H. Alvestrand, J. Hodges, and R. Morgan, "Authentication Methods for LDAP," 2000.

[84] J. Kohl and C. Neuman, "The Kerberos Network Authentication Service (V5)," 1993.

[85] "Amazon CloudWatch." `https://aws.amazon.com/cloudwatch/`. Accessed: 2016.

[86] "Microsoft Azure Log Analytics." `https://azure.microsoft.com/en-us/services/log-analytics/`. Accessed: 2015.

[87] "rSyslog." `http://www.rsyslog.com/`. Accessed: 2015.

[88] "LogStash." `https://www.elastic.co/products/logstash`. Accessed: 2015.

[89] K. A. Scarfone and P. M. Mell, "SP 800-94. Guide to Intrusion Detection and Prevention Systems (IDPS)," tech. rep., Gaithersburg, MD, United States, 2007.

[90] S. Axelsson, "Intrusion Detection Systems: A Survey and Taxonomy," tech. rep., 2000.

[91] C. Modi, D. Patel, B. Borisaniya, H. Patel, A. Patel, and M. Rajarajan, "A survey of intrusion detection techniques in cloud," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 42–57, 2013.

[92] P. García-Teodoro, J. Díaz-Verdejo, G. Maciá-Fernández, and E. Vázquez, "Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges," *Comput. Secur.*, vol. 28, pp. 18–28, Feb. 2009.

[93] "Snort." `http://www.snort.org/`. Accessed: 2015.

[94] "Suricata Open Source IDS Engine." `https://suricata-ids.org`. Accessed: 2015.

[95] "The Sagan Log Analyis Engine." `https://quadrantsec.com/sagan_log_analysis_engine/`. Accessed: 2015.

[96] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-time," *Comput. Netw.*, vol. 31, pp. 2435–2463, Dec. 1999.

[97] "Stealthwatch Flow Collector." `https://www.lancope.com/products/stealthwatch-flowcollector`. Accessed: 2015.

[98] "Cisco NGIPS." `http://www.cisco.com/c/en/us/products/security/ngips/index.html`. Accessed: 2015.

[99] "Towards a Taxonomy of Intrusion-detection Systems," *Comput. Netw.*, vol. 31, pp. 805–822, Apr. 1999.

[100] W. R. Cheswick, S. M. Bellovin, and A. D. Rubin, *Firewalls and Internet Security: Repelling the Wily Hacker*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2 ed., 2003.

[101] A. D. Keromytis and V. Prevelakis, *Designing Firewalls: A Survey*, pp. 33–49. John Wiley and Sons, Inc., 2006.

[102] K. A. Scarfone and P. Hoffman, "SP 800-41 Rev. 1. Guidelines on Firewalls and Firewall Policy," tech. rep., Gaithersburg, MD, United States, 2009.

[103] "Trusted Code Base." `https://www.ibm.com/support/knowledgecenter/linuxonibm/liaat/liaatsectcb.htm`. Accessed: 2016.

[104] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A Hardware-assisted Integrity Monitor," in *Proceedings of the 13th International Conference on Recent Advances in Intrusion Detection*, RAID'10, (Berlin, Heidelberg), pp. 158–177, Springer-Verlag, 2010.

[105] A. M. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. C. Skalsky, "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity," in *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, (New York, NY, USA), pp. 38–49, ACM, 2010.

[106] Z. Wang and X. Jiang, "HyperSafe: A Lightweight Approach to Provide Lifetime Hypervisor Control-Flow Integrity," in *2010 IEEE Symposium on Security and Privacy*, pp. 380–395, May 2010.

[107] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh, "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, (Berkeley, CA, USA), pp. 13–13, USENIX Association, 2004.

[108] Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering Kernel Rootkits with Lightweight Hook Protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, (New York, NY, USA), pp. 545–554, ACM, 2009.

[109] A. Baliga, V. Ganapathy, and L. Iftode, "Automatic Inference and Enforcement of Kernel Data Structure Invariants," in *Proceedings of the 2008 Annual Computer Security Applications Conference*, ACSAC '08, (Washington, DC, USA), pp. 77–86, IEEE Computer Society, 2008.

[110] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the Semantic Gap in Virtual Machine Introspection," in *2011 IEEE Symposium on Security and Privacy*, pp. 297–312, May 2011.

[111] S. Bahram, X. Jiang, Z. Wang, M. Grace, J. Li, D. Srinivasan, J. Rhee, and D. Xu, "DKSM: Subverting Virtual Machine Introspection for Fun and Profit," in *2010 29th IEEE Symposium on Reliable Distributed Systems*, pp. 82–91, Oct 2010.

[112] B. D. Payne, D. D. A. Martim, and W. Lee, "Secure and flexible monitoring of virtual machines," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pp. 385–397, IEEE, 2007.

[113] "LibVmi." `https://github.com/libvmi/libvmi/releases`. Accessed: 2016.

[114] C. Mazzariello, R. Bifulco, and R. Canonico, "Integrating a network IDS into an open source Cloud Computing environment," in *Information Assurance and Security (IAS), 2010 Sixth International Conference on*, pp. 265–270, Aug 2010.

[115] M. Ficco, L. Tasquier, and R. Aversa, "Intrusion Detection in Cloud Computing," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*, pp. 276–283, Oct 2013.

[116] K. Kourai and S. Chiba, "HyperSpector: Virtual Distributed Monitoring Environments for Secure Intrusion Detection," in *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, VEE '05, (New York, NY, USA), pp. 197–207, ACM, 2005.

[117] B. D. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pp. 233–247, IEEE, 2008.

[118] F. Lombardi and R. Di Pietro, "KvmSec: A Security Extension for Linux Kernel Virtual Machines," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, (New York, NY, USA), pp. 2029–2034, ACM, 2009.

[119] "VMware inc. Next Generation Firewalls." `http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/whitepaper/products/nsx/vmware-nsx-palo-alto-networks-white-paper.pdf`. Accessed: 2017.

[120] "Amazon Web Services- Web Application Firewall." `https://aws.amazon.com/blogs/aws/new-aws-waf/`. Accessed: 2017.

[121] "SteelApp Web Application Firewall." `https://support.riverbed.com/content/support/software/steelapp/web-app-firewall.html`. Accessed: 2015.

[122] A. Srivastava and J. Giffin, *Tamper-Resistant, Application-Aware Blocking of Malicious Network Connections*, pp. 39–58. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008.

[123] K. Kourai, T. Azumi, and S. Chiba, "A Self-Protection Mechanism against Stepping-Stone Attacks for IaaS Clouds," in *2012 9th International Conference on Ubiquitous Intelligence and Computing and 9th International Conference on Autonomic and Trusted Computing*, pp. 539–546, Sept 2012.

[124] A. Wailly, M. Lacoste, and H. Debar, "VESPA: Multi-layered Self-protection for Cloud Resources," in *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pp. 155–160, 2012.

[125] "Amazon Web Services Identity and Access Management." `https://aws.amazon.com/iam/details/`. Accessed: 2017.

[126] "Google Identity and Access Management." `https://cloud.google.com/iam/`. Accessed: 2017.

[127] "Microsoft Identity Manager." `https://www.microsoft.com/en-us/cloud-platform/microsoft-identity-manager`. Accessed: 2017.

[128] "Amazon Web Services, Shield." `https://aws.amazon.com/shield/?hp=tile&so-exp=below`. Accessed: 2017.

[129] "Amazon Web Services Security Products." `https://aws.amazon.com/products/`. Accessed: 2017.

[130] "Google Security Scanner." https://cloud.google.com/security-scanner/. Accessed: 2017.

[131] "Google Resource Manager." https://cloud.google.com/resource-manager/. Accessed: 2017.

[132] "VMware inc. SpoofGuard." https://pubs.vmware.com/NSX-6/index.jsp?topic=%2Fcom.vmware.nsx.admin.doc%2FGUID-F11F7B52-70EB-4532-9E0E-2FCB64707A1D.html. Accessed: 2017.

[133] "VMware inc. TrustPoint." http://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/products/trustpoint/vmware-trustpoint-solution-overview.pdf. Accessed: 2017.

[134] "Microsoft Advanced Threat Analytics." https://www.microsoft.com/en-us/cloud-platform/advanced-threat-analytics. Accessed: 2017.

[135] "Microsoft Cloud App Security." https://www.microsoft.com/en-us/cloud-platform/cloud-app-security. Accessed: 2017.

[136] D. Serrano, S. Bouchenak, Y. Kouki, F. A. de Oliveira Jr., T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens, "SLA Guarantees for Cloud Services," *Future Gener. Comput. Syst.*, vol. 54, pp. 233–246, Jan. 2016.

[137] "Open vSwitch." http://openvswitch.org/. Accessed: 2015.

[138] "Libvirt." https://libvirt.org/. Accessed: 2017.

[139] "Inotify- Monitoring Filesystem Events." http://man7.org/linux/man-pages/man7/inotify.7.html. Accessed: 2015.

[140] A. Giannakou, L. Rillling, J.-L. Pazat, F. Majorczyk, and C. Morin, "Towards Self Adaptable Security Monitoring in IaaS Clouds," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pp. 737–740, May 2015.

[141] "Emerging Threats Suricata Rules." https://rules.emergingthreats.net/open/suricata/rules/. Accessed: 2015.

[142] D. J. Barrett and R. E. Silverman, *SSH, The Secure Shell: The Definitive Guide*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2001.

[143] "LMbench - Tools for Performance Analysis." http://lmbench.sourceforge.net. Accessed: 2016.

[144] "Systat Utilities." http://sebastien.godard.pagesperso-orange.fr/. Accessed: 2017.

[145] "Metasploit Penetration Testing Software." https://www.metasploit.com/. Accessed: 2017.

[146] A. Giannakou, L. Rilling, J. L. Pazat, and C. Morin, "Al-safe: A secure self-adaptable application-level firewall for iaas clouds," in *2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 383–390, Dec 2016.

[147] "Nftables." http://netfilter.org/projects/nftables/. Accessed: 2016.

[148] "Volatility Memory Forensics." http://www.volatilityfoundation.org. Accessed: 2016.

[149] "Iperf." https://iperf.fr. Accessed: 2016.

[150] "Apache Web Server." http://www.apache.org. Accessed: 2016.

[151] "Apache HTTP server benchmarking tool." https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed: 2016.

[152] "MySQL ." https://www.mysql.com/. Accessed: 2017.

[153] "RabbitMQ ." https://www.rabbitmq.com/. Accessed: 2017.

[154] "Bro Network Security Monitor." https://www.bro.org. Accessed: 2015.

[155] "Qirinus." https://qirinus.com/index.php/en/. Accessed: 2017.

Annexe A

Résumé en français

A.1 Contexte

La virtualisation des serveurs permet une répartition à la demande de ressources informatiques (par exemple, CPU et RAM) selon le modèle ≪ paiement à l'usage ≫, un modèle économique où les clients ne sont facturés que pour le temps et la quantité des ressources utilisées. L'un des principaux modèles de cloud qui a attiré une attention particulière au cours des dernières années est le modèle *Infrastructure as a Service* (IaaS) où les ressources de calcul, de stockage et de réseau sont fournies aux clients sous la forme de machines virtuelles (VM) et de réseaux virtuels. Les organisations externalisent une partie de leurs systèmes d'information sur des infrastructures virtuelles (composées de VM et de réseaux virtuels) hébergées sur l'infrastructure physique du fournisseur de cloud. Les termes qui réglementent l'allocation des ressources sont déclarés dans un contrat signé par les clients et le fournisseur de cloud, appelé contrat de niveau de service (*Service Level Agreement* ou SLA). Les principaux avantages des clouds IaaS incluent : la flexibilité dans l'allocation des ressources, l'illusion d'une capacité illimitée de ressources informatiques et réseau et l'administration automatisée de systèmes d'information virtualisés complexes.

Bien que le passage au cloud puisse générer d'importants gains en termes de coûts et d'efficacité, la sécurité continue de rester l'une des principales préoccupations dans l'adoption du modèle de cloud. La cohabitation de plusieurs clients, l'une des caractéristiques clés d'une infrastructure de cloud, crée la possibilité que des machines virtuelles légitimes soient co-localisées avec des machines virtuelles contrôlées par des attaquants. Par conséquent, les attaques contre des infrastructures en cloud peuvent provenir de l'intérieur et de l'extérieur de l'environnement de cloud. Une attaque réussie pourrait permettre aux attaquants d'accéder et de manipuler les données hébergées par un cloud, y compris les informations d'identification légitimes du compte utilisateur, ou même d'obtenir un contrôle complet de l'infrastructure de cloud et de la transformer en une entité malveillante. Bien que les techniques de sécurité traditionnelles telles que le filtrage du trafic ou l'inspection du trafic puissent fournir un certain niveau de protection contre les attaquants, elles ne suffisent pas à contrer les menaces sophistiquées qui ciblent les infrastructures virtuelles. Afin de fournir une solution de sécurité pour les environnements en cloud, une architecture de sécurité autonome automatisée qui intègre des outils de sécurité et de supervision hétérogènes est requise.

A.2 Motivation

Dans un environnement de cloud IaaS typique, le fournisseur est responsable de la gestion et de la maintenance de l'infrastructure physique, alors que les clients ne sont responsables que de la gestion de leur propre système d'information virtualisé. Les clients peuvent prendre des décisions concernant le cycle de vie des VMs et déployer différents types d'applications sur les VMs fournies. Étant donné que les applications déployées peuvent avoir accès à des informations sensibles ou effectuer des opérations critiques, les clients s'occupent de superviser la sécurité de leur infrastructure virtualisée. Ces préoccupations peuvent s'exprimer sous la forme d'exigences relatives à la surpervision de sécurité, c'est-à-dire la surveillance d'actions de types spécifiques de menaces dans l'infrastructure virtualisée. Les solutions de supervision de sécurité pour les environnements en clouds sont généralement gérées par le fournisseur du cloud et sont constituées d'outils hétérogènes pour lesquels une configuration manuelle est requise. Afin de fournir des résultats de détection corrects, les solutions de surpervision doivent tenir compte du profil des applications déployées par le client ainsi que des exigences spécifiques de sécurité des clients.

Un environnement en cloud présente un comportement très dynamique avec des changements qui se produisent à différents niveaux de l'infrastructure de cloud. Malheureusement, ces changements affectent la capacité d'un système de supervision de sécurité du cloud à détecter avec succès les attaques et à préserver l'intégrité de l'infrastructure en cloud. Les solutions existantes de supervision de sécurité des clouds ne permettent pas de prendre en compte les changements et de prendre les décisions nécessaires concernant la reconfiguration des dispositifs de sécurité. En conséquence, de nouveaux points d'entrée pour les attaquants sont créés, ce qui peut entraîner une compromission de l'infrastructure entière du cloud. À notre connaissance, il n'existe toujours pas de système de supervision de sécurité capable d'adapter ses composants en fonction des différents changements qui se produisent dans un environnement de cloud.

L'objectif de cette thèse est de concevoir et mettre en œuvre un système de supervision de sécurité auto-adaptatif capable de réagir aux événements dynamiques qui se produisent dans une infrastructure en cloud et d'adapter ses composants afin de garantir un niveau adéquat de supervision de sécurité pour les infrastructures virtuelles des clients.

A.3 Objectifs

Après avoir présenté le contexte et la motivation de cette thèse, nous proposons maintenant un ensemble d'objectifs pour un système de supervision de sécurité auto-adaptatif.

A.3.1 Auto-adaptation

Un système de supervision de sécurité auto-adaptatif devrait pouvoir adapter ses composants en fonction de différents types d'événements dynamiques qui se produisent dans une infrastructure de cloud. Le système devrait considérer ces événements comme des sources d'adaptation et prendre en conséquence des mesures qui reconfigurent ses composants. Le processus d'adaptation peut modifier la configuration des dispositifs de supervision existants ou en créer d'autres. Le système peut décider de modifier les quantités de ressources informatiques disponibles pour un dispositif de supervision (ou un sous-ensemble de dispositifs de supervision) afin de maintenir un niveau de supervision adéquat. L'adaptation de la quantité de ressources informatiques devrait également être effectuée afin de libérer des ressources sous-utilisées. Le système devrait prendre des décisions d'adaptation afin

de garantir un équilibre entre sécurité, performance et coût à tout moment. Les actions d'adaptation peuvent affecter différents composants et le système devrait pouvoir effectuer ces actions en parallèle.

A.3.2 Personnalisation

Les exigences relatives aux clients et concernant les cas de surpervision spécifiques devraient être prises en compte dans un système de supervision de sécurité auto-adaptatif. Le système devrait être en mesure de garantir une supervision adéquate des types spécifiques de menaces demandés par le client. Une demande de supervision pourrait se référer à l'infrastructure virtuelle complète d'un client ou à un sous-ensemble spécifique de machines virtuelles. Le système devrait fournir le type de supervision demandé jusqu'à ce que la demande du client change ou que les machines virtuelles auxquelles le type de supervision est appliqué n'existent plus. En outre, le système devrait prendre en compte les seuils définis par le clients (par des SLA spécifiques) qui font référence à la qualité du service de supervision ou à la performance de types spécifiques de dispositifs de supervision.

A.3.3 Sécurité et correction

Le déploiement d'un système de supervision de sécurité auto-adaptatif ne devrait pas ajouter de nouvelles vulnérabilités dans l'infrastructure virtuelle supervisée ou dans l'infrastructure du fournisseur. Le processus d'adaptation et les entrées qu'il requiert ne devraient pas créer de nouveaux points d'entrée pour un attaquant. En outre, un système de supervision de sécurité auto-adaptatif devrait pouvoir garantir qu'un niveau de supervision adéquat soit maintenu tout au long du processus d'adaptation. Le processus d'adaptation ne devrait pas interférer avec la capacité du système à détecter correctement les menaces.

A.3.4 Minimisation des coûts

Le déploiement d'un système de supervision de sécurité auto-adaptatif ne devrait pas avoir d'impact significatif sur le compromis entre sécurité et coût pour les clients et le fournisseur. Du côté du client, un système de supervision de sécurité auto-adaptatif ne devrait pas influer de manière significative sur les performances des applications hébergées dans l'infrastructure virtuelle, quel que soit le profil de l'application (utilisant beaucoup les CPUs ou beaucoup le réseau). Du côté du fournisseur, la capacité de générer des profits en louant ses ressources informatiques ne devrait pas être affectée de manière significative par le système. Le déploiement d'un tel système ne devrait pas imposer de pénalité importante dans les opérations normales du cloud (par exemple, migration de VM, création, etc.). En outre, la proportion des ressources informatiques dédiées aux composants du système auto-adaptatif devrait refléter un accord entre les clients et le fournisseur pour la distribution des ressources informatiques.

A.4 Contributions

Afin d'atteindre les objectifs présentés dans la section précédente, nous concevons un système de supervision de sécurité auto-adaptatif capable de dépasser les limites des systèmes de supervision existants et de gérer les événements dynamiques qui se produisent dans une infrastructure en cloud. Dans cette thèse, nous détaillons comment nous avons conçu, mis en œuvre et évalué nos contributions : un système générique de supervision de

sécurité auto-adaptatif et deux instanciations avec des systèmes de détection d'intrusion et des pare-feu.

A.4.1 Un système de supervision de sécurité auto-adaptatif

Notre première contribution est la conception d'un système de supervision de sécurité auto-adaptatif capable de modifier la configuration de ses composants et d'adapter la quantité de ressources informatiques disponibles selon le type d'événement dynamique qui se produit dans une infrastructure de cloud. Notre système réalise l'adaptation automatique et la personnalisation en fonction des clients tout en fournissant un niveau adéquat de supervision de la sécurité grâce au processus d'adaptation. Notre système comprend les composants suivants : le gestionnaire d'adaptation (ou *Adaptation Manager*), les sondes de supervision d'infrastructure (ou *Infrastructure Monitoring Probes*), la base de données de dépendances (ou *Dependency Database*), l'API côté client et enfin les dispositifs de sécurité. Le gestionnaire d'adaptation est au cœur de notre système et est chargé de prendre les décisions d'adaptation lorsque des événements dynamiques se produisent. Les sondes de supervision d'infrastructure sont capables de détecter des événements dynamiques liés à la topologie et de transmettre toutes les informations nécessaires au gestionnaire d'adaptation. La base de données de dépendances est utilisée afin de stocker des informations importantes concernant les dispositifs de sécurité, tandis que, via l'API côté client, les clients peuvent exprimer leurs propres exigences de supervision de sécurité. Enfin, les dispositifs de sécurité assurent différentes fonctionnalités de supervision de sécurité.

A.4.2 SAIDS

Notre deuxième contribution constitue la première instanciation de notre système et est axée sur les systèmes de détection d'intrusion en réseau (NIDS). Les NIDS sont des éléments clés d'une infrastructure de supervision de sécurité. SAIDS atteint les objectifs au cœur de notre système tout en fournissant une solution passant à l'échelle pour répondre aux nécessités d'adaptation parallèles. Notre solution est capable de passer à l'échelle en fonction de la charge du trafic surveillé et de la taille de l'infrastructure virtuelle. Les composants principaux de SAIDS sont : le pilote d'adaptation maître (ou *Master Adaptation Driver*), le travailleur d'adaptation (ou *Adaptation Worker*) et les capteurs locaux de détection d'intrusion (LIDS). Le *Master Adaptation Driver* est chargé de la traduction des arguments d'adaptation en des paramètres de configuration pour les LIDSs alors que le travailleur d'adaptation est chargé d'effectuer la reconfiguration en tant que telle des LIDSs. Les capteurs locaux de détection d'intrusion sont les dispositifs de sécurité qui effectuent la détection réelle des événements de sécurité. SAIDS maintient un niveau de détection adéquat tout en minimisant le coût en termes de consommation de ressources et de performance des applications déployées. Nous avons évalué la capacité de SAIDS à obtenir un compromis entre les performances, les coûts et la sécurité. Notre évaluation consiste en différents scénarios qui représentent des environnements de production. Les résultats obtenus démontrent que notre prototype passe à l'échelle et peut gérer plusieurs capteurs de détection d'intrusion réseau en parallèle. SAIDS impose des coûts additionnels négligeables pour les applications des clients ainsi que pour des opérations de cloud typiques telles que la migration de VM. En outre, nous avons prouvé que SAIDS maintient un niveau de détection adéquat au cours du processus d'adaptation.

A.4.3 AL-SAFE

Notre troisième contribution constitue la deuxième instanciation de notre système et est axée sur les pare-feu applicatifs. AL-SAFE utilise l'introspection de machine virtuelle afin de créer un pare-feu sécurisé qui fonctionne à l'extérieur de la machine virtuelle surveillée mais conserve la visibilité intra-VM. AL-SAFE suit une stratégie d'introspection périodique et permet au client de spécifier la période d'introspection. Les jeux de règles appliqués par le pare-feu sont adaptés en fonction des événements dynamiques qui se produisent dans une infrastructure virtuelle. Les composants principaux d'AL-SAFE sont : l'agent d'extraction d'informations (ou *Information Extraction Agent*), le composant d'introspection de machine virtuelle (ou *Virtual Machine Introspection*), les générateurs de règles (ou *Rule Generators*) et deux pare-feu distincts. L'agent d'extraction d'informations est chargé d'identifier les processus autorisés à établir des connexions alors que le composant Virtual Machine Introspection effectue l'introspection en tant que telle de la VM supervisée. Les générateurs de règles sont utilisés pour produire les règles pour les deux pare-feux. Nous avons évalué la capacité d'AL-SAFE à proposer un compromis équilibré entre sécurité, performance et coût. Notre processus d'évaluation se compose de différents scénarios qui représentent des environnements de production. Les résultats obtenus démontrent que AL-SAFE est capable de bloquer toutes les connexions non autorisées et que les règles résultant du processus d'adaptation sont correctes et opérationnelles. Les coûts additionnels d'AL-SAFE pour les opérations typiques du cloud, comme la migration de VM, sont indépendants de l'intensité de l'activité de la VM, tandis que les coûts additionnels pour les applications des clients dépendent de la période d'introspection et du profil d'application (réseau ou calcul).

A.5 Perspectives

Nous avons identifié plusieurs axes de recherche pour les travaux futurs. Nous les organisons sur des objectifs à court, moyen et long terme.

A.5.1 Perspectives à court terme

Nos objectifs à court terme se concentrent sur les améliorations de conception et de mise en œuvre des versions actuelles de nos prototypes ainsi que sur la mise en œuvre de deux des composants de notre système que nous n'avons pas eu le temps de mettre en œuvre. Dans SAIDS, nous souhaitons ajouter des découvertes automatiques de service afin que les règles de détection liées aux services exécutés, et appliquées dans les LIDSs affectés, soient automatiquement adaptées. Le mécanisme d'introspection d'AL-SAFE pourrait être utilisé comme outil de découverte automatique des services. Dans AL-SAFE, nous souhaitons remplacer le modèle d'introspection périodique par un modèle d'introspection déclenchée par des évènements, de sorte que les coûts additionnels dans les applications des clients soient réduits. Enfin, nous souhaitons mettre en œuvre deux autres composants de notre système, la base de données de dépendances et l'API côté client.

A.5.2 Perspectives à moyen terme

Nos objectifs à mi-parcours visent à aborder des problèmes plus complexes qui sont intrinsèques aux environnements de cloud, comme la cohabition des clients. La version actuelle de notre système de supervision de sécurité ne traite pas des problèmes qui se posent dans les environnements multi-clients. Afin de permettre la supervision de la sécurité pour

différents clients, nous devons considérer le partage des dispositifs de supervision entre les clients. Le partage de dispositifs entre les clients peut également être perçu comme un aspect supplémentaire de la réduction des coûts. Nous souhaitons étudier les changements nécessaires tant dans SAIDS que dans AL-SAFE afin d'atteindre cet objectif. En outre, nous souhaitons inclure d'autres types de LIDS comme les systèmes de détection d'intrusion hôte et les analyseurs de réseau dans le prototype SAIDS. Les autres perspectives de recherche à moyen terme incluent la combinaison de la supervision de la sécurité des clients et du fournisseur ainsi que l'intégration de SAIDS dans un système à grande échelle, grâce à une collaboration avec la startup Qirinus.

A.5.3 Perspectives à long terme

Dans une perspective à long terme, nous nous intéressons à la conception d'un système de supervision de sécurité auto-adaptatif entièrement autonome. Un système de supervision entièrement autonome devrait pouvoir réagir aux événements de sécurité et prendre des mesures en conséquence afin d'isoler les machines virtuelles potentiellement infectées et empêcher les attaquants de prendre le contrôle de l'infrastructure virtuelle. La réaction repose essentiellement sur la capacité du système à traduire les résultats de la supervision de sécurité (par exemple, les alertes des systèmes de détection d'intrusion) en des décisions d'adaptation qui reconfigurent des dispositifs de supervision. Dans le contexte de cette thèse, une telle capacité est liée à l'inclusion d'événements de sécurité dans l'ensemble des sources d'adaptation possibles. Actuellement, notre système de supervision de sécurité auto-adaptatif prend en charge l'adaptation des dispositifs de sécurité en fonction de trois types d'événements dynamiques : ceux liés à la topologie, aux services déployés, et à la charge de travail en analyse. Les événements de sécurité (c'est-à-dire les attaques) en tant que source d'adaptation potentielle n'ont pas été pris en compte.

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Supervision de sécurité auto-adaptative dans les clouds IaaS

Nom Prénom de l'auteur : GIANNAKOU ANNA

Membres du jury :
- Monsieur PAZAT Jean-Louis
- Madame BOUCHENAK Sara
- Monsieur DEBAR Hervé
- Madame MORIN Christine
- Monsieur TOTEL Eric
- Monsieur RILLING Louis
- Monsieur SCOTT Stephen
- Monsieur CARON Eddy

Président du jury : TOTEL ERIC

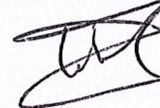
Date de la soutenance : 06 Juillet 2017

Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état
~~Thèse pouvant être reproduite après corrections suggérées~~

Fait à Rennes, le 06 Juillet 2017

Signature du président de jury



Le Directeur,

M'hamed DRISSI

Résumé

Les principales caractéristiques des clouds d'infrastructure (IaaS), comme l'élasticité instantanée et la mise à disposition automatique de ressources virtuelles, rendent ces clouds très dynamiques. Cette nature dynamique se traduit par de fréquents changements aux différents niveaux de l'infrastructure virtuelle. Étant données la criticité et parfois la confidentialité des informations traitées dans les infrastructures virtuelles des clients, la supervision de sécurité est une préoccupation importante pour les clients comme pour le fournisseur de cloud. Malheureusement, les changements dynamiques altèrent la capacité du système de supervision de sécurité à détecter avec succès les attaques ciblant les infrastructures virtuelles. Dans cette thèse, nous avons conçu un système de supervision de sécurité auto-adaptatif pour les clouds IaaS. Ce système est conçu pour adapter ses composants en fonction des différents changements pouvant se produire dans une infrastructure de cloud. Notre système est instancié sous deux formes ciblant des équipements de sécurité différents : SAIDS, un système de détection d'intrusion réseau qui passe à l'échelle, et AL-SAFE, un firewall applicatif fondé sur l'introspection. Nous avons évalué notre prototype sous l'angle de la performance, du coût, et de la sécurité pour les clients comme pour le fournisseur. Nos résultats montrent que notre prototype impose un coût additionnel tolérable tout en fournissant une bonne qualité de détection.

Abstract

Rapid elasticity and automatic provisioning of virtual resources are some of the main characteristics of IaaS clouds. The dynamic nature of IaaS clouds is translated to frequent changes that refer to different levels of the virtual infrastructure. Due to the critical and sometimes private information hosted in tenant virtual infrastructures, security monitoring is of great concern for both tenants and the provider. Unfortunately, the dynamic changes affect the ability of a security monitoring framework to successfully detect attacks that target cloud-hosted virtual infrastructures. In this thesis we have designed a self-adaptable security monitoring framework for IaaS cloud environments that is designed to adapt its components based on different changes that occur in a virtual infrastructure. Our framework has two instantiations focused on different security devices: SAIDS, a scalable network intrusion detection system, and AL-SAFE, an introspection-based application-level firewall. We have evaluated our prototype focusing on performance, cost and security for both tenants and the provider. Our results demonstrate that our prototype imposes a tolerable overhead while providing accurate detection results.

INSA RENNES

N° d'ordre : 17ISAR 19 / D17 - 19

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coëmes • CS 70839 • F-35708 Rennes Cedex 7

Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96