User-centric security monitoring in cloud environments

Amir Teshome Wonjiga

To cite this version:

Amir Teshome Wonjiga. User-centric security monitoring in cloud environments. Cryptography and Security [cs.CR]. Université de Rennes, 2019. English. NNT : 2019REN1S080 . tel-02570591

HAL Id: tel-02570591
https://theses.hal.science/tel-02570591

Submitted on 12 May 2020

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.

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

UNIVERSITÉ DE RENNES 1

THÈSE DE DOCTORAT DE

L'UNIVERSITE DE RENNES 1
COMUE UNIVERSITE BRETAGNE LOIRE

Ecole Doctorale N°601
Mathèmatique et Sciences et Technologies
de l'Information et de la Communication
Spécialité : *Informatique*

Par

« Amir Teshome WONJIGA »

« User-Centric Security Monitoring in Cloud Environments »

Thèse présentée et soutenue à Rennes , le 3 Juin 2019
Unité de recherche : Inria Rennes – Bretagne Atlantique
Thèse N° :

Rapporteurs avant soutenance :

Hervé DEBAR, Professeur, Télécom SudParis
Pierre SENS, Professeur, Sorbonne Université

Composition du jury :

Président : Ludovic MÉ, Professeur, Centrale-Supélec
Examinateurs : Eric ALATA, Maître de conférences, INSA Toulouse
Hervé DEBAR, Professeur, Télécom SudParis
Marc LACOSTE, Chercheur, Orange Labs
Thomas LEDOUX, Maître de conférences HDR, IMT Nantes Atlantique
Sean PEISERT, Staff scientist, Lawrence Berkeley National Laboratory, Berkeley, USA
Pierre SENS, Professeur, Sorbonne Université

Dir. de thèse : Christine MORIN, Directrice de recherche, Inria Rennes – Bretagne Atlantique
Co-dir. de thèse : Louis RILLING, Ingénieur-Chercheur, DGA-MI

"*His command is only when He intends a thing that He says to it,"Be," and it is.*"

Quran 36:82

Abstract

One of the major factors which lead to the success of cloud computing is its economic model. Cloud computing presents a win-win economic model for both service providers and tenants. For tenants, cloud computing offers a pay-per-use model where tenants are expected to pay only for the resources they use. For providers, it allows profit maximization by dynamically configuring and allocating resources that are requested by tenants. The downside of cloud emerges from its own characteristics. When migrating to the cloud the tenant loses full control of the physical infrastructure and the provider is responsible for managing the infrastructure including its security. As this forces tenants to rely on service providers for the security of their service, it creates a trust issue. Service providers acknowledge the trust issue and provide a guarantee through an agreement called Service Level Agreement (SLA). However, almost all existing SLAs do not guarantee the security aspect of tenants' hosted services. In this context, we propose several contributions for including security monitoring features in cloud SLAs. First, to define the agreement we design an extension for a formal SLA language to include security monitoring features. To customize the SLA for each tenant, the language is capable of taking users requirements in the form of vulnerabilities. Second, to prepare the service level objectives (SLOs) in the SLA we proposed an efficient performance estimation method for a security monitoring device. The metrics used in the SLO characterize the performance of a monitoring device by taking all the necessary parameters into account. Third, in order to check the satisfaction of an SLO we propose an in situ evaluation method of the security monitoring configuration. This method requires cooperation between tenants and providers. Finally, in order to remove the need for cooperation, we present an SLO verification method based on a secure trusted and distributed ledger (blockchain). The performance estimation method and the two SLO verification methods were implemented and experimentally evaluated.

Acknowledgement

Undertaking this PhD has been a great experience for me and it would not have been possible to reach at this point without the support and guidance that I received from many people.

Firstly, I would like to express my sincere gratitude to my advisors Louis Rilling and Christine Morin for their continuous support of my PhD study and related research, for their patience, motivation, and immense knowledge. Their guidance helped me in all the time of research including writing of this thesis. I could not have imagined having a better advisor and mentor for my study.

Many thanks also to Daniel Hirschkoff and Eddy Caron who supported me a lot in my master study and helped me during our many discussions in Lyon that I should pursue my doctoral degree. They greatly helped to create the connection with Myriads team where I studied for my PhD.

I thank my fellow labmates in for the stimulating discussions, for all the fun we have had, for their friendship and the warmth they extended to me during my time in the lab. My thanks also go out to the support I received from the collaborative work I undertook with the team at LBNL. I am especially grateful to Sean Peisert for believing in me and for all the support I received through my internship.

A special thanks to my family. Words cannot express how grateful I am to my mother, father and all family members for all the sacrifices that you have made on my behalf. Your prayer for me was what sustained me thus far.

This PhD study would not have been possible without the cooperation and support from lots of friends I met throughout my stay in France. I'm deeply thankful and blessed for the special, unique and amazing people in my life who support me, uplift me and comfort me throughout my studies.

Table of Contents

1 Introduction 1
1.1 Computing as Utility 1
1.1.1 Why Cloud Computing ? 1
1.1.2 User-Centric Service 2
1.1.3 User-Centric Cloud Security Monitoring 2
1.2 Objectives 3
1.3 Contributions 4
1.4 Outline of this Thesis 5

2 State Of the Art 7
2.1 Cloud Computing 7
2.1.1 Characteristics 7
2.1.2 Deployment Models 8
2.1.3 Service Models 8
2.2 Virtualization 9
2.2.1 Machine Virtualization 9
2.2.2 Network Virtualization 11
2.2.3 OpenStack, an Example IaaS Platform 15
2.3 Information System Security 16
2.3.1 Security Policy 17
2.3.2 Security Properties 17
2.3.3 Security Mechanisms 18
2.3.4 Security Threats in the Cloud 19
2.4 Security Monitoring 20
2.4.1 What is Security Monitoring 20
2.4.2 Security Monitoring Devices 21
2.4.3 Intrusion Detection System (IDSs) 23
2.4.4 Measuring the Performance of Security Monitoring Devices 25
2.4.5 Security Monitoring in Clouds 32
2.5 Cloud Service Level Agreement (SLA) 35
2.5.1 SLA and its Characteristics 35
2.5.2 SLA Components 35
2.5.3 Phases of an SLA Life-Cycle 36
2.5.4 SLA Standards and Languages 38
2.5.5 Security SLAs 41
2.6 Summary 43

3 User-Centric Security Monitoring 45
3.1 Introduction 45
3.2 The Need for User-Centric Security Monitoring in Clouds 46
3.2.1 Cooperation Between Providers and Clients 46
3.2.2 Objectives 47
3.2.3 Challenges 47
3.3 System Model: Outsourced Information System Resource 48
3.4 Threat Model 49
3.5 Including Users in the Security Monitoring Process Through an SLA 49
3.5.1 SLA-Based Users Participation 49

3.5.2 Defining Security Monitoring SLAs 50
3.5.3 Enforcing SLA Terms 51
3.5.4 Verification of SLO Fulfillment 51
3.6 Summary 52

4 Defining Security Monitoring SLAs 53
4.1 Introduction 53
4.2 Objectives and Problem Description 54
4.2.1 Objectives 54
4.2.2 Problem Description 54
4.2.3 Contributions 55
4.3 Components of Security Monitoring SLAs 55
4.3.1 KPI for Security Monitoring SLO 56
4.3.2 Class of SLOs Considered 56
4.4 Extended Cloud Service Level Agreement (ECSLA) 57
4.4.1 CSLA 57
4.4.2 ECSLA 58
4.4.3 Example Security Monitoring SLA 58
4.5 Including Unknown Base Rate Values in SLO 62
4.5.1 Assumptions and Challenges 62
4.5.2 Metrics Estimation Method 63
4.6 NIDS Performance with a Large Number of Vulnerabilities 64
4.6.1 Modeling Rule Interference and its Effect on NIDS Performance 64
4.6.2 Rule Interference in NIDS 65
4.6.3 Building a Knowledge Base by Clustering Vulnerabilities 68
4.7 Evaluation 69
4.7.1 Experimental Setup 69
4.7.2 Collecting Data Points and Generating an Estimation Model 70
4.7.3 Clustering NIDS Rules 72
4.7.4 Discussion 73
4.8 Summary 75

5 Monitoring an SLA Describing the Performance of an NIDS 77
5.1 Introduction 77
5.1.1 Simplified SLA Enforcement Mechanisms for NIDS 78
5.2 Problem Description and Challenges of SLA Verification 78
5.2.1 General Problems 78
5.2.2 Specific Problems for Security Monitoring SLAs 79
5.3 Threat Model 79
5.4 SLO Verification Method 80
5.4.1 Unit of Measure 80
5.4.2 Architecture 80
5.4.3 Attack Injection Algorithm 82
5.5 KPI Computation 84
5.6 Implementation 86
5.7 Evaluation 86
5.7.1 Experimental Setup 87
5.7.2 Performance Impact 87
5.7.3 Correctness, Usefulness and Security Analysis 88
5.8 Shortcomings of the Proposed SLO Verification Method 91
5.9 Summary 92

6 Monitoring an SLA on Data Integrity 93
6.1 Introduction 93
6.1.1 Dependency Between Tenants and Providers 93
6.1.2 Trusted Component in the SLA Verification Process 93
6.1.3 Considered SLA 94
6.2 SLA Life-cycle for Monitoring Data Integrity 95
6.3 Problem Description 95

6.4 Background and Related Works . . . 96
6.4.1 Blockchain . . . 96
6.4.2 Related Works . . . 99
6.5 Threat Model . . . 101
6.6 Data Integrity Checking Process . . . 101
6.6.1 Architecture . . . 102
6.6.2 Integrity Checking Process . . . 103
6.7 Implementation . . . 106
6.8 Evaluation . . . 106
6.8.1 Experimental Setup . . . 106
6.8.2 Performance Evaluation . . . 107
6.8.3 Analysis and Discussion . . . 108
6.9 Summary . . . 109

7 Conclusion and Future Work **111**
7.1 Conclusion . . . 111
7.1.1 Security Monitoring SLA Language . . . 111
7.1.2 Preparing SLA Templates . . . 112
7.1.3 SLA Verification . . . 112
7.2 Future Work . . . 113
7.2.1 Short Term . . . 114
7.2.2 Medium Term . . . 115
7.2.3 Long Term . . . 115

A Appendix **117**
A.1 XML schema of ECSLA . . . 117

Bibliography **119**

List of Figures

2.1 Responsibilities in different service models . . . 9
2.2 Virtualized and Non-Virtualized Infrastructure . . . 10
2.3 Software defined networking APIs . . . 14
2.4 Openstack services overview . . . 14
2.5 Openstack networking overview . . . 15
2.6 Security concepts and relationships . . . 16
2.7 Block representation of IDS . . . 25
2.8 Example ROC curve . . . 27
2.9 Venn diagram showing entropy, mutual information and realistic NIDS situation . . . 30
2.10 NIDS evaluation process as proposed in [180] . . . 31
2.11 GCCIDS architecture from [216] . . . 32
2.12 IDS management architecture [189] . . . 32
2.13 DISCUS architecture and DSL deployment as presented in [187] . . . 33
2.14 Virtual Machine Introspection (VMI) architecture [89] . . . 34
2.15 Phases of SLA life-cycle . . . 36
2.16 Structure of an SLA in different languages . . . 38
2.17 Fuzziness and Confidence ratio graphical representation [123] . . . 40
2.18 SPECS framework [199] . . . 42
2.19 SPECS enforcement process [199] . . . 42
2.20 Sec-rSLA concepts overview [114] . . . 42
2.21 LoM2HiS framework architecture [74] . . . 43

4.1 CSLA meta model [123] . . . 57
4.2 ECSLA model diagram . . . 59
4.3 Base rate (B) vs C_{ID} . . . 63
4.4 NIDS outputs . . . 65
4.5 Clustering NIDS rules . . . 68
4.6 Experimental setup . . . 70
4.7 Plot of B vs TPR and B vs FPR for B in $(0.06 - 0.1)$ . . . 71
4.8 Estimated and actual TPR values . . . 72
4.9 Plot of the estimated and actual TPR values . . . 72

5.1 Example attack running setup . . . 81
5.2 Architecture for metrics evaluation . . . 85
5.3 Verification time line . . . 88
5.4 Impact of verification on the mean response time for the four types of legitimate requests 89
5.5 Response time overhead as a result of verification, with $B = 10^{-2}$, the four types of legitimate request and increasing traffic injection rates . . . 89
5.6 The rate of attack packets (base rate) in the network throughout the experiment targeting a base rate value of 10^{-2} . . . 90

6.1 Simplified block chain representation . . . 96
6.2 Consortium blockchain network formed by tenants and a provider . . . 101
6.3 Data integrity verification architecture . . . 102
6.4 Data integrity verification process, (A) setup phase and (B) verification phase . . . 103
6.5 Time required for setup phase operations (upload, hash and publish) . . . 107
6.6 Time overhead as a result of integrity verification . . . 109

List of Tables

2.1 Confusion matrix showing IDS outputs 26
2.2 CR, TPR and FPR as metrics derived from TP, FP, FN and TN 26
2.3 Amazon service commitments and service credits 37

4.1 Example of service list and attack types figuring in an SLO 56
4.2 NIDS metrics with probabilistic description 62
4.3 Interference Matrix (IM) 66
4.4 Multiple interference matrix from groups using different heuristics 69
4.5 TPR, FPR, TNR, FNR values of an NIDS from our experiment and calculated C_{ID} value for varying B 71
4.6 Rule classification based on application types 73
4.7 Rule classification based class type 74

5.1 Traffic injected using $B = 10^{-2}$ and parallelism degree of 3. 88

To my mother Gene Shifa and my father Teshome Wonjiga

Chapter 1

Introduction

Utility computing changes the business model of information technology and it is a primary reason for the success of the cloud computing paradigm. Cloud computing introduced a different kind of service model for information system services. Historically, obtaining computing resources required upfront capital investments. Utility computing reduces or removes these requirements allowing organizations to use computing resources more extensively at the time of need.

In this chapter, we introduce the concept of computing utilities and specifically the cloud computing paradigm. Furthermore, we introduce the concept of user-centric services and security monitoring in clouds. We also present the objectives and contribution of this thesis, which is to design and implement user-centric security monitoring services in the cloud. Finally, we conclude the chapter by describing the structure of this document.

1.1 Computing as Utility

In 1969, Leonard Kleinrock [120], a scientist who made a significant contribution on a mathematical theory of packet networks, wrote: "As of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of "computer utilities" which, like present electric and telephone utilities, will service individual homes and offices across the country".

Such a vision of computing as utility led to the development of different computing paradigms (e.g. grid computing, cluster computing ...). Cloud computing is one such paradigm evolved to give highly reliable, scalable, and autonomic service. It supports ubiquitous access with a modular computing service. Similar to other utility services, in a computing service users (also known as *tenants*) pay providers only when they access computing resources, which removes the need to buy and maintain a self-owned information system infrastructure. On the other hand, service providers make efficient use of resources to satisfy tenants demand and increase their profitability.

In the next sections, we describe the main factors that contribute to the success of cloud computing and security challenges faced by the cloud computing services. We also introduce a unique feature of cloud computing, user-centric nature of the service. Then we describe the need to have user-centric security services in the cloud which is the problem addressed in this thesis.

1.1.1 Why Cloud Computing ?

There are different factors which lead to the success of cloud computing. One of the major factors is the economics, and cloud computing presents a win-win economic model for both service providers and tenants. For tenants, cloud computing offers a pay-per-use model where tenants are expected to pay only for the resources they use. In comparison, investing in infrastructure to build one's own system (known as *on-premise* infrastructure) costs many folds more. For providers, it allows profit maximization by dynamically configuring and allocating resources that are requested by tenants. Immediate access to hardware resources, flexibility (easy to scale up and down), increased collaboration, convenience to meet government compliance requirements, and sustainability are few examples of other advantages of clouds.

However, cloud computing also has disadvantages compared to on-premise infrastructures. From a security perspective, the cloud is flawed by design. This is because in the cloud tenants cannot control the physical infrastructure. Tenants trust the provider to handle the underlying infrastructure responsibly and submit their data, application or system to deploy on the provided infrastructure. One can provide

a counterargument by stating the abilities of cloud providers to implement the most secure standards in their infrastructure. Assuming that most on-premise infrastructures will not keep up with all the updates and security patches the counterargument is acceptable. Nevertheless, it does not change the fact that deploying on an infrastructure which is owned by a different entity is problematic by itself. Mainly because it forces the tenant to trust the service provider.

In a 2018 survey made by Morphean [176], a physical security as a service provider, which collects information from 1500 IT decision makers across Europe (France, Germany, and UK) reported that around 89% of business decision makers are using cloud-based solutions in their company. The study described that contrary to the general assumption that the cloud is not secure, tenants are moving to cloud for security reasons. In the survey, nearly 27% of respondents said that the reason they moved applications to the cloud was to improve security, compared to 26% who were primarily motivated by cost implications and 22.7% who looked to improve collaborative working. This can be explained by the increase of trust in providers and maturity of tenants to understand the risks associated with the cloud. Overall, combining the advantages and the increased attention on security produces a weighty argument for using cloud-based technologies.

Service providers acknowledge the need for trust from tenants, and they provide a guarantee for their services. Providers and tenants sign an agreement regarding different aspects of the provided service in a contract called Service Level Agreement(SLA). The agreement, in addition to the provided service, describes a penalty for the cases of violation. Having such terms in SLA gives assurance to existing tenants and helps to attract more users into the cloud.

1.1.2 User-Centric Service

User-centric service could be introduced as a design pattern to favor adoption of the cloud and to make it trustworthy for tenants. User-centric design philosophy tries to optimize the product around how users can, want, or need to use the product, rather than forcing the users to change their behavior to accommodate the product. In such a system, users are the center of the design. Ideally, such a design is preferable for the cloud computing paradigm. In a world without cloud computing services, users will set up their own infrastructure according to their own needs and requirements. When such users migrate to the cloud, they expect to have some level of customization.

The rapid growth of needs for computing and diversity of applications make a user-centric design more attractive and increase the use of cloud computing. For any kind of application, the cloud provides computing resources specific to the requirements or needs of the application. The same resources can be dynamically reconfigured to be used for other applications. Dynamic reconfigurability allows the cloud to be configured according to the tenants' demand, thus helps to provide user-centric services.

From a provider perspective, user-centric service is both a challenge and an opportunity. Since multiple tenants, most probably with different computing service needs, reside under a single service provider, the provider aims to satisfy as many of its customers needs as possible. On the other hand, each customization helps a provider to maximize profit, i.e. the more a service is customized for a tenant, the more profit it brings.

The cloud continues to be more user-centric, and the resources offered as a service continue to become more specific. The market branched down from selling a fully operational virtualized machine towards selling specific resources (e.g. memory and I/O resources for a few seconds ...) and from fully hosted software service towards specific function, action, or piece of business logic (e.g. Function as a Service). Such functions are expected to start within milliseconds, process an individual request, and then the process ends. Such fine-graining of services allows tenants to build more complex systems according to their needs and requirements.

Virtualization is the technology behind cloud computing. Virtualization is a technique to create a virtual version of a device or resource, such as a server, storage device, network or even an operating system where the framework divides the resource into one or more execution environments. This notion of virtual instance helps the provider to manage the services provided to the tenants.

1.1.3 User-Centric Cloud Security Monitoring

Similar to the functional service needs in the cloud, tenants also have specific security monitoring requirements. Tenants consider different threat models and need to be monitored against different vulnerabilities. Even tenants with the same service needs may consider different security monitoring requirements.

For example, two tenants hosting the same software in the cloud may be interested in different vulnerabilities that exist in the software.

Although the functional service in the cloud is becoming more specific to tenants, security remained mostly generic. There are a few developments regarding user-centric security in the cloud. Mainly the types of security services addressing specific security threats (e.g. services addressing denial of service (DOS)). However, addressing a specific security property depending on the needs and requirements of a tenant remains as a challenge.

Security monitoring is the process of collecting and analyzing indicators of potential security threats, then triaging these threats by appropriate action. It is clear that organizations of all sizes must take steps to secure their data and systems in the ever-growing threat landscape. Security threats range from malware, hackers, vulnerable systems (software or hardware), third party service providers to intentional or unintentional actions from employees. Given the omnipresence and unavoidable nature of security risks, quick response time is essential to maintaining system security. Therefore, automated, continuous security monitoring is the solution to fast threat detection and response.

We can observe that security monitoring tasks are tightly coupled with the functionality of a system and the needs or requirements of the tenants. Thus, service providers need to adapt the security monitoring tasks for each tenant as much as possible. Additionally, like other services, providers need to offer a guarantee for the security monitoring aspect of the system. In this regard, we have yet to see more improvements.

To better understand the usefulness of security guarantees in SLA, let us describe the concern of information system users about the cloud. In the same study described above [176], two thirds (66%) of all the respondents said that they consider security as an important factor when evaluating cloud solutions. This is more than the subscription costs (51%) and speed of deployment (46%). The survey also reported that security is the biggest concern (45%) for businesses that did not migrate to the cloud followed by the migration cost (34%) and internal policy (32%).

The lack of security guarantee in general and the lack of security monitoring terms in SLA specifically in the cloud is a bottleneck for its adoption, especially for enterprises and cautious consumers. It creates a paradox to take away the physical infrastructure out of users control, motivating them to host their service in providers infrastructure and not assuring the security of their system.

In this thesis, we present the work we have done to address the issues related to user-centric security monitoring SLAs in clouds. To our knowledge currently existing cloud SLAs do not provide a guarantee regarding the security monitoring aspect of the information system. Our goal is to include security monitoring terms into cloud SLAs. This requires to supplement each phase in the cloud service life-cycle, starting before the actual service deployment until the final stage of the service. More detailed objectives are described in the next section.

1.2 Objectives

In the previous sections, we have presented how a user-centric design assisted and continues to assist in the adoption of clouds. We also presented the need for security monitoring tasks to follow the same design philosophy in order to improve the trust in cloud services and increase the success of clouds. In general, the goal of our work is to include security monitoring terms into cloud SLAs.

We achieve this by including terms into SLAs that guarantee the performance of security monitoring devices. The SLA defines the performance of a security monitoring device, where the device is configured according to the tenants' requirement. Then, the requested infrastructure will be configured towards achieving the objectives defined in the agreement. Moreover, participants can verify the infrastructure for the satisfaction of the expected level of performance. In cases when the infrastructure does not perform as expected, penalties will be applied as described in the SLA. Following this, we define the objectives of user-centric security monitoring SLAs.

- Like any other types of SLAs, to define security monitoring SLAs, we need an SLA language. The purpose of such language is to facilitate and standardize the communication between tenants and providers. The language should allow describing SLA components:

    1. It should allow describing cloud security monitoring services. In this thesis, a security monitoring service is described as the performance of a device which implements the monitoring task.

2. As we are aiming to have a user-centric security monitoring SLA, the language should allow tenants to describe their security monitoring needs.

In addition, the language needs to be semantically balanced between tenants and providers. i.e. it should not be very low level, in order to be understandable by regular cloud tenants, and it should not be very high level in order to automate the implementation as much as possible.

- In order to guarantee the performance of a security monitoring device, we need metrics. Providing metrics describing the performance of a security monitoring devices requires taking different parameters into account. These include both internal and external factors. For example, internal factors like configurations of the monitoring device, which is related to the tenant's requirement and external factors like operational environment (the network throughput, the rate of attacks ...) of the monitoring device. Two devices with the same internal configuration will not behave the same way when operating in a different external environment. For example, it is proven that the rate of occurrence of an attack greatly affects the performance of Network Intrusion Detection Systems (NIDS) [21].
- The metrics used in SLA terms should be relevant for tenants. In our SLA definition, the terms describe the performance of security monitoring devices. Specifically, a metric which describes the performance of a security monitoring device should take relevant parameters into account. The SLA should define how to measure, and compute the expected metric and the metric should be computable by all participants in the agreement.
- When providing security monitoring services, providers need to know their performance before advertising it for potential tenants. That means, providers need to test their monitoring capability beforehand. However, since there are tens of thousands or more security vulnerabilities and each tenant is interested in different subsets (perhaps in very few) of the vulnerabilities, there should be an efficient and practical method for the service providers to conduct the test. Such a method should allow providers to estimate their performance capability with an acceptable level of preparation task.
- The terms defined in the agreement should be verifiable by all participants. Verification or checking the satisfaction of objectives is performed after a tenant and a provider agreed on some SLA terms and the infrastructure is configured accordingly. For this purpose, we need to have a transparent verification mechanism, i.e. a verification method which allows both tenants and providers to perform the task. In addition, any party should be able to perform verification without relying on the other party. Since verification is allowed to reduce the amount of trust on the other party, the verification process should require as less trust as possible from the other party.

1.3 Contributions

In this section, we present our contributions. We proposed solutions in order to achieve the objectives described in the previous section. To address the problem of including security monitoring terms into SLA, we proposed solutions based on the SLA life-cycle. We designed and implemented the tasks required for the SLA definition and SLA verification phases. We also presented a mechanism to reduce the dependency between tenants and providers using a secure, trusted and distributed ledger. Our main contributions are:

- We designed extensions to an existing SLA language called Cloud SLA (CSLA). Our extension, called Extended CSLA (ECSLA), allows tenants to describe their security monitoring requirements in terms of vulnerabilities. Additionally, ECSLA correlates the user's requirements with other services. That is, in ECSLA a security monitoring service is described as a relation between user requirement as vulnerability, a software where the vulnerability exists and an infrastructure where the software is running.
- To offer security monitoring SLAs, providers need to measure the performance of their security monitoring capability before starting to negotiate with tenants. In the presence of tens of thousands of vulnerabilities and diversified tenants, service providers need to perform a large number of evaluations just to estimate their performance. In order to assist service providers in the preparation of SLA templates, we proposed an efficient performance estimation method which reduces the

required number of evaluations. The proposed solution introduces two new ideas; first, we describe a knowledge base building method which uses clustering to categorize a bunch of vulnerabilities together in groups using some heuristics, and second, we propose a model to quantify the interference between operations of monitoring vulnerabilities. Using these two methods, we can estimate the performance of a monitoring device with a fewer number of evaluations compared to the naive approach.

- The metrics used in our SLA terms consider the operational environment of the security monitoring devices. Considering such a parameter requires reconciling two facts. First, the parameters representing operation environments are non-deterministic (i.e. cannot be measured before the expected event happens). For example, if a metric takes the rate of occurrence of attacks (the base rate) in its formula, we cannot estimate its value before the occurrence of an actual attack. On the other hand at the time of SLA definition, it is required to specify the performance of a security monitoring device. To reconcile these two issues, we proposed an estimation mechanism where the performance of a monitoring device is measured using known parameters, and the result is used to model its performance and estimate for unknown values of that parameter. An SLA definition will contain the model then it can be used whenever the measurement is performed.

- We propose an in situ evaluation method for security monitoring configurations. Our method can evaluate the performance of a security monitoring setup in a production environment. The method uses an attack injection technique, but injected attacks will not affect the virtual machines used in the production environment. The method can be used by either of the parties. It also allows computing the required metric. However, it requires cooperation between tenants and providers.

- In order to have an end-to-end security monitoring service, tenants require information like the output of the monitoring devices and providers require knowledge about the services running inside the tenants' environments. As a result, there is a need for cooperation from both sides, which creates a dependency between tenants and providers. In order to remove this dependency while performing verification, we proposed a verification method which uses a logical secure component. The use of a logical secure component is illustrated in an SLA addressing data integrity in clouds rather than security monitoring devices. The method uses a secure trusted and distributed ledger (blockchain) to store evidence of data integrity. The method allows checking data integrity without relying on the other party. If there is any conflict between tenants and providers, the evidence can be used to resolve the situation.

1.4 Outline of this Thesis

In this section, we describe the organization of the remainder of this document.

In Chapter 2 we describe the state of the art around cloud, security in cloud and Service Level Agreements (SLAs). The chapter is organized into five main sections. While describing the current state of the art, we show the limitations from the perspective of user-centric security monitoring service. The first two sections provide a brief description of cloud computing and virtualization technologies. Then we present information system security in general and a detailed description of security threats in the cloud. Following that, we explain security monitoring with various types of tools to perform monitoring activities. We describe existing metrics and mechanisms that are used to measure security monitoring devices. Cloud security monitoring is also addressed in this section. The final section of the chapter features cloud SLAs and related issues like their characteristics, components, etc. It also presents SLAs that are used to address cloud security.

Chapter 3 describes how to achieve a user-centric security monitoring service and the challenges associated with it. It describes how we correlate user-centric design and cloud SLAs. It also presents how we decompose the problem of including security monitoring terms in cloud SLAs. To address the problem, we proposed a three-step incremental strategy based on the SLA life-cycle. The next chapters describe our contribution on these phases of security monitoring SLAs in clouds, with the exception of the enforcement phase. In Chapter 7 we present the remaining phase as future work.

Chapter 4 presents our security monitoring SLAs definition mechanism. It starts by describing the problems addressed regarding user-centric security monitoring SLA definition. Our SLA definition guarantees the performance of NIDS in clouds, while the NIDS is configured based on the tenant's requirements. An explanation for our extension of an existing SLA language is also presented. Following user-centric design, the language takes tenants requirements in terms of security vulnerabilities. It

describes the performance of Network Intrusion Detection System (NIDS) using a metric which takes the operational environment of an NIDS into account (specifically, the base rate). It also presents our metric estimation model which uses known values of base rate to prepare a model and estimate performance of NIDS for previously unknown base rate values. Our proposed method for service providers to efficiently estimate their security monitoring capabilities over a large number of vulnerabilities is also presented in this chapter. Finally, the chapter presents our prototype implementation and experimental evaluations to show the feasibility of our approach for a metrics estimation method and on the solution for large numbers of vulnerabilities to estimate the monitoring performance.

In Chapter 5 the third phase of the SLA life-cycle is presented. We investigate verification for SLAs describing the performance of NIDSs. The chapter starts by explaining a simplified SLA enforcement mechanism and identifying the challenges to achieve a transparent verification process. Then it goes to describe the proposed verification method by explaining the architecture and algorithms used for attack injection. The metrics computation method, which is executed after the attack campaign, is also described. Experimental evaluation to measure the overhead of the proposed method and the results are reported before finally describing shortcomings of the proposed method.

Chapter 6 shows an example of using secure components in the SLA life-cycle process. The chapter describes a verification process based on a secure, distributed ledger for SLAs describing data integrity in clouds. It explains the main problem in the verification method presented in Chapter 5 and argues for the need to have a fully transparent verification process. The chapter also presents the life-cycle of a data integrity SLA, provides background on blockchains and related works on both data integrity and application of blockchains for data integrity. Then it describes the integrity checking protocol and finalizes with the implementation, experiments and results measuring the overhead introduced as a result of the verification process.

Chapter 7 concludes the thesis by describing the contributions and contrasting them with the objectives which are defined in Section 1.2. It goes in detail describing the ideas, experiments, and results found in our work. It also provides a future work to propose an improvement, to fill the remaining works and an ambition of fully automated user-centric security monitoring services. The future works are categorized into short, middle and long term by the required amount of time needed to achieve each task.

Chapter 2

State Of the Art

In this chapter, we present the state of the art which serves as a background for our work. The chapter is organized into five sections. First, we introduce cloud computing and virtualization. Next, we describe information system security and security monitoring. Finally, Service Level Agreements (SLAs) and their relation with security monitoring in clouds are presented.

2.1 Cloud Computing

Cloud computing evolved as one of the major computing utility as a service over the Internet. The description presented by the National Institute of Standards and Technology (NIST) [143] incorporates most of the commonly accepted characteristics of cloud computing. NIST defines cloud computing as follows:

Definition 2.1. 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.

Service Providers (*Providers* in short) are entities providing such service and *tenants* refer users of the service. The main characteristics of cloud computing are *On-demand self-service, Broad network access, Resource pooling, Elasticity and Measured Service.*

2.1.1 Characteristics

- *On-demand self-service:* Provisioning a computing service does not necessarily require human interaction with the service provider. A dedicated online Control Panel (CP) or an Application Programing Interface (API) helps users to attain the service.
- *Broad network access:* Cloud services are provided through a network (Internet or local network). Any device (e.g. phone, smart device, computer ...) having access to the network can access the service.
- *Resource pooling:* A service provider puts together its available resources to provide services for more than one tenant with different needs at the same time (multi-tenancy). Tenants needs could be achieved dynamically, and a tenant has no knowledge over the exact location of the provided resource. As part of requirements for a service, tenants could specify location at a high level of abstraction (e.g., country, state, or data center).
- *Elasticity (Dynamic Scaling):* The size of resources can dynamically and automatically scale up and down according to the current workload. This is greatly attractive for tenants as it reduces their costs compared to investing on a self owned (on-premises) infrastructure for a given service (i.e. tenants pay only for what they use).
- *Measured Service:* Resource usage is measured in order to control and optimize usage. Measuring a service helps to provide guarantees for tenants (such as quality of service). Besides, it helps as a feedback loop in autonomic computing, allowing services to scale on-demand and to perform automatic failure recovery.

These characteristics contributed to the growth of cloud computing. For companies, the alternative to cloud computing is to invest in *on-premises* infrastructures (buying their own hardware and building the required infrastructure). This will not only require a high budget but also requires maintaining and managing hardware components including their security. Cloud computing helps to reduce this burden hence reducing the cost.

In the next sections, we describe different cloud deployment and service models.

2.1.2 Deployment Models

- *Private Cloud* is an infrastructure used only by a single organization. It can be owned and managed by the organization or a third party,
- *Public Cloud* is an infrastructure used by the general public. Owned and managed by a third-party cloud provider,
- *Community Cloud* is an infrastructure shared between several organizations from a specific community with common concerns (security, compliance, policy, etc.),
- *Hybrid Cloud* is an infrastructure comprising two or more different cloud deployment models (private, public, community). A hybrid cloud service crosses isolation and provider boundaries so that it cannot be put in one category. An organization can store sensitive data in a private cloud and outsource some services over the data (without compromising sensitivity) in a public cloud for additional capabilities.

2.1.3 Service Models

There are three main types of models to deliver cloud computing services. In general, they describe different layers of abstraction to show which cloud resources are offered. These are:

- *Infrastructure as a Service (IaaS):* Provides processing, storage or networking resources as a service. Tenants can deploy any software, which can include operating systems and applications. Examples of IaaS clouds include Amazon Web Services (Amazon Elastic Compute (EC2) [7] for computing and Amazon Simple Storage Service (Amazon S3) [8] for storage), Google Compute Engine [97] and Digital Ocean [67].
- *Platform as a Service (PaaS):* Provides resource to deploy tenants' application developed using languages and tools supported by the provider. Tenants do not manage or control the underlying cloud infrastructure (network, servers, operating systems ...). Examples of PaaS include Google App Engine [95] and Heroku Runtime [104].
- *Software as a Service (SaaS):* Provides applications running on providers' infrastructure and mostly accessed through the Internet. With the possible exception of limited user specific configuration settings, tenants have no control on the underlying infrastructure. Services are accessible through a web interface (CP) or API. Examples of SaaS includes Dropbox [52] and Google Apps [96]

Figure 2.1 shows the responsibilities of providers and tenants in managing various tiers of the cloud with different service models. There are inter-dependencies between these models (i.e usually SaaS and PaaS are running on top of IaaS). While the above classification is widely accepted the authors of [115] argued that it is not precise enough to give users and providers a common terminology to differentiate between existing cloud offers. In their paper they propose a more fine-grained abstraction. Example, IaaS services can be classified into *Hardware as a Service (HWaaS)* which gives tenants control on the HW including the OS and *Operating System as a Service (OSaaS)* which offers a fully managed OS including the underlying HW resources.

In practice a more fine-grained new service model like *Resource as a Service cloud (RaaS) [27]* has emerged. This is a transition from a model of selling bundles of resources packaged as a server (e.g a virtual machine with 4 CPU and 8 GB RAM) to continuously selling individual computing, memory, and I/O resources for a few seconds at a time (e.g Amazon Spot Instance [6] changes its price every 5 minute). RaaS is less reliable relatively to traditional IaaS models but it provides a computing service with a much cheaper price. (e.g Amazon Spot has a discount up to 90% compared to regular IaaS).

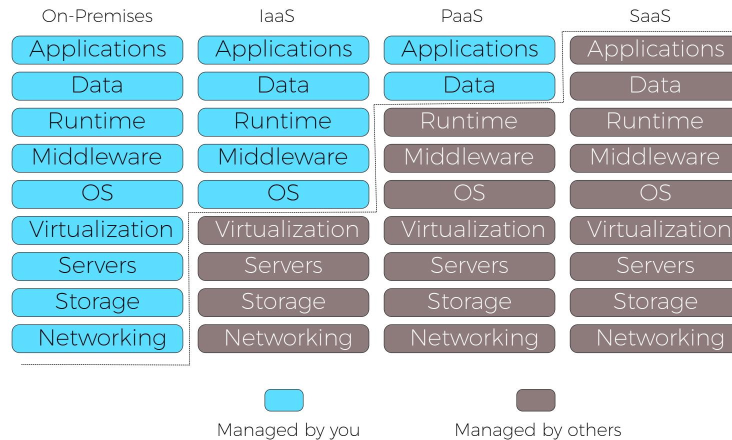


Figure 2.1: Responsibilities in different service models

Such a fine-grained model with the concept of Cloud Federation [42] and Supercloud [112] (a cloud that uses resources from various private and public clouds) is being used to increase reliability and remove the vendor lock-in problem which forces a tenant to be locked with a single service provider, which results a fear of price spike.

Virtualization is the main technology which enables such a dynamic computing utility service. In the next section we present a detailed description of Virtualization.

2.2 Virtualization

Virtualization refers to the act of creating a virtual (rather than actual) replica of an information system resource. The concept dates back in the late 1960s and early 1970s as IBM introduces CP/CMS time-sharing operating system with virtual memory support [32]. Since then there have been different studies on simulation of a full computer on a different system [94]. A machine created by such a technique is called *virtual machine*. In [178] GJ Popek et al. formally defined virtual machine as *"an efficient, isolated, duplicate of the real machine"*.

Virtualization has lots of advantages, one of its advantage is to maximize hardware resource utilization. With virtualization a single physical resource is shared between multiple users, where each user performs its operation as if she/he was using the actual machine.

Next we present virtualization techniques for two types of resources, *Machine* and *Network*. It should be noted that virtualization is not limited to these resources. We focus on technologies directly related to our PhD thesis work.

2.2.1 Machine Virtualization

Machine or server virtualization is the virtualization of a physical computer. It allows running multiple servers in a single physical computer, a server running on top of the virtualized environment is called a *guest* and a *host* can serve multiple guests. Each guest believes that it is running on its own hardware.

Figure 2.2 shows virtualized and non-virtualized systems. In a virtualized system, an intermediate component (labeled *Virtual Machine Monitor (VMM)*) is used to imitate the functionality of the physical machine for guest OSs running on the top. Different solutions were developed to achieve such functionality. We describe the major techniques in the following subsections.

Emulation

Emulation is the first attempt towards imitating one system in a different system. Historically such a technique has been used to play classic (and/or forgotten) console video games using programming code, which is interpreted by a current-era system. The goal is to mimic the behavior of a specific system. One of the major techniques that is used for emulation is *Binary Translation (BT)* [201, 51]. It maps

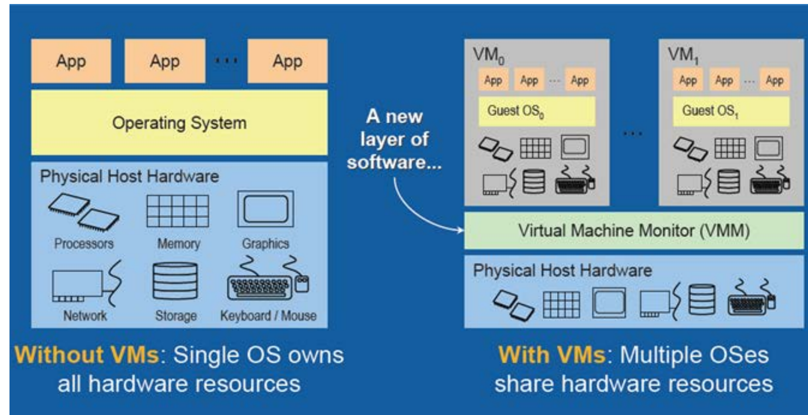


Figure 2.2: Virtualized and Non-Virtualized Infrastructure[1]

instructions from a given binary file to another set of instructions, with equivalent output, to be executed a on different architecture. When emulating a machine, the emulator reads instruction from guest OS and translate it into equivalent instructions using binary translation for the underneath physical device.

For example, say there two systems, *A* (something we already have) and *B* (which we may not have), emulating enables giving input of system B to System A but obtain the same result as system B. A and B can have different architectures. Many printers, for example, are designed to emulate Hewlett-Packard LaserJet printers because so much software is written for HP printers. Theoretically any system can be emulated within any other system. QEMU [26] and Bochs [30] are examples of machine emulators supporting execution of unmodified OS.

Full Virtualization

In full virtualization, the underneath hardware is simulated enough to run an unmodified OS as a guest and guest instructions are run natively on the CPU. Usually the virtual instance will has the same architecture as the underlying hardware. The guest is not aware that it is running on a simulated (virtualized) environment. The guest OS on a host is managed by the *VMM* also known as *hypervisor*. The hypervisor controls the flow of instructions to the CPU and input/outputs between the guest OS and the physical hardware like memory and network interface card. The hypervisor runs in the most privileged section of the system (close to the physical hardware) and it is responsible for the isolation of guests so that each of them has access to only its own resources (although it is possible to provide shared resource, like files in the host). Usually guests and the physical machine have similar architectures which is not usual case for emulation.

Full virtualization offers the best isolation and security for virtual machines, and simplifies migration and portability as it does not require the guest OS to be modified. But there are some instructions that are not virtualizable. These are sensitive instructions that can modify part of the machine's resources but do not trap (non-privileged) [82]. Hence, they can modify the state of a machine without the hypervisor managing it, which can have negative effect. Binary translation is used as one solution to overcome such problem. Examples of full virtualization technology include VMware Workstation [219], Kernel Virtual Machine(KVM) [118] and VirtualBox [217].

Hardware Assisted Virtualization

Another technique to support non-virtualizable instructions is by using hardware assistance. Different hardware components have been developed to simplify virtualization techniques. This is to decrease the overhead from software-only binary translation by having an additional hardware component. The hardware component is designed to support non-virtualizable instructions with a new CPU execution feature that allows the hypervisor to run in root mode with higher privilege than a regular architecture [138]. This allows trapping of non-virtualizable instructions and removing the need for binary translation. Intel VT-x [213] and AMD-V [11] are examples of processor extensions providing hardware assistance for virtualization.

[1] https://software.intel.com/en-us/articles/the-advantages-of-using-virtualization-technology-in-the-enterprise

Para-virtualization

As the name indicates ("Para" means "beside" or "alongside"), para-virtualization provides virtualization by means of the guest OS interacting with the hypervisor to improve performance and efficiency. The kernel of the guest OS requires *modification* to be able to run in a para-virtualized infrastructure. Some instructions are non-virtualizable and the modification replaces such instructions with *hypercalls*. A hypercall is a software trap from a VM to the hypervisor, just as a syscall is a software trap from an application to the kernel. The hypervisor provides an API to accept hypercalls. The hypervisor also provides hypercall interfaces for other critical kernel operations such as memory management, interrupt handling and time keeping.

The selling point of para-virtualization is lower overhead but such advantage can vary greatly depending on the workload. Para-virtualization also has poor compatibility and portability because it requires the guest OS to be modified. Xen [23] is the most prominent example of such technology.

In practice, existing virtualization services incorporate more than one of the methods. For example Xen [23] supports both full and para-virtualization for different components in the system.

Operating-system-level Virtualization

In operating-system-level virtualization the hardware is not virtualized. Instead, the host OS kernel allows the existence of multiple isolated user-space instances. This is different from the classical OS architecture where there is one user and one kernel space. The isolated instances are called *containers*. Applications running inside containers have the illusion to be in separate machines.

Such a method provides flexible and low-overhead virtualization which shares a single kernel. Each container can only run a system of the same type as the host environment, e.g. Linux guests on a Linux host. From a security perspective privileged containers (any container where the container uid 0 is mapped to the host's uid 0) are not and cannot be root-safe [135], i.e privileged processes within a container should be treated as privileged processes running outside the container.

Most known examples of such technologies include Docker [68], LXC [136] and Free BSD Jails [84].

2.2.2 Network Virtualization

Network virtualization is the process of creating software-based, isolated virtual network, potentially multiple networks, that are running above a single physical network infrastructure. It is mainly characterized by decoupling the network functionalities from the underlying physical infrastructure. Historically, network virtualization has been looked at from two perspectives (i) as a tool for evaluating new architectures (ii) as a fundamental diversifying attribute of the next-generation architecture itself [50].

After an application generates a message and prepares to send it over network, the messaging starts from the Network Interface Card (NIC) in the host. The NIC is connected to layer 2 (L2) network endpoint (Ethernet). Multiple L2 segments are connected one another with a switch to form a L2 network (a subnet). Multiple subnets connect via routers to form the Internet. Network virtualization refers to the virtualization of each component (NIC, switch, router ...) in this stack.

Making significant changes in the Internet architecture is almost impossible and changes are limited to small and incremental modifications. Network virtualization was proposed to overcome such impasse and make the Internet architecture more flexible [16, 212]. Network virtualization allows multiple heterogeneous virtual networks to coexist together in the same infrastructure but isolated from one another.

There have been different technologies for enabling multiple coexisting networks. Examples include *Virtual local area network (VLAN)*, *Virtual Private Networks (VPN)*, *active and programmable networks* and *overlay networks*.

- **Virtual Local Area Network (VLAN)**

  VLAN allows breaking single large flat networks into multiple broadcast domains. Hosts in the same VLAN, regardless of their physical connectivity, have the illusion to be attached to the same wire. Usually VLANs are constructed at the data link layer (L2).

  A simple VLAN divide network based on physical switch ports. Devices connected to the same set of ports can perform a direct communication.

  The most common type of VLAN is not based on physical ports, but is based on logical techniques (e.g adding tags on packets), hence flexible for administration and management. Frames in the

same VLAN have a common ID (*VLAN ID*), and VLAN-enabled switches use both the destination MAC address and the *VLAN ID* to forward frames. This type of VLAN is called *tagged VLAN*. VLANs can be implemented in hardware switches by assigning VLAN ID to a port. Such kind of VLANs are able communicate only with or through other devices in the VLAN. The IEEE 802.1Q standard [98] specifies to add an extra field to Ethernet frames to include a 12-bit *VLAN ID*, allowing dividing a single network over 4000 broadcast domains.

- **Virtual Private Networks (VPN)**

  A VPN allows creating a *private (secure)* communication channel between two or more hosts over a public network like the Internet. Packets to be sent through a VPN connection (*tunnel*) are encrypted and encapsulated inside another IP packet. The outer packet uses the end points of the VPN as a source and destination address, which are usually different from the inner packet addresses.

  Based on the layers where the VPN are constructed, they can be classified into different categories [78], namely *Data link, Network, Application layer VPNs. Data link layer VPNs* connect participants on layer 2 of the OSI model. Frame Relay and Asynchronous Transfer Mode (ATM) are example protocols used to construct data link VPNs. However, they are often expensive, because they require dedicated Layer 2 pathways to be created. *Network layer VPNs* are created using Layer 3 tunneling and/or encryption techniques. IPsec tunneling and encryption protocol, Generic routing encapsulation (GRE) [77] and Multi-Protocol Label Switching (MPLS) [190] are prominent techniques used for network layer VPNs. *Application layer VPNs* focus on specific applications. Examples include SSL-based VPNs and SSH. The drawback of application layer VPNs is that they require configuration for every type of application.

- **Active and Programmable Networks**

  Active and programmable networks [46] were another attempt to create a dynamic network architecture i.e an architecture where the message is not passed through static layers of protocols but dynamically selected by an application when the message is ready to be sent. Programmable networks aim to expose an interface to allow applications and middleware to manipulate low-level network resources to construct and manage services. Active networks go a step further and propose sending executable code to network elements using active packets in order to make "real-time" changes to the underlying network operation.

- **Overlay networks**

  Overlay networks are logical networks built on top of physical networks. The Internet started as an overlay network on top of public telephone networks. They are flexible and easily deployable compared to other types of networks [15, 49]. Overlay network is assumed to be the most recent ancestor of network virtualization in terms of the goal of resolving the ossification issue of the current Internet and of defining innovative network architectures and services [155].

A more recent and higher level concept in creating dynamic and efficiently manageable network infrastructures is *Software Defined Networking (SDN)*. The next section explains this concept in more detail.

Software Defined Networking (SDN)

In 2006 Ethane [38], a new network architecture for enterprise network management, was proposed. Ethane focused on using a centralized controller to manage policy and security in a network. The architecture contains two main components, a *controller* that determines the fate of all packets based on global policy and *Ethane Switches* which are dumb and contain a simple flow table and a secure channel to the controller. This laid the foundation for what is known as *Software Defined Networking (SDN)*.

SDN is a networking paradigm which proposes to separate the *network control* (component which makes decision) and *forwarding functions* (component which makes the actual forwarding of packets).

This enabled the controlling unit to be centralized, programmable and abstracts the underlying components for services running on the top. In [111] the authors described four innovations provided by the paradigm. These are:

- *Separation of the control and data planes:* The functionality in computer networks can be divided into three planes: the *data, control and management* planes. The *data plane* processes messages generated by users. The *control plane* processes message from network routing protocols like Open Shortest Path First (OSPF) or Spanning Tree. The *management plane* is in charge of traffic statistics and state of networking equipment. In traditional networking the *data* and *control* planes are embedded in the same networking devices. Even if decentralization makes the network resilient, it makes the architecture very complex and relatively static. SDN proposes to separate the data and control planes. The control plane implements the logic and intelligence and prepares the forwarding table. The data plane performs simple forwarding according to the table. This reduces the complexity, makes the forwarding device simple and aims to make the price of forwarding devices cheaper.

- *Centralization of the control plane:* The separated controlling unit is called *SDN controller* or *Network Operating System (NOS)*. Contrary to the standard objections to centralized approaches, SDN proposes to centralize the controller for easing management tasks. In [38] the authors argue that standard replication techniques with centralized controller can provide excellent resilience and scalability, which was the advantage of traditional networking. A central controller is responsible to program the forwarding plane using flow-based policies.

- *Programmability of the control plane:* One of the main value propositions of SDN is network programmability. Hence, the programmable control plane is the most important asset. This feature is a consequence of having a centralized controller. A centralized logic unit is easy to program and manipulate.

- *Standardization of APIs:* SDN has got enormous attraction from both industry and academia and a wide range of standardization efforts is being made around SDN. The Open Network Foundation (ONF) [162], an industrial-driven consortium to promote SDN and software defined standards, was created by a group of networking companies and service providers. Most of the implementations are coming from open source contributions [191], attracting a new wave of innovation.

  Figure 2.3 shows SDN architecture and different APIs. OpenFlow [142], dubbed 'a radical new idea in networking' [128], is a prominent API between the control and data planes (also known as southbound API). *Forwarding components (OpenFlow switch)* contain one or more *flow tables* and layers to securely communicate with the *controller* using the OpenFlow protocol. A flow table contains flow entries which are used to determine action against incoming packets. The actions are flow-based, i.e packets in the same flow are treated in the same manner. Other examples of southbound APIs includes OpFlex [204], POF [205] and ForCES [69].

  Figure 2.3 also shows SDN controller implementations, like Floodlight [79], OpenDaylight [166] and Nox [100]. FlowVisor [200] serves as a networking hypervisor which allows multiple logical networks to share the same OpenFlow networking infrastructure. Alternatively OpenVirteX [2] can be used as a proxy between the network operating system and the forwarding devices providing multi-tenancy over an OpenFlow network. Different types of such virtualization tools (e.g. AutoSlice [31] and FlowN [71]), have been proposed with slightly different sets of goals.

With the advances in SDN technology, various commercial SDN tools and services appeared in the market. NSX [218] from VMWare is a network virtualization and security platform for the Software-Defined Data Center (SDDC). It provides a complete set of logical networking elements and services (switching, routing, firewalling, quality of service (QoS), monitoring ...). SDN VE [127] is another network virtualization tool from IBM. SDN VE uses OpenDaylight [166] as one of its components and can support up to 16,000 virtual networks and 128,000 virtual machines with a single instantiation.

Recent studies [144] proposed to include another layer called *Knowledge Plane* to the standard SDN architecture. It is a plane based on machine learning and responsible for learning the behavior of the network and, in some cases, automatically operate the network accordingly. Such type of paradigm is also known as *Knowledge-Defined Networking*.

In [62] the authors discussed pros and cons of SDN from a security perspective. Centralization of the controller with traffic statistics collection gives the controller a global view of the network. This

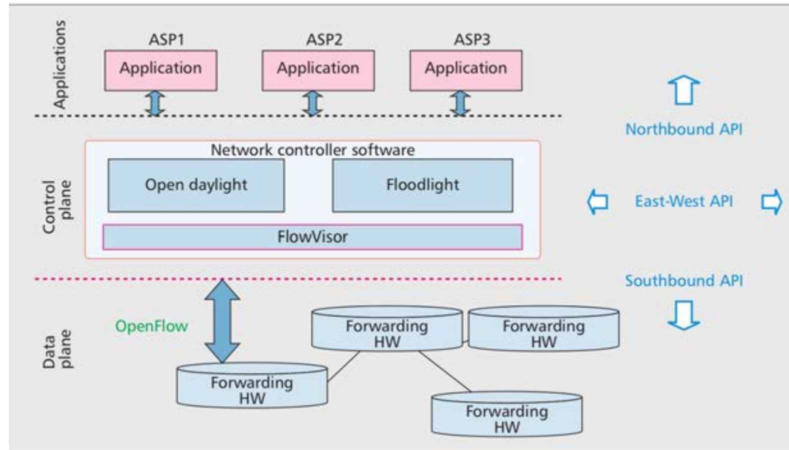


Figure 2.3: Software defined networking APIs [111]

helps to perform network wide intrusion detection, detection of switch's malicious behavior, and network forensics. The statistics with conditional rule also helps to reactively drop or redirect packets. Hence, it helps in detection of to Denial of Service (DoS) attacks and redirection to *honeypot*. Flow based forwarding also helps to have more control on packets and apply access control policies.

On the other hand each plane of SDN is more targeted to specific attacks. Upon the arrival of new flow, if there is no entry in the flow table matching for the new flow, the data plane forwards packets to the controller. Meanwhile, the packet is stored in the buffer of the data plane. This makes the data plane vulnerable to DoS, if a malicious user floods the switch using packets with large payloads that belong to different flows. Flow-based forwarding is also affected by encryption. Since it requires accessing some sections of the packet headers, encrypted packets could hide required information for such operation. In [76] this issue is addressed by constructing models to identify the payload type of the encrypted packets based on analyzing traffic statistics such as message length, inter-packet arrival times, etc. The control plane is also susceptible to DoS attack in the same way as the data plane. *Replication* is one of the solutions against this attack. Nevertheless, the forwarding plane should operate as if there was only one controller (i.e *logical centralization* of controllers).

SDN is an important technology enabling user driven computing utility services (e.g cloud computing). Specifically, it allows provision of isolated multiple networks, helps to dynamically configure each network separately. Hence, it helps to fulfill tenant's requirements without disturbing the configuration of other networks in the same infrastructure.

In the following section we present an example of IaaS platform which comprises most of the concepts that we have discussed. There are different implementations of IaaS platforms both open source and proprietary. We present *OpenStack* open source cloud platform.

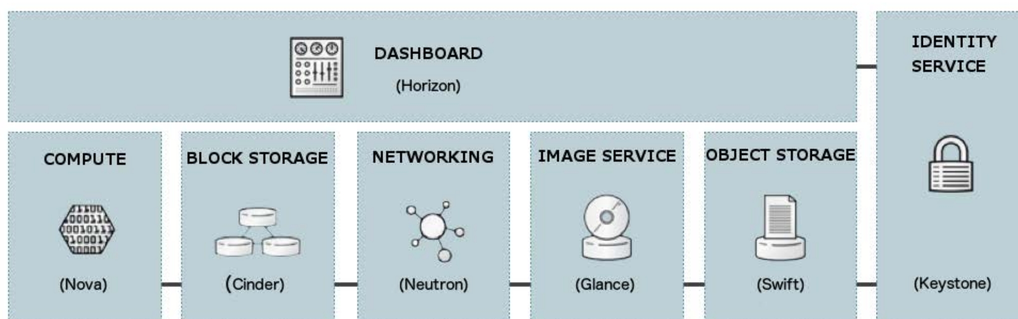


Figure 2.4: Openstack services overview[2]

[2]https://docs.openstack.org/security-guide/introduction/introduction-to-openstack.html

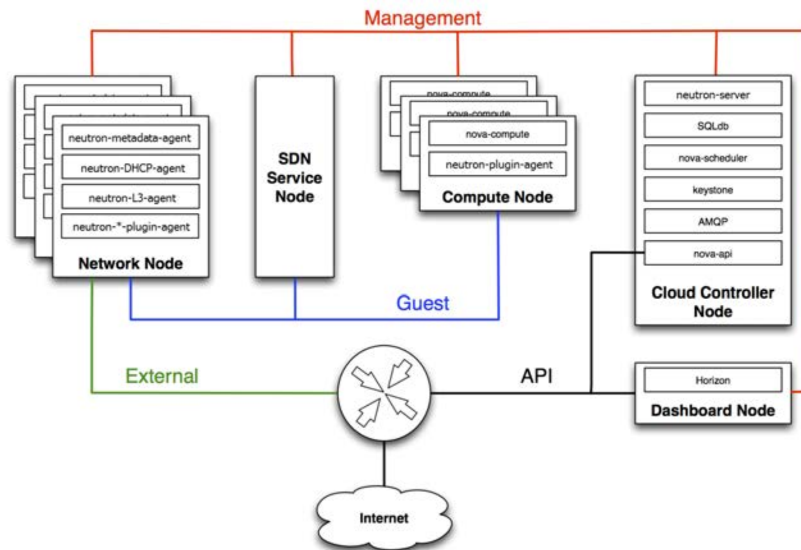


Figure 2.5: Openstack networking overview[3]

2.2.3 OpenStack, an Example IaaS Platform

OpenStack [163] is an open source software platform for cloud computing. It provides an IaaS solution through a variety of complementary services. Each service offers an API that facilitates integration. It is managed by the OpenStack Foundation and more than 500 companies have joined the project. OpenStack is designed with a highly modular architecture, simple implementation, massive scalability, and a rich set of features.

OpenStack is composed of different services including identity, networking, computing, storage dashboard and telemetry services that are communicating through an API. There are different types of implementations for each service; the default implementations are shown in Figure 2.4 (names shown in bracket). Different types of nodes host different services. There is a *controller node* as a central managing entity. It is responsible to accept requests from tenants through a dashboard and execute the requests with the help of other services, monitor available resources and related management tasks. *Compute nodes* host the actual virtual machines (also known as *instances*). The *network nodes* provides services related to networking. Figure 2.5 shows network connectivity of different nodes. Usually VMs are configured with two network connections, one for management and the other one for communication.

The high level steps to create an instance are as follows:

- A user needs to authenticate using the dashboard (Horizon) or command line interface (CLI). The authentication service (Keystone) is responsible to handle authentication requests and it sends back an *auth-token* which will be used for sending requests to other components.
- The dashboard then sends the *create* command to nova-api service.
- The nova-api service performs different checks (e.g. correctness of the auth-token, conflicts with existing VMs, ...), creates a database entry and sends requests to the nova-scheduler.
- The nova-scheduler locates an appropriate host and sends the request to nova-compute.
- Nova-compute gets instance information such as host id and flavor (RAM, CPU and Disk) from nova-conductor and the image URI from glance-api. It allocates network through the Neutron API, and prepares to attach volume to the instance through block storage (cinder-api).
- Finally, nova-compute prepares the request for the hypervisor of the target host and executes it on the hypervisor using libvirt.

All services perform different types of checks including the auth-token and passes results through a database.

[3]https://docs.openstack.org/security-guide/networking/architecture.html

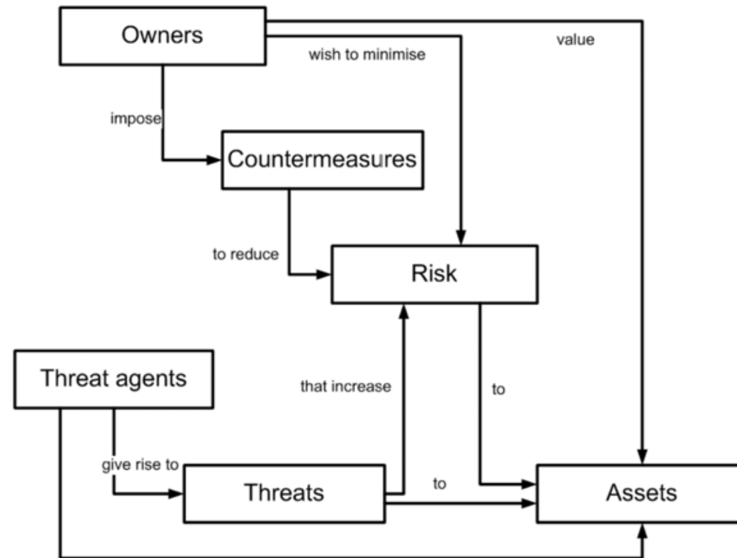


Figure 2.6: Security concepts and relationships (Source [55])

2.3 Information System Security

Information system security means protecting information and information systems from unauthorized access, use, disclosure, disruption, modification, or destruction. An *information system* as described in [126] is interrelated components working together to collect, process, store, and disseminate information to support decision making, coordination, control, analysis, and visualization in an organization. The form of data in an information system can be electronic or physical.

Figure 2.6 shows security concepts and their relations.

- *Owner*: of a system (an organization or a person) is an entity getting value out of a given system.
- *Assets*: are entities that the owner places value upon and security is concerned with the protection of assets. Since, the term "value" is highly subjective, almost anything can be an asset, including intangible things like reputation of an organization.
- *Threat agents*: e.g. hackers or malicious users, attempt to abuse or damage assets contrary to the intension of owners.
- *Threat*: is potential for impairment of the assets in a way that the value of assets to the owners would be reduced.
- *Risk*: the potential that a given threat will exploit vulnerabilities of an asset or group of assets and thereby cause harm to the owner. Threats increase the risk of damage on assets.
- *Countermeasures*: Actions taken to minimize risk on an asset. The countermeasures vary depending on the nature of the risk and possibility of being realized. They could range from software solutions (e.g. firewall, IDS or policy changes) to hardware or even physical guards around the asset.

In this thesis information system security refers to securing digital assets against software based threats. Hence, we do not consider protecting the infrastructure from physical theft. In the context of cloud, as it is described in Section 2.1.3, assets in each service model has different tiers of ownership. This with the multi-tenancy feature tangles security of clouds. Section 2.3.4 provides more details on different types of threats.

Security rules and regulations are described in the security policy. In general, the goal of a security policy is (not limited) to achieve security properties like confidentiality, integrity and availability. There are different mechanisms to reach the desired goal. In the following sections we describe security policy, properties and mechanisms.

2.3.1 Security Policy

The information security policy is an aggregate of directives, regulations, rules, and practices that prescribe how an organization manages, protects, and distributes information [81]. It is a definition of "what it means to be secure" for a system, organization or other entity. For organizations having security policy is crucial to create a secure working environment.

A security policy should be clearly written and as detailed as possible in order to facilitate its implementation. *Usability* is the most important property for a security policy. There is no purpose for a security policy if organizations or individuals within an organization cannot implement the guidelines or regulations defined. The simple existence of these policies does not automatically translate into desirable behaviors. It must be enforceable with security tools where appropriate. The security policy should also specify what auditing processes are put in place to verify compliance and actions to be taken in cases of non-compliance.

The three key principles surrounding information system security are *prevention, detection and response.*

- *Prevention*: is the process or actions taken to reduce the chance of arising security incidents. Prevention should be the first step but prevention does not always work. In fact most tools used to prevent incidents rely on reactive signature-based approaches and attacks are getting more and more sophisticated bypassing prevention tools.
- *Detection*: of security breaches is extremely critical. It determines that either an attack is underway or has occurred and detection helps to report it. The sooner the detection is the better to respond accordingly.
- *Response*: Once an incident is detected there must be a timely response. An owner can limit the breach impact and contain the exposure by having an incident responding plan and efficiently executing it when detection occurs.

Security policies should develop strategies around these principles. There must also be the right mix of investments across prevention, detection and response capabilities within an organization.

2.3.2 Security Properties

Three of the main properties in information security are *Confidentiality*, *Integrity*, and *Availability*, commonly known as the *CIA triads*. The goal is to achieve these properties with efficient policy implementation, all with acceptable overhead on the production environment. These three properties has been widely used to define security, including in US law [215]. They are defined as follows:

Confidentiality

Confidentiality is defined as preserving authorized restrictions on information access and disclosure, including means for protecting personal privacy and proprietary information. It should be noted that confidentiality is similar but not the same as privacy. Privacy as defined in [227] is the desire of a person or right of individuals to hold information about themselves in secret, free from the knowledge of others.

Integrity

Integrity is guarding against improper information modification or destruction. It includes ensuring information non-repudiation and authenticity. To maintain integrity, a system not only need to detect but also need to recover from integrity failure (improper modification). Integrity gives assurance of the accuracy and consistency of an entity over its life-cycle.

Availability

Availability is defined as ensuring timely and reliable access to and use of information by an authorized entity. This security property is not about keeping information in secret or protecting from tamper but it is about protecting disruption. In an information system where components are connected through a network, loss of availability can refer to a wide variety of breaks anywhere in the chain (e.g. network error in local machine or between the local machine and the Internet service provider, power outage or other software errors ...).

2.3.3 Security Mechanisms

It is important to be sure that all the security policies are enforced by mechanisms that are strong enough, otherwise it is impossible to be sure that the system is secure. It should be clear that even enforcing all defined policies does not guarantee full security, it can be counted as the best effort that an entity can make. Security mechanisms are designed to detect, prevent, or recover from a security attack. In this section we describe most common security mechanisms for protecting an information system. To facilitate the discussion, security mechanisms are categorized into two groups: *physical* and *technical* mechanisms. Technical mechanisms are further divided into *hardware, software and distributed algorithm-based* mechanisms. Even if there are some complementary mechanisms each of them address different types of security threats.

Physical security mechanisms

Physical security mechanisms are designed to physically deny access into the information system by keeping the resources behind a locked door and protected from threats of natural and human-made disasters. Physical security involves the use of different systems which include CCTV surveillance, security guards, protective barriers, locks, access control protocols etc.

Technical security mechanisms

Technical security mechanisms are applied in the system as part or beside the functionalities of the system. They are presented in three parts *hardware, software and distributed algorithm-based.*

- ***Hardware-based mechanisms***
  - *Trusted execution environment (TEE):* Mechanisms using a secure and isolated area of a main processor. A TEE ensures that data is stored, processed and protected in a trusted environment. AMD PSP [10], Intel SGX [58], ARM TrustZone [19] and Keystone [119] are examples of TEE.
  - *Memory Management Unit (MMU):* Hardware component, usually implemented as part of the CPU and used to perform virtual memory management. The MMU uses a page table to map virtual page addresses to physical page addresses in main memory. It is used to define separate memory contexts (allowing the CPU to switch between processes without reloading all the translation state information) and for privilege control i.e preventing access of system level areas by user level programs. The MMU is also used to implement address space layout randomization (ASLR) which prevents attacks that rely on known addresses.
- ***Software-based mechanisms***
  - *Authentication:* This is the most common type of security mechanism used to verify the identity of a user. It is often implemented by using a user name and password. It can also be coupled with other mechanisms (e.g hardware or cryptography based authentication). The security comes from the assumption that only a legitimate user can provide a correct combination of both user name and password.
  - *Access control:* As the name indicates it is a security mechanism that controls who can access resources in a system. It protects assets from being accessed by an unauthorized entity. Most of the time access control works with authentication so that access is approved based on successful authentication and based on what the subject is authorized to access. It should be noted that access control is a general concept and it is not only implemented using software-based mechanism.
  - *Traffic padding:* Traffic padding mechanisms keep traffic approximately constant, so that no one can gain information by observing it. This is achieved by producing a continuous packet output even in the absence of data to be sent. If there is no message to be transfered random data is generated and used instead. Sometime this technique is implemented with hardware support.
  - *Routing control:* A routing control mechanism makes it possible to choose a specific path for sending data through a network, hence avoiding undesirable nodes.

  - *Cryptography:* It is a security mechanism where mathematical computation is applied to protect private messages from third party or public. It is based on mathematical theory and cryptographic algorithms are designed around computational hardness assumptions. The security of such mechanisms arises from the fact that the underlying basic problem is hard enough that there is no feasible practical means to break it. Although it may be theoretically possible to break it.

    Cryptography is a generic term as a security mechanism. It is applied in various techniques including *encryption, digital signature, secure multi-party computation* etc ...

- ***Distributed algorithm based***

  - *Consensus algorithm:* Algorithm used to reach an agreement on a single data value between participants of a distributed system. Such type of algorithm is used to keep consistency of a given data across all participants, hence keeping integrity of the data. The security arises from the assumption that the majority of the participants are honest and the correct value is decided by the majority. A notable implementation of this method is used in digital currency (cryptocurrency) applications like Bitcoin [154] and Ethereum [34].

In practice different types of security mechanisms are used together to achieve a full round security. As an example users are requested to authenticate in order to get access to a web service. The communication between a user and a service should be secured to protect from man in the middle types of attack. In addition, the data (user name and password information) stored in the database should be protected using another type of security mechanism in order to prevent information leakage and unauthorized access.

Almost all cloud services require authentication and they provide an interface to submit user name and password. In some cases public-key cryptography (asymmetric key cryptography) is used for authentication. Most cloud service providers also claim to use encryption to protect users data from being accessed by third parties.

In the next section we present security threats in the cloud, specifically threats resulting from virtualization are presented.

2.3.4 Security Threats in the Cloud

Contrary to on-premises infrastructures which are managed by a single entity, the *-as-a-service cloud computing model presents challenges for security. The *-as-a-service model divides responsibilities between the service provider and tenants (see Section 2.1.3). For each component the security responsibility lies on the entity managing it. However, the consequence of a component not being secure may affect other entities too. Hence, securing the entire tier requires cooperation between different entities.

In the following section we present security threats in the cloud in two sections. First, threats related to the cloud features and second threats related to virtualization, which is the underlying technology enabling the cloud.

Threats related to cloud features

Takabi et.al [208] presented the unique issues of cloud computing that exacerbate security and privacy challenges in clouds. The authors discussed challenges in the cloud including the need to have technical and non-technical means to prevent cloud providers from abusing customer data.

In [88] the authors discussed basic features of clouds from their security implication perspective. Features like *scaling, transience, diversity and mobility* are presented. (i) *Scaling*: easy and rapid scaling presents a security challenge for an organization. Specially in cases when administrative tasks are not automated, scaling significantly multiplies the impact of catastrophic events. (ii) *Transience*: VMs last for a short period relative to traditional machines. This property could help an infected VM to appear for a short time, infect other machines, and disappear before it can be detected. (iii) *Diversity*: homogeneity helps organizations to enforce security policies (all machines run the most current patched software) and diversity creates management nuisance (e.g keeping track of different patches for different versions). (iv) *Mobility*: easy portability of VMs facilitates theft and creates difficulties while tracking security problems.

The cloud multi-tenancy feature introduces vulnerability like side channel (passively observing information) and covert channel (actively sending data) attacks [188, 109], as the attacker and target VMs can be co-located in the same host. Some refer to this as an *isolation* problem, adding it as a fourth security property. Ristenpart et al. [188] showed ways to place an attacker virtual machine (VM) on the same physical machine as the targeted VM, and then to construct a side channel between the two VMs on the same physical machine, which enables SSH keystroke timing attacks. In [109] the authors showed the possibility of cross-VM RSA key recovery when the attacker and target VM are co-located in the same host.

Threats related to virtualization technology

Virtualization is the main technology behind cloud computing. It creates a logical equivalent of physical devices and machines running on top of a virtualized environment are independent of the actual physical devices. This decoupling gives virtualization inherent security benefits. Nevertheless, it also introduces new threats and existing security issues also take new form in relation to virtualization.

In non-virtualized environments the OS highly trusts the underlying hardware, the same OS in a virtualized environment will have the same trust level for the virtual hardware, hence for the hypervisor (VMM). With such a trust model a malicious or compromised hypervisor may interfere with the VM. As a result secure virtualization relies on the security of the hypervisor and underlying hardware. Using a *Trusted Platform Module (TPM)* was proposed as a solution for such threats [172]. The TPM provides assurance for integrity of different components at the boot time including the hypervisor and hardware. In [173] the authors proposed the notion virtualized TPM to extend its functionality. Even if trusted execution provides a solution, overall the TPM does not protect against vulnerability exploits after boot time. There has been some works [224] showing the vulnerability of the TPM approach, specifically in its implementation.

The transparency property of VMM creates a challenge to detect malicious VMMs. Fully transparent VMM is a hypervisor which respects the three Popek's properties [178] (efficiency, resource control, equivalence). The equivalence property implies the VMM should look the same as the physical hardware (indistinguishable) from the application point of view. The transparent VMM with the traditional OS trust model creates an environment where a malicious VMM is trusted and could interfere with VMs without being noticed.

Virtual machine introspection [89] is an approach for inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it. It is achieved by the help of the VMM, which helps to interpose and mediate interactions between the virtual hardware and the guest software. Introspection helps in a security monitoring process, but it can also threaten security of virtualized information system. Using introspection technique a fully transparent VMM can alter operations inside VMs without being detected. Hence, if a VMM is untrusted or at risk of being compromised introspection threats should be taken into account. Examples of applications that can be altered using introspection technique include time-limited trial software [83]. To protect against such threats, a system needs to detect if it is running on top of a virtualized environment. Different VMM detection methods have been proposed, the most common method is by comparing time or resource discrepancies between a known physical environment and a known virtualized environment [90]. Pearce et al. [171] discussed more issues and security threats of virtualization.

2.4 Security Monitoring

Given the unavoidable nature of security risks, giving fast response to incidents is very important. For this purpose automated, continuous security monitoring is the key for quick threat detection and response.

In this section we present a detail description of security monitoring, types of hardware and/or software tools used to perform security monitoring and performance evaluation of security monitoring devices. We focus on specific monitoring devices, *Network Intrusion Detection Systems* (NIDSs), as the contribution of this thesis is directly applied to them.

2.4.1 What is Security Monitoring

According to Richard Bejtlich [25] security monitoring is defined as follows,

Definition 2.2. *Security monitoring* is the collection, analysis, and escalation of indications and warnings to detect and respond to intrusions [25].

Indicators are the outputs from a monitoring system. Indicators are conclusions formed by products, as programmed by their developers. Indicators generated by intrusion detection systems (IDSs) are typically called *alerts*. Indicators are not always correct, i.e. a monitoring device is not 100% accurate when indicating an incident. There are errors (e.g false positives) to some degree. Hence, we can classify indicators by their accuracy and measure the configuration of a monitoring system. Some configurations yield better value than others. Analysts analyze indicators and generate *warnings*. The process of bringing information to the attention of decision makers is known as *escalation*. It is important to escalate warnings as it requires an attention from a responsible entity.

Indicators are *collected* by hardware or software probe(s) which are designed for that purpose. Probes can perform *analysis* with the help of an input from a human. In the case of network security monitoring, the source of information for the probe is the network. Probes like IDSs get network packets as an input and perform analysis according to a given rule set.

The goal of collecting and analyzing events and generating indicators is to *detect* and prevent intrusions. In addition, when prevention eventually fails to *respond* to incidents as quickly as possible and understand how the intruder achieved its attack and what damage it made. Different types of security monitoring devices and techniques are used for different components. In the next section we describe the most common security monitoring devices.

2.4.2 Security Monitoring Devices

Security monitoring devices are tools that can be used to facilitate actions described in Definition 2.2 (collection, analysis, detection and response). In general any device (software or hardware) which generates logs (the log can be specific about security or a general functional log) can be used for security monitoring as any log indicates the behavior of the system and events that are happing in the system. From these logs collecting and analyzing devices can be used to perform security monitoring tasks. For example logs from firewalls and OS can be used for security monitoring purposes.

In the following we present devices that are used in the security monitoring process (collecting, analyzing, detecting and responding).

- **Antivirus**

  Antivirus (AV) are used for detection, prevention and removal of malware (viruses, worms, backdoors and other anomalies). Most of the time anti-virus tools take actions to protect the system from malware and generate logs. Such logs can be passed to collectors and aggregated from different sources to understand security incidents in a system. Modern antivirus software can protect from a wide range of computer threats by doing an active (real-time) scanning. Different methods are used to identify malware including sandboxed behavioral analysis, signature-based detection, heuristics, rootkit detection and recently data mining techniques. Avast antivirus [20], Microsoft Security Essentials (MSE) [146], ESET [147] are examples of commercially available antiviruses.

- **Vulnerability Scanners**

  Vulnerability scanners are tools used to analyze a system to detect vulnerabilities. The output of a vulnerability scanner can be passed to monitoring probes where an active monitoring can be performed. A vulnerability scanner is a proactive tool, i.e. it tries to identify vulnerabilities before they are exploited by an attacker. Most vulnerability scanners use signature-based techniques to identify vulnerabilities (i.e. they use a vulnerability database, like [56], that contains a list of publicly known vulnerabilities). Vulnerability scanners are the main components in a vulnerability management solution. There are two main approaches for scanning vulnerabilities [92] (i) *White box testing*: by analyzing source code of the application (ii) *Black box testing*: by analyzing the execution of applications (i.e. penetration testing which is a manual vulnerability scanning). Examples of vulnerability scanners include OWASP-ZAP [168], Nessus [156], OpenVAS [164] and w3af [221].

- **Firewalls**

  Firewalls are a collection of components interposed between two networks that filter the traffic between them according to some security policy [48]. Basically, they allow or deny connection

between two networks and generate logs on events and actions performed. A firewall can be configured at different locations in the network. One of the most common place to deploy a firewall is at the connection point of an organization with the outside world. A best practice is to categorize hosts inside an organization by trust level. Hosts that need to be accessed from the outside (e.g web or mail servers) have a lower trust level while hosts used for internal services have a higher trust level. Services that are accessed only by administrators have a much higher trust level. Firewalls are configured accordingly, i.e a firewall protecting the internal hosts block any access from the outside.

Even if firewalls are considered necessary in the general case they have some limitations. Configuring a firewall is not an easy task. Since they are customized to the needs of their environment it requires a careful verification and testing procedure in order to get the expected functionality. Fragmentation-based attacks can bypass firewalls, as most firewalls do not reassemble fragmented packets and make decision based on the first fragment. In an environment where there is high throughput firewalls may become congestion points. This is a result of the gap between processing and networking speeds. Science DMZ [63] is a network architecture for High Performance Computing (HPC) environments which proposes to remove firewalls due to the negative impact they have on performance.

- **Intrusion Detection System (IDSs)**

  IDSs are one the most common types of security monitoring devices in a computer system. Most of our work described in this thesis focuses on IDSs. Hence, we describe IDSs in more detail in the next section.

- **Log Collectors and Aggregators**

  Logs generated by individual devices provide information about that specific device. To get information at a higher level (overall system) and to have an easy management of logs it requires collecting and aggregating them in a logical manner. Logs from different components or systems may not have the same format, which is a bottleneck for analyzing, searching and reporting on the data. Aggregation helps to put all the data coming from different sources in one place for easy management of logs. Log collection can be done easily by copying log files using tools like rsync [193] or by configuring rsyslog [193] or syslog-ng [207], which implement the syslog protocol with additional functionalities, to transmit logs to a central location. There are also tools with more features like Apache Flume [105], which collect logs and store them on Hadoop Distributed File System (HDFS). In the cloud there are also log aggregation as a service tools like Amazon CloudWatch [4] and PaperTrail [170].

- **Security Information and Event Management (SIEM)**

  Security Information and Event Management is a combination of security information management (SIM) and security event manager (SEM). SEM provides correlation of events, notifications and console views with real-time monitoring features. On the other hand SIM provides long-term storage, analysis and reporting of log data. SIEMs help to aggregate and correlate data, generate alerts, retain data for long time and visualize using a dashboard. In addition, they are also used to generate reports for compliance purposes.

  Previously, the schema of SIEM tools was rigidly bounded and analyzes only structured data by design. However, new big data applications are starting to become part of security management software because they can help to clean, prepare, and query data in heterogeneous, incomplete, and noisy formats efficiently [37]. In addition, big data tools help to detect advanced persistent threats (APT). APTs operate in quiet mode over a long period of time without being noticed by monitoring tools like IDSs (deliberately avoiding recognition by security tools). SIEM tools are available as software, as appliances or as managed services. Examples of SIEM tools include OSSIM [3] and IBM QRadar [107].

2.4.3 Intrusion Detection System (IDSs)

In the late '70s and early '80s administrators used to search printed audit logs as a forensic tool to determine the cause of a particular security incident after the fact [117]. As storage becomes cheap, logs become available online and programs were developed to analyze the data [113]. However, the analysis was computationally intensive and slow. As a result it was done when the system was relatively in low load, hence intrusions were detected after they happened. Real-time intrusion detection systems that are used to analyze audit logs as soon as they are generated were developed in the early '90s. Since then different types of IDSs were developed and used to monitor computer systems.

IDSs are the "burglar alarms" (in our case, "intrusion alarms") of an information system. Formally they can be defined as software or hardware systems that automate the process of monitoring the events occurring in a computer system or network, analyzing them for signs of security problems [197]. Collecting information from source(s) is the first step in IDSs working procedure. Then collected information is analyzed using a detection engine and finally an output, known as 'alert', is generated if any intrusion is suspected.

According to NIST [197], all types of IDSs have the following key functionalities:

- *Recording information related to observed events*: information is collected locally and it may also be sent to a central SIEM system;
- *Notifying security administrators of important observed events*: alerts are generated and presented in different formats;
- *Producing reports*: reports summarizing observed events or details about specific alerts are produced;

In addition to these key functionalities IDSs can have other purposes like the following ones:

- *Identifying security policy problems*: as an example alerting when it sees network traffic that should have been blocked by the firewall but was not because of a firewall configuration error;
- *Documenting the existing threat to an organization*: the logs from IDSs can be used to understand the frequency and characteristics of attacks against an organization which can be used for different purposes including to identify appropriate security measures and educate employees about the threats that an organization faces.
- *Deterring individuals from violating security policies*: If an individual knows the existence of monitoring devices, she/he may be less likely to commit violation of policies.

One of the common characteristics of IDSs is that they do not to provide a hundred percent (100%) accurate detection of all intrusions. They can either misidentify legitimate events as an intrusion, creating *false positives*, or not detect an actual malicious event, creating *false negatives.* We know from [54] that there is no algorithm that can perfectly detect all possible intrusions. Therefore, the goal of a security administrator is to detect as many intrusions as possible while reducing false positives. Having a high number of false positives creates huge impediments for security administrators, i.e an administrator will spend a considerable amount of time on separating true positive values from false positives.

An extension to IDS technologies was developed, namely *Intrusion Prevention Systems (IPS).* In addition to detecting intrusions, IPSs take actions to prevent potential intrusions. IPSs use different techniques to perform prevention including, (i) Stopping the attack, for example by terminating the connection (ii) Changing the security environment, for example reconfiguring the firewall to block access (iii) Changing the attack content, for example removing an infected file attachment from an e-mail and allowing the cleaned email to reach its recipient.

Types of IDSs

Based on different criteria, IDSs can be divided into groups. In this section we describe categories of IDSs based on the detection method (*signature-based* or *anomaly-based*) and based on the location where the detection takes place (*Network IDS* or *Host IDS*). First, we describe the types of IDS based on their detection technique.

- **Signature-based IDS** refers to a detection method where observed events are compared with predefined sets of specific patterns known as *signatures*. A signature describes malicious content in the observed entity, such as byte sequences in network traffic or known malicious instruction sequences used by a malware. This type of detection method, as it is dependent on the definition of signatures, is very effective in detecting known attacks but it fails to detect new attacks. Other drawbacks of such method is that it has little understanding of states (protocols) and it requires keeping the database of known attacks up to date. Examples of signature-based IDS include Snort [13] and Suricata [206].

- **Anomaly-based IDS** refers to a detection method where a normal behavior of the monitored system is modeled and events are compared with that model to detect abnormal behaviors. Such modeling of normal behavior gives advantage to detect unseen (unknown) attacks. However, anomaly-based IDSs produce a high number of false positives. In addition, building a normal behavior of a system is time consuming, challenging and requires extensive "training sets". It should also be noted that at the time of training (preparing the model) the IDS is not functional. There is not a lot of widely used open source anomaly-based IDS. Most of the time tools developed for monitoring purposes are adapted to perform anomaly detection. Examples of such tools include Bro [210] and RRDtool [192]. Cisco NGIPS [157] is an example of commercial anomaly-based IDS.

These two detection approaches are complementary and many systems attempt to combine both of them. The problem of false positives in anomaly-based IDSs cause most commercial IDSs to focus on signature-based methods.

Based on the location where the detection takes place, IDS can be classified into *Network IDSs* and *Host IDSs*.

- **Network IDSs (NIDS)** monitor network packets to detect intrusions. NIDSs are placed at strategic points within the network to monitor traffic to and from all devices on that network. This type of IDS is the most common type of commercial IDS and it can be used on-line, dealing with the network in real time through network taps, or off-line, dealing with stored data.

- **Host IDSs (HIDS)** monitor individual hosts, like incoming and outgoing traffics of a device in a network, modification or access to critical files in a system, running processes, application activity and so on. Host-based IDSs are most commonly deployed on critical hosts such as publicly accessible servers and servers containing sensitive information. As in the case of NIDS, there can be on-line and off-line HIDSs. OSSEC (Open Source SECurity) [196] is a prominent example of HIDS.

From now on, when the term *IDS* is used it refers to both NIDS and HIDS. For specific cases the terms *NIDS* and *HIDS* are used.

In an environment where events are generated at a high rate the IDS engine can be assisted with special hardware [150, 195] to get significant amount of speed gain. For example the hardware based regular expression engine for the SNORT IDS presented in [150] gives a maximum speedup of *353*× over a software-based solution.

IDSs can also get fooled by an attacker. *IDS evasion* is a class of techniques to deliberately modify attacks in order to avoid detection by IDSs. Most IDSs, specially signature-based IDSs, are vulnerable to evasion. There are different ways to evade an IDS. Common types of evading techniques are described in [47].

- *Denial-of-service (DoS)*: Exhausting IDS's resources (e.g CPU and the memory space) in order to disrupt and deny its detection service. The exhausting technique can be implemented in different ways, by sending a large number of requests or by exploiting a bug as in case of packets of death [169]

- *Packet splitting*: Like IP fragmentation, packet splitting is chopping data into smaller packets so that the NIDS may not reassemble the packets for signature matching.

- *Duplicate insertion*: With duplicate insertion attackers insert duplicate or overlapping segments (or IP fragments) to confuse the NIDS. If the NIDS and the target system handle the duplication in different ways an attacker may succeed without being detected by the IDS.

- *Payload mutation*: In payload mutation an attacker transforms the malicious packet payloads into semantically equivalent ones. Therefore, the payload looks different from the signatures that an IDS expects, so the attack can evade the detection.

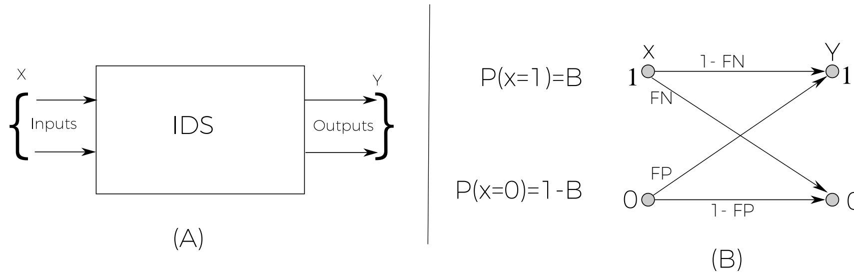


Figure 2.7: Block representation of IDS

- *Shellcode mutation*: The Shellcode mutation encodes a shellcode into polymorphic forms to evade an IDS. If the IDS does not recognize one or a few variants of that shellcode it may get evaded.
- *Encrypted Tunnels*: An attacker may use an encrypted tunnel in order to move across networks without being detected.

In this thesis most of our work directly applies to signature-based NIDSs. However, application of the proposed approach to HIDS should be straightforward and adapting the work to other types of monitoring devices including anomaly-based IDSs should be feasible with additional studies.

2.4.4 Measuring the Performance of Security Monitoring Devices

In this section we describe the metrics used to measure the performance of security monitoring devices. Each device has specific attributes that are relevant to be measured and in this section we focus on attributes of NIDSs.

Measuring the performance of NIDS

Some of the fundamental questions around NIDS (and IDS in general) technologies rotates on *evaluating the performance of an IDS*, *determining the best configuration of an IDS* and *comparing two or more IDSs*. To answer these and other related questions it is required to have metrics, that can be used to quantify different aspects of NIDSs.

The performance of an NIDS can be measured mainly from two aspects:

- *Efficiency* measures the resources needed for the functioning of the IDS itself, resources like CPU, memory and disk usage.
- *Effectiveness* measures the classification ability of an NIDS, i.e. how much it correctly differentiates intrusive and non-intrusive activities.

Other aspects of NIDS including resilience to stress [182], ability to resist to attacks directed at the NIDS itself [181, 47], ease of use, interoperability and transparency can be measured. These properties are correlated. For example a change in the resource used by an NIDS will affect the effectiveness or making an NIDS resilient to attacks may require more resources. Here we are interested in the *effectiveness* of an NIDS as it enables us to compare different NIDSs and to fine-tune an NIDS, i.e find the best configuration of an NIDS.

In the rest of this section we describe different metrics that are used to express the effectiveness of NIDS. An NIDS can simply be modeled as a system which takes a series of events and classifies those events into groups of either *legitimate* or *non-legitimate (intrusive)* events. Figure 2.7 (A) shows the box model of an NIDS.

NIDSs take flows of network packets as an input. Every packet in a flow is processed by the engine and classified to either normal or abnormal. In addition, the inputs in reality are either attacks or legitimate traffics. We can say the classification is correct if the output of an NIDS matches with the actual grouping. For binary classification with two input and two output values, there are four possible outcomes, Table 2.1 shows these possibilities (also known as confusion matrix).

		NIDS output	
		Attack	Normal
Actual	Attack	TP	FN
	Normal	FP	TN

Table 2.1: Confusion matrix showing IDS outputs

Metrics	CR	TPR	FPR
Formula	$=\frac{TP+TN}{TP+FP+FN+TN}$	$=\frac{TP}{TP+FN}$	$=\frac{FP}{FP+TN}$

Table 2.2: CR, TPR and FPR as metrics derived from TP, FP, FN and TN

- *True positive (TP)* is intrusion that is successfully detected by the NIDS.
- *False positive (FP)*is non-intrusive input wrongly classified as intrusive.
- *True Negative (TN)* is non-intrusive input successfully identified by the NIDS as normal.
- *False Negative (FN)* is intrusive input that is wrongly classified by the NIDS as normal (missed attacks).

The confusion matrix shows a representation of the possible outcomes, different evaluation metrics are derived from these probable outcomes. These includes:

- **Classification rate (CR)**: Also known as *accuracy*, measures the rate of correctly classified events (both normal and intrusive). It can be calculated as shown in Table 2.2
- **True positive rate (TPR)**: Also known as *detection rate*, shows the ratio of correctly detected attacks. It is calculated as the ratio between the number of correctly detected attacks and the total number of attacks (see Table 2.2).
- **False positive rate (FPR)**: The FPR is defined as the ratio between the number of normal events detected as attacks and the total number of normal events (see Table 2.2).

Alternatively, from an information theoretic point of view [99], NIDS can be modeled as shown in Figure 2.7 (B). Every input of an NIDS is either intrusive or normal traffic. Hence, inputs can be represented with a random variable x. Similarly, output is decision by NIDS classifying inputs into either attack or legitimate. We can represent the output with a random variable y. The value $x = 1$ represents an intrusive input and $x = 0$ indicates normal traffic. The same way, $y = 1$ indicates IDS classified the corresponding input as intrusive and $y = 0$ indicates NIDS's classification as non-intrusive.

Using this model, we can define the FPR as the probability of $y = 1$ when $x = 0$ or $P(A|\neg I)$, i.e the probability that there is an alert, A, knowing that there is no intrusion, $\neg I$. The same way $TPR = P(A|I)$, i.e the probability that there is an alert, A, knowing that there is intrusion, I. Note also that we can define FNR and TNR, which are complementaries of TPR and FPR respectively (i.e $FNR = 1 - TPR$ and $TNR = 1 - FPR$).

We cannot use TPR or FPR separately to describe the performance of an NIDS or to compare different NIDSs. Having a TPR high value is not useful if there is a large number of false positives. Also, having a low FPR value does not help if there are too many missed attacks. Hence, TPR and FPR are coupled metrics.

As it is described in [21, 86, 99] TPR and FPR are basic metrics and they can be misleading when describing the performance of IDS. To explain this, assume there are two IDSs (IDS_1 and IDS_2). IDS_1 can detect 10% more attacks, but IDS_2 can produce 10% less false alarms, deciding which IDS is better using only this information is difficult. Rather than coupled metrics, having a single metric describing effectiveness of an IDS helps to make decisions.

The tradeoff between TPR and FPR can be intuitively analyzed with the help of the *receiver operating characteristic (ROC) curve.*

- **Receiver Operating Characteristic (ROC) curve**: ROC originated from signal detection theory [72] and it is used to correlate TPR and FPR. It helps to visualize the relation between TPR and FPR using graphs and it shows the probability of detection provided by the NIDS at a

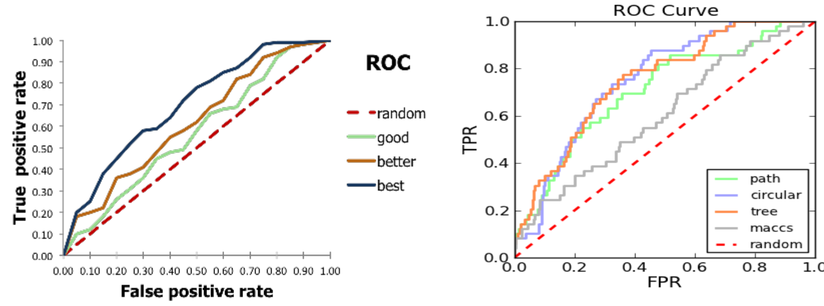


Figure 2.8: Examples of ROC curves

given false alarm rate. By itself ROC cannot be used to compare different operation points of an NIDS (a single NIDS with different configurations) but it can be used to compare different NIDSs. If $NIDS_i$ has higher TPR value for every FPR value than $NIDS_j$ the ROC curve of $NIDS_i$ will be always on top of $NIDS_j$ (see Figure 2.8 (left)). But if the curves of $NIDS_i$ and $NIDS_j$ crossed each other, it is not easy to identify which one is a better IDS. (see Figure 2.8 (right)). Hence, it makes ROC curves less useful.

Computing the area under the curve to compare crossing ROC curves was proposed. But computing the area takes every operation point into account, this is not the case for NIDS because NIDSs are fine-tuned on a specific operation point. ROC is basically dependent on TPR and FPR. In [21, 86, 99] it is shown that the information provided by TPR and FPR is not enough to describe NIDSs performance. Specifically these metrics do not take the NIDS operation environment into consideration. Examples of parameters describing NIDS operation environments include maintenance costs and the hostility of the operating environment (the likelihood of an attack).

In an effort to take NIDSs' operation environment into consideration while evaluating its performance, different metrics were proposed taking operation environments such as *uncertainty of the environment* and *costs* spent as a result of NIDSs' classification error (i.e the cost of responding for false positives and cost of not responding for intrusion) into account.

- **Base rate fallacy, Positive and Negative Predictive Values (PPV and NPV)**: Axelsson [21] pointed out the effect of uncertainty (likelihood of an attack) in the performance of NIDSs, the base rate fallacy problem in NIDSs. The problem raises as a result of the difference between the number of false alarms and actual intrusions. NIDSs generate a large number of false alarms in an environment where there are not many intrusions to begin with. Correctly identifying intrusions from the flood of network traffic is truly a problem of "finding a needle in a haystack".

  Let's assume I indicates intrusion and A indicates alarm. The reverse, $\neg I$ indicates normal event and $\neg A$ indicates no alarm. The likelihood of an attack can be represented by $P(I)$, the probability of intrusion. From Figure 2.7 (B) $P(I) = P(x = 1)$, and it is called the *base rate* value. It is the parameter which quantifies the hostility of the environment. But in practice, it is difficult to measure the actual value of the base rate. It is because the occurrence of an attack is not deterministic.

  We would like to know how many of the alerts are "true" indications of intrusions, i.e the probability of an intrusion knowing that there is an alert (i.e $P(I|A)$). This metric is called *Bayesian detection rate* or *Positive Predictive Values (PPV)*. Applying the general form of the Bayes' theorem in NIDSs events,

$$PPV \equiv P(I|A) = \frac{P(I).P(A|I)}{\sum_{i=0}^{1} P(x_i).P(A|x_i)} = \frac{P(I).P(A|I)}{P(I).P(A|I) + P(\neg I).P(A|\neg I)} \quad (2.1)$$

  Substituting $P(I)$ with base rate (B), $P(A|I)$ with TPR, and $P(A|\neg I)$ with FPR we get,

$$PPV = \frac{B \times TPR}{B \times TPR + (1 - B) \times FPR} = \frac{B \times TPR}{B \times (TPR - FPR) + FPR} \quad (2.2)$$

To explain the base rate fallacy problem, assume an environment where there is only 1 attack event in 100,000 events (i.e $B = 10^{-5}$). Take an NIDS in this environment which detects all attacks (i.e $TPR = 1$) and only gives 1% of false alarms ($FPR = 0.01$). The probability of an alarm indicating an intrusion can be computed as

$$P(I|A) = \frac{10^{-5} \times 1}{10^{-5} \times (1 - 0.01) + 0.01} \approx 0.001 \tag{2.3}$$

That is, even though the NIDS detects all intrusions ($TPR = 1$) with very low false alarms ($FPR = 0.01$), which is unrealistic, only 1 out of 1000 alarms is a real intrusion and the remaining are false alarms. To put it differently, in our hypothetical environment, for approximately half of the alarms to be true intrusions (i.e to get $P(I|A) = 50\%$), it is required to have a very low false alarm rate ($FPR \approx 10^{-5}$).

In [21] the author describes the advantage of PPV in realistic environments. PPV is more useful for security officers because low PPV values indicates to *safely ignore all alarms*, even though their absolute numbers would theoretically have allowed complete investigation of all alarms. From Equation 2.2 we can see that PPV increases as FPR goes to zero, regardless of the value of TPR.

To incorporate the effect of changes in TPR, *Negative Predictive Values (NPV)* or ($P(\neg I|\neg A)$) is defined and there is trade-off between the PPV and NPV.

$$NPV = \frac{(1 - B) \times (1 - FPR)}{(1 - B) \times (1 - FPR) + B \times (1 - TPR)} \tag{2.4}$$

From formula 2.4 we can see that PPV and NPV are a function of B, TPR and FPR. As in the case of TPR and FPR, PPV and NPV are coupled metrics. In the field of Artificial Intelligence (AI) a metric called Bayesian Receiver Operating Characteristic (B-ROC) curves [35] is proposed which correlates PPV and NPV with the same intuitive as ROC curves. But B-ROC, rather than using FPR, uses (1 - PPV) called Bayesian false alarm rate $P(\neg I|A)$ and it shows what percentage of generated alarms are false positive. But B-ROC have issues similar to ROC.

- **Cost Based Metric**: To overcome the limitations of ROC curves, another metric was proposed to better integrate TPR and FPR. In [86, 36] ROC curves and cost analysis are combined to compute *expected cost (C_{exp})* for each NIDS operation point. The response of NIDSs for an incoming event is either generating alert or not. When an NIDS generates an alert it has consequences, i.e security operators need to check the alert and take appropriate action. The expected cost quantifies the response of security operators to alerts. Then the goal is to select an operation point with the least expected cost.

  The cost is defined for actions taken as a result of misclassification by NIDSs (as a result of FPR and FNR). The cost of actions for correct classification (TPR and TNR) is usually assumed either to be zero or as a profit deductible from errors.

  In [86] the following cost estimation method is presented. Let's assume α represents FPR and β represents FNR, then C_α indicates the cost of FPRs and similarly C_β shows the cost resulting from FNR. For a given point the cost ratio is defined as $C = C_\beta / C_\alpha$. The expected cost of operating at a given point on the ROC curve is calculated using a decision tree model (see [86]). For a given point it can be calculated as,

$$C_{exp} = Min\{C\beta B, (1 - \alpha)(1 - \beta))\} + Min\{C(1 - \beta)B, \alpha(1 - B)\} \tag{2.5}$$

  It is clear that C is a very important factor in determining the expected cost. In practice choosing the value of C is very difficult as it is highly subjective. Gu et.al [99] proposed a metrics called *Intrusion Detection Capability (C_{ID})* which objectively measures the effectiveness of an NIDS.

Intrusion Detection Capability(C_{ID})

The C_{ID}, presented in [99], originated from the field of information theory. Generally we want the NIDS to reflect the actual grouping of inputs i.e either intrusion or normal. From an information-theoretic point of view, we should have less uncertainty about the input given the IDS output. C_{ID} incorporates most of the other metrics including TPR, FPR, PPV, NPV and B. C_{ID}, as it is shown in [36], can also

be expressed as an expected costs optimization problem. To describe C_{ID}s' formula in detail we first present a background on the basics of information theory as presented in [59].

Assume two random variables X and Y as shown in Figure 2.7 (B).

- **Entropy**: Given a random variable X, the entropy is a measure of uncertainty in the random variable X. It is defined as

  Definition 2.3. $H(X) = -\sum_{x \in X} p(x) \log p(x)$

  The higher $H(X)$ shows the more uncertainty in variable X. This definition can be extended to pairs of random variables *(X, Y)*. The uncertainty in *(X, Y)* is called *joint entropy.*

- **Joint and Conditional Entropy**: Given a pair of discrete random variables *(X, Y)* with a joint distribution *p(x,y)*, the joint entropy can be defined as

  Definition 2.4. $H(X,Y) = -\sum_{x \in X} \sum_{y \in Y} p(x,y) \log p(x,y)$

  It is similar to single variable entropy if (X, Y) can be considered to be a single vector-valued random variable.

  Conditional entropy is the uncertainty of one variable knowing the other variable. $H(X|Y)$ shows the remaining uncertainty of X after Y in known. It is defined as

  Definition 2.5. $H(X|Y) = -\sum_{y \in Y} \sum_{x \in X} p(x,y) \log p(x|y)$

  From the above definitions we can see that the entropy of a pair of random variables is the entropy of one plus the conditional entropy of the other. That is also called *Chain rule*

  $$H(X,Y) = H(X) + H(Y|X) \tag{2.6}$$

  If the value of X is completely determined by the value of Y, we can say $H(X|Y) = 0$. If we can certainly determine the value of $H(X)$ given H(X|Y) then we can say $H(X|Y) = H(X)$, i.e X and Y are completely independent. $H(X|Y)$ has the following property

  $$0 \leq H(X|Y) \leq H(X) \tag{2.7}$$

- **Mutual Information**: Given two random variables X and Y with a joint probability mass function $p(x,y)$ and marginal probability mass functions $p(x)$ and $p(y)$, the mutual information $I(X;Y)$ is defined as

  Definition 2.6. $I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}$

  It is a measure of the amount of information that one random variable contains about another random variable and vice-versa. It is a shared information between X and Y. Hence, $I(X;Y) = I(Y;X)$. Mutual information can also be expressed as reduction in the uncertainty of one random variable due to the knowledge of the other. That is

  $$I(x;y) = H(X) - H(X|Y) = H(Y) - H(Y|X) \tag{2.8}$$

  This shows the relationship between conditional entropy and mutual information. This can also be represented in a Venn diagram as shown in Figure 2.9 (it is also presented in [99]). The intersection area indicates mutual information.

As described above, we want to be less uncertain about the input of an NIDS given its output. As shown in Figure 2.9 (B), in a realistic environment the actual number of intrusions is far less than the number of alerts generated by NIDSs. Mutual information could be used to measure the uncertainty in the inputs, because it captures the reduction of original uncertainty (intrusive or normal) given that we observe the NIDS alerts. C_{ID} measures exactly that, it is the ratio of the reduction of uncertainty of the NIDS input, given the NIDS output, formally it is defined as follows,

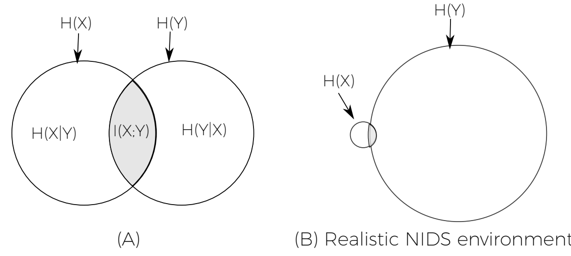


Figure 2.9: (A) Venn diagram showing entropy and mutual information (B) Realistic NIDS situation, where the number of actual intrusions are very small.

Definition 2.7. Let X be the random variable representing the IDS input and Y the random variable representing the IDS output. The Intrusion Detection Capability is defined as

$$C_{ID} = \frac{I(X;Y)}{H(X)} \tag{2.9}$$

The mutual information is normalized by the original uncertainty $H(X)$, and its value range is [0,1]. The higher C_{ID} indicates a better effectiveness of an NIDS. By expanding Equation 2.9, we can observe that C_{ID} is a function of *TPR, FPR, PPV, NPV and B.*

In an ideal case where all classifications are correct ($FP = FN = 0$), $C_{ID} = 1$ as the IDS is not making mistakes. For cases when the inputs are all in one group (all legitimate, $B = 0$ or all attack, $B = 1$), C_{ID} is defined to be one because $H(X) = 0$. In a realistic situation, where the base rate is very low, a decrease in FPR gives higher C_{ID} than the same change in FNR. This indicates C_{ID} is more sensitive to FPR than FNR. Given the NIDS output (set of alarms) the goal is to identify which alarms are a true indication of attacks. In a realistic situation, where the rate of occurrence of attacks is very small, from large set of alarms a very tiny portion is a true intrusion indicator. Thus, for a metric to be more sensitive to the rate of false alarms (FPR) helps the entity managing the output of the NIDS to safely ignore alarms when the C_{ID} is lower than some threshold.

Hypervisor Factor (HF)

In [149] the authors presented the effects of elasticity on the NIDS attack detection accuracy. In a virtualized environment the resources available to an NIDS is not fixed (elasticity), hence the performance may vary accordingly. They argue that existing metrics cannot evaluate virtualized environments in an accurate manner. They proposed a metrics called *hypervisor factor (HF)* which is a cost-based metric, generated using a decision tree model as presented above. *HF* is different from other metrics, it takes into account the resources available for NIDS while making decision, whether the NIDS had enough resource (*Baseline*), it had more resources than required (*Overprovisioning*) or it had less (*Underprovisioning*). It adjusts the cost accordingly.

The common difficulties in all metrics is the inclusion of unknown parameters in their formulas. Either it is a *base rate* or *the cost* of NIDS misclassification. In practice it is difficult to find the value of base rate before the intrusion happens. Finding the cost of misclassification may be easier but it is highly subjective, as the effect of an intrusion is different for every organization or even for the same organization on different occasions.

Mechanisms to measure NIDSs performance

In the previous section we presented metrics that are used to describe the performance of NIDSs. In this section we present mechanisms used to perform the evaluation. Generally we can categorize evaluation techniques into three groups:

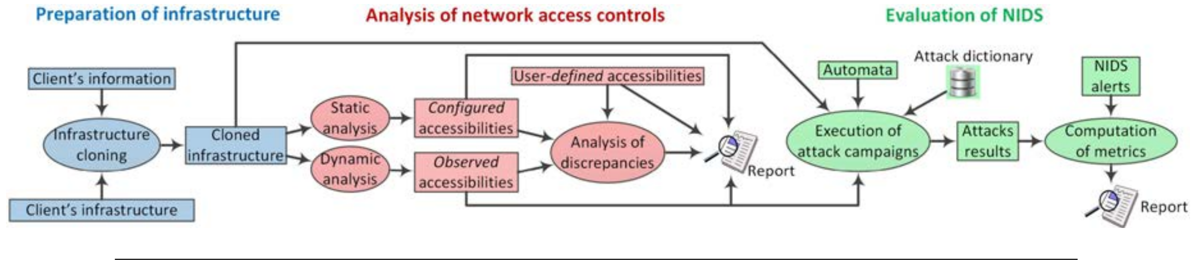


Figure 2.10: NIDS evaluation process as proposed in [180]

- *On-line evaluation*: also known as dynamic evaluation, is done by configuring the NIDS, injecting attacks and analyzing the output of the IDS. The injection could be done live (real-time) or replaying recorded traffic.
- *Off-line evaluation*: also known as static evaluation, is done by statically analyzing configuration files, rules and behavior models of an NIDS.
- *Hybrid*: evaluating by using both dynamic and static approaches and aggregating the results.

One of the challenges in evaluating NIDSs is the generation of representative traffic traces that contain attacks. In 1998 DARPA [130] started a series of annual intrusion detection evaluations (updated until 2000) with the goal of exposing strength and weakness of NIDSs to drive iterative performance improvements. It used an on-line attack injection technique and evaluated large systems. In addition, a large network flow trace containing both background traffic and attacks was generated and shared with in the research community. The shortcomings of the DARPA dataset including the taxonomy developed to categorize the exploits, the metrics used to present the results of the evaluation and the bias on the process of generation are presented in [141].

Massicotte et.al [139] proposed a traffic trace generation technique and an NIDS evaluation framework. The trace generation process produces a documented traffic trace which is used to test NIDSs in an automatic fashion. The traffic generation environment is an isolated virtual infrastructure used to run attacks and record traces. The NIDS evaluation framework takes these data sets and replays it on a target NIDS. Finally, the output of the NIDS is analyzed. However, generating traces in isolated environments may help to get "clean" trace but it removes some uncertainty that could happen in real environments. It eliminates the behavior of a real operation environment of an IDS from the evaluation process. Hence, using traces generated by such a process misleads to describe the performance of an NIDS.

Probst et.al [180] presented an NIDS evaluation method on cloud environments. The paper presents a hybrid approach which is based on cloning the target infrastructure, static and dynamic analysis of configured firewalls to have knowledge of network accessibilities. The output of a hybrid analysis is used for the attack injection process. The paper claims it is "expensive, tedious and hard to automate" heterogeneous cloud environments and instead the authors used a finite-state automaton as a representation. To evaluate an NIDS, an attack campaign is executed on a cloned infrastructure taking the automaton and the result of a hybrid analysis as an input. The NIDS output is analyzed to compute final results (see Figure 2.10).

Doing the evaluation on a cloned environment reduces the network overhead on the production resources but it also takes the NIDS out of its real operating environment. In addition, both [180] and [139] assume only basic metrics for evaluating NIDSs. As it is shown in [21] not taking the *base rate* parameter into account misleads the description of the NIDS performance.

Milenkoski et al [148] presented an approach for evaluating IDSs that are used to monitor *hypercall* attacks. The lack of publicly available attack scripts that demonstrate hypercall attacks was stated as one of the challenges and the paper presents *hInjector*, a tool for injecting hypercall attacks and generating a traffic trace. The proposed method applies to monitoring devices looking for events that are related to hypercalls and functionally similar to hypercalls. The method could be applied to do monitoring for IaaS providers, since the hypervisor is monitored by the provider but it does not provide a way to include tenants in the process. In addition, the range of attacks addressed by this method is limited.

The authors in [80] presented an evaluation mechanism for the security of web applications. The method consists of *injecting real vulnerabilities* and carefully *exploiting* those vulnerabilities in a controlled manner by injecting attacks. However, the vulnerability injection phase requires source code

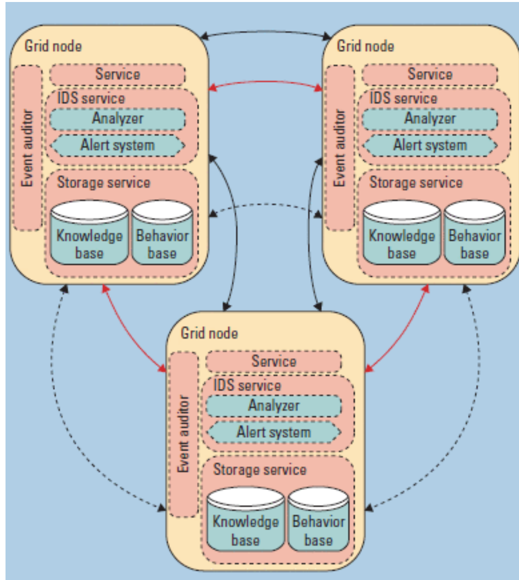


Figure 2.11: GCCIDS architecture from [216]

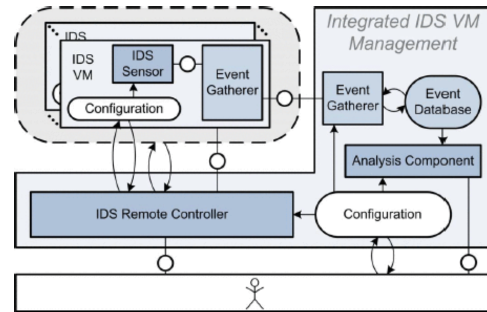


Figure 2.12: IDS management architecture [189]

analysis and creating a modified version of the application (code mutation) which may not be always feasible.

2.4.5 Security Monitoring in Clouds

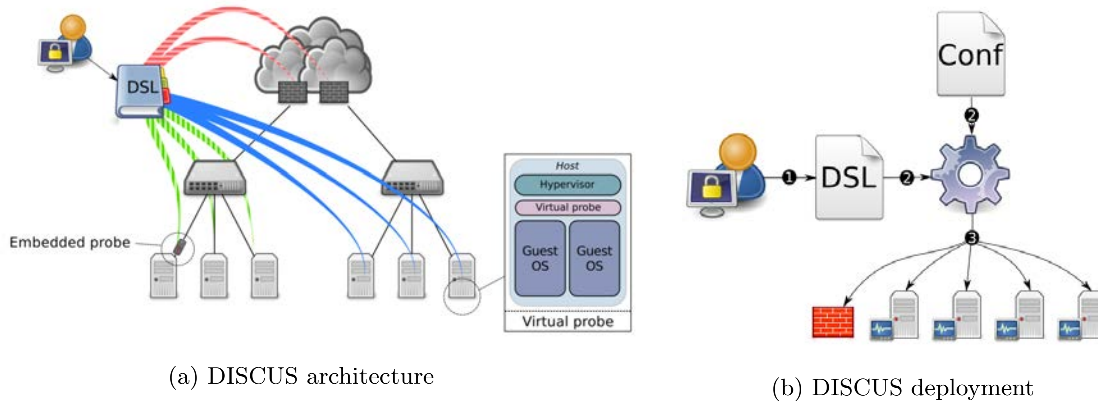
In this section we present security monitoring tools and frameworks that are used by tenants and providers in the cloud environment to perform security monitoring tasks. Some traditional monitoring tools (which existed before the adoption of clouds) can be used to perform security monitoring tasks in clouds but to address new threats occurring in virtualized environments some specific IDSs have been designed. We can categorize types of IDSs used in the cloud into four groups. Host-based IDS (HIDS), Network-based IDS (NIDS), Hypervisor-based IDS (HyIDS) and Distributed IDS (DIDS).

Host-based IDS (HIDS) in Clouds

An HIDS, as described in Section 2.4.3, monitors events in the host where it is configured. In the cloud HIDS can be placed on VMs, host machines or hypervisors and it can monitor events in the corresponding system including configurations, logs, network traffic, user actions and so on. Vieira et al [216] proposed an HIDS service called *Grid and Cloud Computing IDS (GCCIDS)* offered in the middleware layer to cover attacks that network and host-based systems cannot detect. As shown in Figure 2.11 each node contains an IDS used as a local detector and information is shared between nodes when an intrusion is detected. It combines both signature and anomaly-based IDSs to leverage the advantages from both. GCCIDS does not detect all kinds of attacks as it is designed to complement NIDS and HIDS hence, it cannot be used to replace NIDS or HIDS.

Network-based IDS (NIDS) in Clouds

NIDSs also presented in Section 2.4.3, monitor events in the network. With the aim to direct the traffic through the NIDS, NIDSs are usually placed between firewalls and various hosts of the network. Roschke et al [189] proposed an IDS management architecture designed for clouds, which consists of several sensors and a central management unit. Figure 2.12 shows the architecture presented in the paper. The management unit consists of (i) an *event gatherer* to collect events flagged by sensors (ii) an *event database* to store information about collected events (iii) an *analysis component* to access event database and analyze events and (iv) a *remote controller* responsible for remote configuration and control of all connected IDS Sensors. The idea of collecting events from different hosts to perform analysis is also a characteristic of SIEM tools as described in Section 2.4.2. The method is easy to scale, i.e new IDS sensors can be added easily. It also gives some control to users through the *remote controller* component to configure NIDS sensors.



(a) DISCUS architecture

(b) DISCUS deployment

Figure 2.13: DISCUS architecture and DSL deployment as presented in [187]

NIDSs are specifically helpful to detect attacks which abuse the network like denial of service and Address Resolution Protocol (ARP) spoofing attacks. The authors in [137] presented an ARP spoofing detection mechanism which uses the Bayesian theorem to calculate the probability of a VM being an attacker. The method is used to address the internal ARP attack issue of cloud centers with the application of SDN technology. The authors identified four basic ARP attack features and probabilistic computation is performed to detect the occurrence of those features. Information about ARP frames is collected from the SDN controller and the information is analyzed to look for defined features, hence to detect attackers.

Mazzariello et.al [140] presented practical implementation experience of different NIDS deployment strategies in clouds. The paper describes the pros and cons of having a single NIDS on strategic location in the network and setting multiple NIDS to monitor sub-part of the network. Obviously, the best strategy depends on different factors including characteristics of the application scenario, administrator's and users' requirements.

Distributed Intrusion Detection Systems (DIDS)

DIDS refers to a combination of multiple IDSs to monitor large systems. A DIDS has an advantage by using both NIDS and HIDS which can be either signature or anomaly-based. DIDS correlates alerts from different sources to be able to understand events in the system. Dastjerdi et.al [64] proposed a scalable, flexible and cost effective method to detect intrusion for cloud applications regardless of their location using mobile agents. When a central controller receives an alert, a Mobile Agent (MA) is sent to collect evidences of an attack from all the attacked VMs. The MA sends back information to the central controller and further investigation is done to identify real intrusions. However, the method introduces a high network load when the number of VMs increases.

Riquet et.al [187] proposed *DISCUS*, a distributed IDS architecture using heterogeneous probes. Physical and virtual probes are scattered across the network to cooperatively detect intrusion. The paper recommends for probes to be as close to the host as possible, physical probes behind physical hosts or integrated with switches and virtual probes integrated into physical hosts. Virtual probes could be based on existing security solutions like Snort [13]. For administrating the probes, *Discus Script*, a domain specific language (DSL) is proposed. An administrator writes global security rules and a compiler is responsible for generating binaries for every type of security probe in the infrastructure. Figure 2.13 shows the architecture and DSL deployment. From a user-centric point of view the method doesn't separate entities and their responsibilities i.e. it doesn't state what actions should be done by the users or providers. In addition, the proposed DSL is very low level to describe users requirements. It is used to express detection algorithms on IDSs.

Hypervisor-based Intrusion Detection Systems (HyIDSs)

HyIDSs are detection mechanisms in clouds using information from the hypervisor. *Hypervisors* as described in Section 2.2 are software components that provide virtual instances of hardware and facilitate communication between the virtual and hardware layers. HyIDSs offer few advantages over other

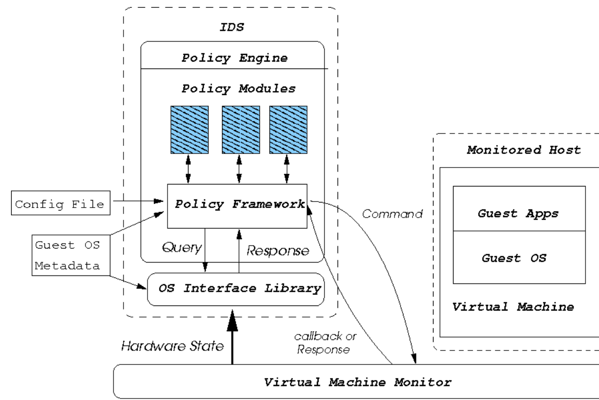


Figure 2.14: Virtual Machine Introspection (VMI) architecture [89]

techniques. First they take the monitoring probe outside the VM, which removes the risk of being compromised if the VM is compromised and for users it removes the burden of installing additional software inside VMs. Second, HyIDSs complement both NIDS and HIDS on the coverage of attack types.

HyIDSs use information such as packets transmitted/received, block device read/write requests and CPU utilization. These parameters were used to build an HyIDS in [158]. The paper proposes an HyIDS which contains three components (i) an *end point* to gather data about running VMs from the hypervisor and present the data to the controller (ii) a *controller* responsible for near real time analysis of performance data (iii) a *notification service* which provides a notification when an attack is identified. Although the method provides an HyIDS solution, the range of attacks (e.g. application level attacks) that can be detected is very limited as it uses little information.

Virtual Machine Introspection (VMI) is one of the methods to implement HyIDSs. It is an approach for inspecting a virtual machine from the outside for the purpose of analyzing the software running inside it. Garfinkel et.al [89] proposed *Livewire*, a VMI-based IDS. It leverages three properties of hypervisors namely *isolation, inspection and interposition.* The architecture of Livewire is shown in Figure 2.14. It is divided into two parts, (i) the *OS interface library* which provides an OS-level view of the virtual machine's state and (ii) the *policy engine* which executes IDS policies by using the OS interface library and the VMM interface. Livewire performs passive monitoring, i.e it does not interrupt events to take action. On the other hand it helps to perform monitoring without the need to add hooks in the hypervisor.

Ibrahim et.al [108] presented CloudSec, a VMI-based monitoring appliance that provides active, transparent, and real-time security monitoring for VMs in the cloud platform. CloudSec is different from Livewire in a way that Livewire works in offline (it can't detect events in the cloud e.g VM migration) while CloudSec supports live migration as it is running on both the source and target hosts listening to the same address and port number. In addition, Livewire works passively while CloudSec is active. CloudSec monitors volatile memory to detect and prevent the kernel data rootkits. When the VM boots CloudSec identifies the memory layout of the VMs hardware by inspecting the control registers of the VMs CPU. Afterwards, CloudSec requests for Kernel Structure Definition (KSD) through the hypervisor. Then CloudSec attempts to construct the view of the live VM by mapping physical memory bytes to the KSD. Then, the constructed view is passed to the defense module in order to perform detection.

Summary

In this section we presented the state of the art in security monitoring. We started by defining security monitoring and listing probes that are used to perform security monitoring. We have learned that security monitoring is a process rather than a goal by itself. Different probes like antivirus, firewalls and IDSs assist the process to reach the desired goal.

We detailed IDSs as they are one of the main monitoring probes and the work described in this thesis directly applies to it. Different metrics that are used to measure the performance of an IDS are also described in this section. We point out the need for having a single unified metrics which can take the base rate parameter as an input. We studied a metric called C_{ID}, which satisfies both the required properties. Finally, we presented different approaches to perform security monitoring in clouds. Specifically, we showed different types of IDSs that are used by tenants and providers in the cloud to perform security monitoring tasks.

2.5 Cloud Service Level Agreement (SLA)

In this section we describe *cloud service level agreements* or *SLA* for short. First a definition an SLA and its characteristics are presented, then its components and life-cycle are described. An example SLA is shown to explain the SLA life-cycle in detail. We then describe the languages used to define SLAs. The languages described below are the background for the discussion in Chapter 4 where we extend a language, named CSLA, to describe the performance of security monitoring devices in SLAs. Finally, as we are interested in security, we conclude by describing the status of cloud security in SLAs.

2.5.1 SLA and its Characteristics

Before the introduction of cloud computing, organizations used to host their own computing resources (networks, servers, storage, applications, and services). By using clouds, tenants benefit from cost reduction (in both building and management), an elastic infrastructure and broad network access. One of the risks of moving to a public cloud is losing full control of the information system infrastructure. The service provider is in charge of monitoring the physical infrastructure and providing the required service to tenants. It monitors every aspect from resource allocation, performance monitoring to security. This creates a trust issue between providers and tenants. Providers acknowledge the issue and endeavor to provide assurance through an agreement called *service level agreement.*

Definition 2.8. A Service Level Agreement (SLA) is a negotiated document between the service provider and tenant. The document describes the provided service, the rights and obligations of all participants in the agreement and state penalties for when the specified terms are not respected. In addition, it contains a quantitative description of the targeted quality of service using Key Performance Indicators (KPI).

Such types of agreement have been used long before the cloud, for example for web services [133]. The main difference between the traditional SLA and cloud SLA is that the former uses the subscription/license-based model (fixed price) whereas the latter uses the pay-per use model (dynamic price). SLAs have become important components in the transformation of a product-oriented economy into a service-oriented economy [222].

SLAs help in building a trustworthy relationship between providers and tenants. It provides a guarantee for tenants, as it includes reward (as penalty) for cases of violation. If there are situations or cases where term(s) can not be fulfilled, it should be explicitly stated. An SLA is negotiated, hence users can specify their needs and requirements to get a service well tailored to their needs.

In addition to being a legal document for the provided service, an SLA is a key component in the process of an autonomic system [106]. Such systems make decisions based on knowledge acquired by analyzing data from different inputs. SLAs provide an input in an autonomic computing decision making process by providing a high level human-based goals, e.g. business goals.

Cloud SLAs are characterized by some features including *Unambiguous, Measurable, Attainable and Verifiable.*

- *Unambiguous*: The definition of an SLA should be clear and not open for multiple interpretations.
- *Measurable*: The expected quality of service should be described with quantitative measurement (KPI).
- *Attainable*: As SLAs are legally binding, service providers agree on achievable objectives. Otherwise, penalties could follow for any possible violation. Having defined KPIs is SLAs is useless if the terms cannot be enforced.
- *Verifiable*: an SLA being an agreement between multiple parties any party has the right to check for the satisfaction of the terms and for violation by other party.

SLAs are composed of different components, which are presented in the next section.

2.5.2 SLA Components

In this section we describe components of cloud SLAs. SLAs differ between service providers and the objectives called *service level objectives* also differ by service model (SaaS, PaaS, IaaS) and deployment model (Public, Private, Hybrid). There are no globally agreed sets of components for SLAs but we can state that to satisfy the basic characteristics of an SLA it requires the following four main components.

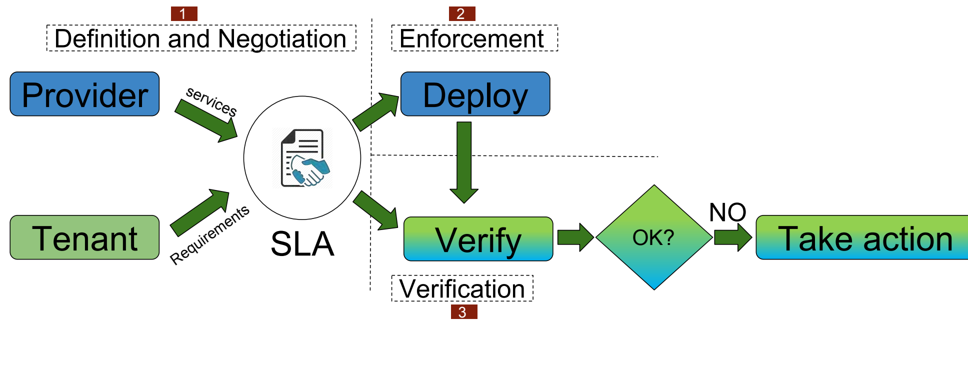


Figure 2.15: Phases of SLA life-cycle

Service Description

The service description component describes the service which is covered under the SLA. It can also describe the time period until which the agreement is valid. The description should be provided in a clear and non ambiguous way to have a common understanding between providers and tenants. Having a formal language to describe SLA terms would give additional helps, like it helps to compare services from different providers and helps to automate the negotiation and enforcement process. Examples of formal SLA description languages include WS-Agreement [17] and WSLA [133]. More details are presented in Section 2.5.4.

Service Level Objectives (SLO)

SLOs describe the targeted quality of service using Key Performance Indicators (KPI) metrics. There are different metrics depending on the service model. The IaaS model contains metrics about compute instances and networking while a database service PaaS includes metrics related to data.

In addition, if some metrics are computed from basic metrics, the computation mechanism should be stated. Moreover, if the frequency of measuring the metrics is an important factor, it should explicitly be stated. If there are exceptions where the SLO cannot be achieved it needs to be precisely described.

Parties

Parties are the entities that will be bound by the agreement, namely cloud providers, tenants and possibly third party entities. The SLA should clearly describe the roles and responsibilities of each participants.

Penalties

Penalties are remedies for cases of SLA terms violations. Violation happens when an SLA commitment is not met, and a penalty or credit can be issued to compensate for the breach of an SLA commitment. Usually penalties are service credits given back to the tenant or applied towards future payments. It can be seen as a counter-incentive for providers to violate an SLA.

2.5.3 Phases of an SLA Life-Cycle

In the previous section we presented the most common components of cloud SLAs. This section presents the life-cycle of an SLA. At the end of this section we provide an example to illustrate the different phases of the SLA life-cycle.

As in the case of SLA components, there are no globally accepted phases in an SLA life-cycle [65, 203, 17]. Alessandra et.al [65] presented five SLA life-cycle phases, namely *negotiation, implementation, monitoring, remediation and renegotiation*. In [203] seven phases of SLA life-cycle are presented in correlation with three cloud service life-cycles. The phases in both [65] and [203] are different classifications of the same process. Below we describe the most common phases of an SLA life-cycle. The phases of an SLA life-cycle are depicted in Figure 2.15.

SLA Definition and Negotiation

The SLA *definition* is a pre-negotiation phase where service providers draft SLA templates according to the services they provide. They should clearly specify the SLOs with clear KPIs. The draft SLA is proposed when a potential client requests a service. The *negotiation* process continues from here. In the negotiation phase, tenants identify their requirements and perform assessments. In [179] the Cloud Standard Customer Council (CSCC) proposed a practical guide containing ten steps for evaluating different SLA offers based on tenants requirements.

If a tenant is not satisfied with the current offer she/he can decline the offer and request an updated SLA offer. In the case of multilevel SLAs (a multilevel SLA is split into different levels, each addressing a different set of customers for the same services, in the same SLA) there could be rounds of negotiation. But usually the negotiation step is missing in simple public cloud SLAs. A simple public cloud SLA is signed when a single user is getting service from a provider, such providers follow a "take it or leave it" policy without any negotiation process.

SLA Enforcement

Once the SLA is defined and agreed, the next step is to enforce the terms that are defined in the SLA. Enforcement means to implement what is written in the agreement into the actual infrastructure. It ranges from the start of setting up services to configuring mechanisms used for checking the fulfillment of expected quality. In public clouds the underlying infrastructure is managed by providers hence, the service provider is responsible for deploying the requested infrastructure while respecting the SLA and towards achieving SLOs in the agreement.

SLA Verification

Once the infrastructure is deployed, it should be monitored regularly (or with the frequency stated in the agreement) to verify its compliance with the SLOs promised in the SLA. Checking the fulfillment of an SLO at specific times is verification, while continuously checking is called monitoring. The SLA verification can be performed either by providers, tenants or a third party entity. Transparency is important if providers do the verification. When an SLA is violated tenants have to react in order to initiate the next step according to the SLA definition. The response for violation could be remediation, renegotiation or termination.

To explain the life-cycle phases with an example, we present the *availability* SLA term offered by Amazon Compute Services [5]. The service commitment reads as follows:

Example 2.9 (Amazon compute services SLA)**.** *"AWS will use commercially reasonable efforts to make the Included Products and Services each available with a Monthly Uptime Percentage (defined below) of at least 99.99%, in each case during any monthly billing cycle (the "Service Commitment"). In the event any of the Included Products and Services do not meet the Service Commitment, you will be eligible to receive a Service Credit as described in Table 2.3 "*

Monthly Uptime Percentage	**Service Credit Percentage**
Less than 99.99% but $\geq$ 99.0%	10%
Less than 99.0%	30%

Table 2.3: Amazon service commitments and service credits

Example 2.9 describes the *SLA definition* between two parties, a *user* and the *Amazon Web Service* provider. But the example presents only the objective section (SLO). Other components like "included service and products", "definitions of terms", "SLA exclusion" ... can be found in [5] which describes every term without any ambiguity. The *Enforcement* of this SLA term is accomplished by deploying the services (i.e. instantiating the requested VMs, containers or block storage) at the appropriate time and on the requested location.

Verification can be performed by both the provider and tenant. In our example verification is checking the availability (up-time) of the service or product. Violation happens if the sum of total down-time (as it is defined in [5]) is greater than 0.01% (4.32 minutes per month). In that case clients need to start a *credit request and payment procedure* to get compensation for violation, i.e. penalty will be paid by the service provider.

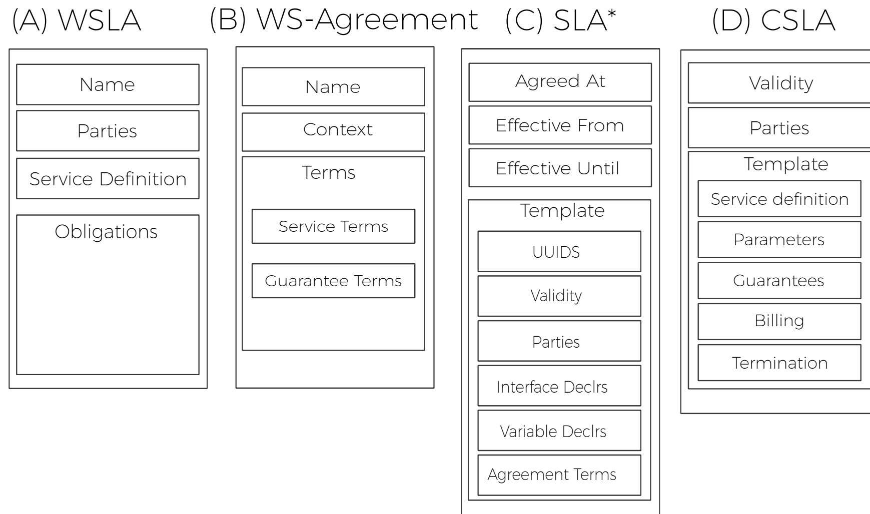


Figure 2.16: Structure of an SLA in different languages

As described in the next section, leaving the burden of violation detection to tenants is one of the shortcomings of current existing SLAs.

2.5.4 SLA Standards and Languages

In this section we describe standards and languages that are proposed to formalize the definition of SLAs. The advantage of formalizing the SLA definition is many folds including to automate the life-cycle of SLA, to pass SLA as an input for an autonomic system and comparability i.e comparing offers from different providers. Below we describe widely used SLA languages.

Web Service Level Agreement (WSLA)

The WSLA [133] language was proposed as part of the WSLA framework by IBM for the definition and monitoring of SLAs in web services. The WSLA language is based on XML and is defined as an XML schema. Figure 2.16(A) shows the structure of an SLA in WSLA. An SLA in the WSLA language contains three parts:

- *Parties*: a description of participants, their role (provider, customer, third party) and the action interface they expose to others
- *Service description*: defines one or more services and service objects have parameters which show how they are measured,
- *Obligations*: which contain service level objectives and actions to be performed in case of SLA violation.

However, the management of penalties is limited mainly to notification in case of violation. The latest semi-stable version of WSLA was published in 2003, since then it is integrated into *WS-Agreement.*

Web Services Agreement (WS-Agreement)

A WS-Agreement [17] is an SLA specification presented by the Grid Resource Allocation and Agreement Protocol (GRAAP) Working Group of the Compute Area of the Open Grid Forum [12]. The objective of the WS-Agreement specification is to define a language and a protocol for advertising the capabilities of service providers and creating agreements based on creational offers, and for monitoring agreement compliance at runtime. It is also an XML based language. WS-Agreement introduces the concept of pre-defined agreement template containing customizable aspects and creation constraints.

The structure of a WS-Agreement SLA is shown in Figure 2.16(B). It contains *Name and Context*, providing information about the agreement document (e.g. agreement initiator, expiration time ...) and a

reference to a *Template* containing information about the service provided and service levels which should be guaranteed. WS-Agreement is generic enough to be extended for domain specific agreements and it contains more information about the service functionality than WSLA. However, the negotiation process in WS-Agreement is not significantly versatile, it only offers the "accept or reject" model. Moreover, the WS-Agreement semantics is not defined precisely. To overcome this issue Semantic WS-Agreement Partner Selection (SWAPS) [160] presented an extension. It adds *tags* into the original WS-Agreement to allow the incorporation of semantics. For example the original WS-Agreement defines a response time SLO as

$$\text{duration} < 3 \text{ seconds} \tag{2.10}$$

In the SWAPS Schema this is expressed as

$$\text{predicate type} = \text{`}less\text{`},\ \text{parameter} = duration,\ \text{OntConcept} = qos :: responseTime,\ \text{value} = 3,\ \text{unit} = second \tag{2.11}$$

SLAng

SLAng [124] is a language for defining Service Level Agreements (SLAs) that describe different layers of services (network services, storage services and middleware services). In [124] the authors present a reference model for inter-organizational service provision at storage, network, middleware and application levels. The SLAng syntax is defined using an XML schema. In addition to contractual statements and service level specification it includes an end-point description of the contractors (e.g., information on customer/provider location and facilities).

However, SLAng focuses on a specific group of applications (it assumes the use of component oriented middleware). Hence, its applicability even for the general web services is limited. In addition, Skene pointed out in [202] that the definition of the language was highly imprecise and open to interpretation. He proposed additional features in SLAng. He presented a specification which is based on the Essential Meta-Object Facility (EMOF) and Object Constraint Language (OCL) [202] standards. But using the method requires technical expertise, which leads to difficulties for users and high costs for its adoption.

All the above languages were not designed considering the cloud, although WS-Agreement with different extensions have been used in many European projects to describe SLAs for clouds (e.g SPECS [199], Contrail [153] and Paasage [151]). Other languages have been developed specifically for cloud services. Moreover, some features (like pre-generating templates) have been adapted for cloud scenarios. In th following we present notable SLA languages that are designed for the cloud.

SLA*

SLA* [222] was developed as part of the FP7 ICT Integrated Project SLA@SOI. It is a domain-independent, abstract syntax for machine-readable SLAs and SLA templates. It is inspired by WSLA and WS-Agreement, but it is different from them in the fact that SLA* is language independent (it does not use XML). It is not specific to web services, as it deals with services in general. It supports the formulation of SLAs in any language for any service. SLA* can be seen as a syntax because it serves to specify the organization of symbols in an SLA document, but it is abstract in that it leaves unspecified the particular symbols used to instantiate this organization.

The structure of SLA defined using SLA* is shown in Figure 2.16 (C). SLA* providers define SLA templates from which the SLA as well as additional attributes like the time at which the SLA was agreed and its effective lifespan are derived. The SLA template comprises information like parties, services described in terms of their functional interfaces and agreement terms including SLOs. The multi-domain approach in SLA* introduces some drawbacks. In particular for each domain it requires developing symbols. For example to use SLA* for cloud SLA description, it requires defining symbols for every component like participants, services, parameters .... Moreover, there is no general precise semantics related to SLA*, creating difficulties to automate SLAs.

SLAC

SLAC [214] is a domain-specific SLA language for clouds. The authors argue that the existing languages do not provide a formal semantics description and also cannot cope with the distinctive characteristics of clouds, specifically with the business aspects of SLAs (e.g pricing model). The main components of SLAC are (i) the description of the contract containing at least two parties (ii) the terms of the agreement

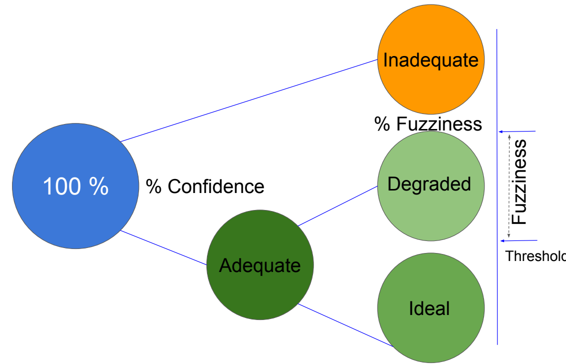


Figure 2.17: Fuzziness and Confidence ratio graphical representation

to express the service and SLOs, (iii) the guarantees which describe penalties in cases of violation. It also presents an extension to capture the business aspects of an SLA with a method adopted from [116]. It divides the SLA into three phases: (i) Information Phase, in which the details about the services, consumers and providers are browsed and collected; (ii) Agreement Phase, in which the participants negotiate and define the terms and the pricing model; and (iii) the Settlement Phase which is related to the evaluation and enforcement of the SLA.

SLAC comes with an open source management framework. A formal semantics description of the language is presented in the [214]. The semantics of an SLA is formulated as a Constraint Satisfaction Problem (CSP) that satisfies: (i) at negotiation-time, whether the terms composing the agreement are consistent; and (ii) at enforcement-time, whether the characteristics of the service are within the specified values.

In SLAC the metrics and the way to measure them are pre-defined in the language. Though, the authors claim it can be extended according to the needs of the involved parties. While defining SLA using SLAC, guarantee definitions are optional. Hence, it may lead to an SLA which do not include penalty for violation. In addition, in cases where a guarantee is defined the actions when violating term(s) are restricted to either notification or renegotiation.

CSLA

CSLA [123], a language to express SLA in the context of clouds, addresses cloud instability to better control SLA management (in particular SLA violations) and cloud scalability. Service providers offer different qualities of service using a pay-per use model (as described in Section 2.1.3, e.g. with Amazon Spot instances), CSLA allows coping with error rates in SLAs to enable a service provider to continue operating properly in the case of some violation.

Figure 2.16 (D) shows the structure of SLAs in CSLA. The definition has been influenced by WSLA [133] and SLA* [222]. SLAs in CSLA contain three main parts (i) a reference to the template used to create the agreement, (ii) validity of the agreement, and (iii) parties involved in the agreement. The template is made up of five parts namely: cloud services definition, parameters, guarantees, billing and termination.

CSLA, in addition to the traditional fixed price billing model, allows the possibility to use the pay-as-you-go model, which is one of the basic characteristics of clouds. Additionally, CSLA introduces novel properties for SLA metrics called *fuzziness and confidence* ratios. The former defines the acceptable margin around the threshold value of an SLO parameter, whereas the latter defines the percentage of compliance of SLO clauses.

Figure 2.17 shows the three states of a request with respect to SLO violation to illustrate fuzziness and confidence. To describe them with an example, let us assume that an SLO for the response time of a service is 2 seconds, the fuzziness value is 0.5 and the confidence is 95%. This means out of 100 requests at least 95 of them should have a response time between 0 and 2 and maximum 5 can be between 2 and 2.5 without violating the SLA. This is an interesting property to have in the cloud. Indeed, as the environment is very dynamic respecting a fixed SLO for every fraction of seconds may not be feasible. Specially assuming complex metrics measuring different properties of the cloud (e.g performance of a security monitoring device), achieving a fixed target SLO is challenging.

The next section focuses on a specific type of SLAs, *Security SLAs*.

2.5.5 Security SLAs

In the previous section we have presented SLAs and languages that are used to describe SLAs in clouds and web services. In this section we focus on types of SLAs covering security aspects of the cloud. The challenge of including security services in SLAs has been a subject of study before the advent of cloud [103].

Henning [103] stated one approach to incorporate security management services into enterprise information services and the proposed economic model is through the definition of SLA. The paper pointed out that security management activities have not been quantified or expressed as SLAs. The paper presents a three-step process to generate a preliminary group of SLAs (i) policy analysis: analyzing regulations and standards (ii) architecture analysis: policy are validated against the customer's network and architecture diagrams (iii) on-site interviews : designed to solicit additional details about representative site implementations. However, other than defining properties the paper falls short of quantifying security properties and of how to enforce and monitor SLAs.

In IaaS clouds tenants have no control on the physical infrastructure of the system, hence relying on providers for securing services. This is one of the main sources of concern in the cloud service models and it augments the importance of assurance for security in the cloud. In [186, 44] the authors applied the method proposed in [103] to define metrics to be used in such contracts. In [44] a list of initial guides to start the process is presented. Metrics like password management (how often passwords should be changed), backup policies, and repair time are described. The authors also mentioned the importance of having measurable and quantifiable metrics in SLAs.

Bernsmed et.al [28] discussed the need to include security in SLAs and described security in SLAs as *Quality of Protection (QoP)*, which comprises the ability of a service provider to deliver service according to a set of specific security requirements. QoP is described with one or more security mechanisms. Five groups of security mechanisms are presented namely, Secure Resource Pooling (RP), Secure Elasticity (E), Access Control (AC), Audit, Verification & Compliance (AU), and Incident Management & Response (IM). In another paper [29] a similar method is presented for composing security SLAs for federated cloud services.

These works are very first steps but they lack quantitative descriptions of security services, enforcement mechanisms, and verification techniques.

To address the issue of verifiability, Da Silva et.al [61] proposed an approach to build a Security-SLA. They used two classes of metrics to define security SLAs (i) infrastructure metrics: containing hardware devices and related cloud software, such as networking, firewalls, routers, proxies, operating systems etc (ii) services metrics: consisting of SaaS, PaaS or IaaS hardware/software components that provide the service contracted by the tenant. Monitoring agents run both inside and outside the tenant environment to collect security metrics. Collected metrics are mapped to scale values from [0-4] representing the range of security values as [Critical, High, Medium, Normal, Zero], a technique used also in [60] to facilitate operations. However, the proposal did not consider all critical security aspects and services in the cloud. It is not clear which metrics are being collected and how the critical level is assigned. For example, how are the metrics from different types of monitoring device mapped to [0-4]. Moreover, other than stating monitoring agents in different locations of the cloud, it is not clear how the monitoring is done i.e. who is responsible on different service modes, what are the measurement procedures ...

In [175] the authors studied conceptual and technical barriers to implement a Sec-SLA monitoring service. Some barriers include the lack of acceptance and maturity of SLA management systems, the difficulty of mapping high level security properties to low level monitoring parameters, the lack of deployment-layer agnosticism, and extra complexity introduced by virtualization. This study was used as an input for SPECS [184].

SPECS is a European project which intends to explore a user-centric framework and a platform dedicated to offer Security-as-a-Service using an SLA-based approach. The assumption that "security features should be applied not starting from what the provider offers, but from user needs" makes it more user-centric. An SLA-based approach is used to have a common language between providers and users. The SPECS project focused on securing cloud resource provisioning using SLA management [39]. It extended WS-Agreement to describe security SLAs [40]. The core of the SPECS framework consists of three components, *negotiation, enforcement and monitoring modules* as shown in Figure 2.18. Figure 2.19 shows the enforcement process. Users specify security requirements using provided templates. The desired template is passed to the planning component which is responsible for evaluating the feasibility and

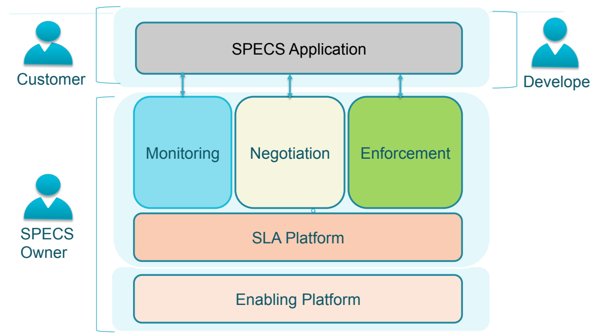


Figure 2.18: SPECS framework [199]

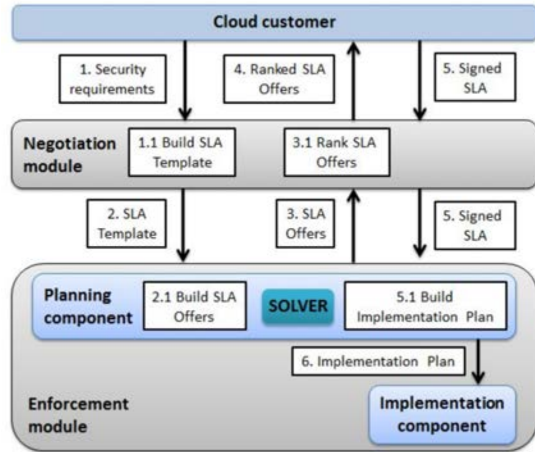


Figure 2.19: SPECS enforcement process [199]

building plan for the implementation of requested service. The feasible plan is generated by formulating an optimization problem according to the template SLA. The optimization is solved targeting to find the minimum number of resources required to implement the SLA. If there is no solution satisfying the constraints then the template is dropped. Once a plan is generated, a final SLA offer is proposed for client. Upon return of a signed SLA the implementation component deploys the requested service.

Kaaniche et.al [114] presented security SLA monitoring mechanisms in clouds. The paper extends a language called rSLA [132] to support the description of security requirements. The rSLA runtime service is also extended, named Sec-rSLA, to incorporate the language extension. Figure 2.20 shows an overview concept of Sec-rSLA. A service (S) in the cloud is represented by set of elements, a subset of these elements which are potential targets of cyber-attacks are Vulnerable Elements (VE). A VE has to be evaluated with respect to a set of security properties (SPs) where SPs are based on measurable metrics. SLOs belong to a Security Class (SC) which is the result of aggregated SPs.

However, both SPECS and Sec-rSLA don't use quantifiable metrics describing the security of tenants' environments. Use cases presented in [39, 40] use security policy constraints as an input for building SLAs (e.g the load balancer should not be collocated with a server or the IDS needs exclusive use of machines ...). Similarly, Sec-rSLA uses policies like the number of replicas for a given data as SLO in security SLAs. This is mostly as a result of difficulties to quantitatively measure security.

Emeakaroha et.al. [74] presented the LoM2HiS framework, which is used to map low level monitored metrics to high level SLA parameters. Figure 2.21 shows the LoM2HiS framework, service providers define mapping rules using a DSL which are used by a run-time monitoring tool to map low level raw metrics and form an equivalent of the agreed SLA objectives. However, the focuses of LoM2HiS is not

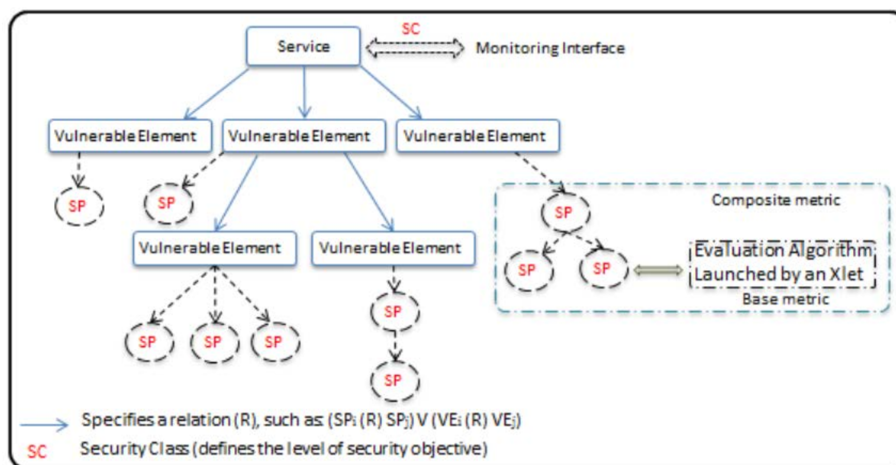


Figure 2.20: Sec-rSLA concepts overview [114]

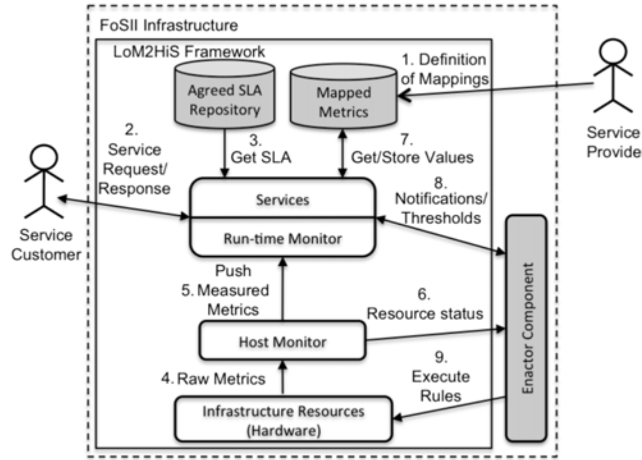


Figure 2.21: LoM2HiS framework architecture [74]

security metrics. It is also difficult to adapt LoM2HiS for security as it requires defining *mapping rules* from low level metrics to high level SLAs, which is difficult to do for security metrics. In addition LoM2HiS is defined in a specific framework.

Currently major IaaS cloud providers give a textual version of SLAs addressing mainly availability of the provided service, which is one of the three security triads (see Section 2.3.2). However, guaranteeing only availability does not address the major concerns of tenants (confidentiality and integrity). Even for the availability SLAs as it is today, there are major drawbacks that we describe in the next Chapter.

2.6 Summary

In this chapter we presented state of the art, which is used as a basis for the next chapters. First, we described clouds with their characteristics, deployment and service models and revised the main technology deriving cloud e.g virtualization. Two types of virtualization namely machine and network virtualization are discussed in detail. We devoted a section to information system security which describes, not exclusively, security mechanisms and threats as a result of virtualization.

We presented in detail security monitoring and related subjects, like tools to perform security monitoring. As the thesis work is directly applicable on NIDSs, we dived in detail describing on how to measure the performance to NIDSs. We also described the current state of security monitoring in clouds. Last but not least we presented Service Level Agreements (SLAs) in clouds. Their characteristics and components were described. Additionally, different languages used to define SLAs are discussed. Finally, efforts to include security in cloud SLAs were covered.

Even if there are lots of efforts to improve SLAs in clouds, the current state of SLAs falls short of some aspects. Notable shortcomings include:

- There is no practically used common language or high level standard that can be used to define SLAs from different service providers. Almost all currently available SLAs are described using natural language. As described in Section 2.5.4, different languages are proposed in the research community but none of them has emerged as a default (generally acceptable) standard to describe SLAs. We think that this could be a combined result of technical difficulties (e.g. correctly mapping users' requirements that are described using high level language into low level configurations) and behavioral and economic factors (e.g a service with more custom requirements would probably cost more than a general service), hence users prioritizing economic factor.

- Almost all existing SLAs cover *availability* of the provided service. As shown in [24] almost no cloud providers offer any performance guarantees for compute service. It should come with no surprise that other aspects like security are not included in SLAs. This problem exacerbates when assuming the fact that security is the most concerning issues for most cloud tenants.

- Currently detecting SLA violation is left as a responsibility for tenants. This problem is the result of not having an automated monitoring system for SLAs. SLAs are defined using natural languages

with detailed definition as possible hence processing natural language to automate the life-cycle of SLA is not feasible. It should be noted that there are third party entities providing SLA monitoring service like Cloudstatus [53] (e.g monitoring virtual infrastructure for detecting down-time).

- Most public IaaS providers offer static, standard SLAs, i.e there is no negotiation process. Tenants have no option other than agreeing if they need the service ("take it or leave it" model).

In our work we want to include users in the process of security monitoring by participating through an SLA. SLA by its definition is an agreement between providers and clients. If we can include security monitoring terms in an SLA users will be able to participate in the life-cycle of such SLAs, i.e users will be able:

- To specify their needs and requirements for monitoring the security of their services;
- To participate in the process of enforcement, e.g if the enforcement requires some action from the tenant side;
- To perform continuous monitoring, check the respect of SLOs and be compensated for any violation.

In the next chapters we will present the contributions of our study. Chapter 3 presents a high level description of how to include users in the security monitoring process through SLAs. It lists the challenges and describes how we address some of them and the ones not addressed in our work. It also gives a detailed presentation on how the SLA life-cycle described in Section 2.5.3 can be applied for security monitoring terms.

Chapter 4 presents the first step in the process of security monitoring SLA life-cycle, i.e the definition of security monitoring SLAs taking users' security needs into account. It focuses on a specific security monitoring probe, NIDS. It presents an example SLA definition and mechanism to estimate performance of NIDS in order to prepare SLA templates. In Chapter 5 a technique is presented to perform verification for SLOs defined as described in Chapter 4. The verification is performed by injecting real attacks but without damaging the production infrastructure.

Chapter 6 presents a different technique to include users in the security monitoring process. In the previous chapters, to verify an SLA a tenant and a provider needs to share some information, like tenants need to disclose the software that they are running and providers need to configure the infrastructure to allow verification for tenants. This creates a dependency between each other. In Chapter 6 an SLA monitoring mechanism, based on a distributed algorithm which requires users participation by design is described. In addition, the method removes the dependency between tenants and providers.

Chapter 3

User-Centric Security Monitoring

3.1 Introduction

In the previous chapter, we presented the state of the art on cloud SLAs and specifically on the state of security in SLAs. We saw that *availability* is the most common property covered by current SLAs. Other aspects of security are the targets of few recent research works but they are not included in any of existing SLAs. Moreover, a few of the research outputs follow a user-centric approach [184]. Even those following user-centric approach do not address (i) how users express security requirements (ii) how to offer quantifiable SLO regarding security properties (iii) how users can verify an SLO in their environment. In the context of clouds, where the provided services are tailored for each tenant, SLAs should be user-centric. Especially when addressing the security of tenants' environments, each tenant has its own security requirements and very often they require different implementation mechanisms.

User-centric refers to users having more control, more choices or more flexibility. Providing user-centric security services means allowing tenants to have more control, choices, and flexibility on types of vulnerabilities to be monitored, over actions to be taken in cases of incident and compensation if damages happen as a result of security breaches. All these can be addressed by allowing tenants to participate in the process of securing their environment.

SLAs can be used to moderate interactions between service providers and tenants. In the context of clouds, engagement of two or more parties in the set of action to a common goal can be regulated by SLAs. By definition an SLA associates the participants in the agreement hence, it becomes a perfect candidate to create common security goals and work on that direction for both tenants and providers.

Theoretically finding a common goal may seem easy but in practice achieving such goal from multiple, maybe conflicting requirements, is very difficult. In the context of cloud security, tenants' requirements focus on threats and risks, while cloud service providers express their offers in terms of security mechanisms (tools and software that implement security policies) and their parameters for configuration. Additionally, as described in Section 2.5.5 quantitatively measuring security is difficult. Hence, measuring security properties and enforcing tenants' requirements is not easy. On the other hand security details of a provider is usually complicated and not interesting for tenants.

In our work, we propose to include *security monitoring terms* into SLAs. As described in Section 2.4 security monitoring by itself is not a goal, it is a process to safeguard and achieve desired security properties. Adding security monitoring terms into SLAs means giving guarantee for tenants on the performance of security monitoring tools and devices. Quantifying performance and capabilities of monitoring devices is easier than quantifying security hence it shortens the gap between tenants requirement and providers description. Our approach fulfills the user-centric design property, as users are participants in the life-cycle of SLAs.

To achieve user-centric SLAs which contain security monitoring terms we propose:

- An SLA description language which is an extension of CSLA (see description in Section 2.5.4)
  - It can describe performance of monitoring devices,
  - It can be used to describe vulnerabilities and users can specify requirements in terms of vulnerability;
- SLA monitoring and verification mechanisms that are transparent (non-secretive) for both providers and tenants;

- Secure, trusted and transparent storage which is used to store SLA verification results.

In this chapter we present arguments why user-centric design is required for security monitoring SLAs. In relation with that we present the objectives of our study and the challenges of designing and implementing user-centric security monitoring SLAs. Before diving into the solution on how we include users in the process of security monitoring in clouds, we present the system model where our solution can be applicable. It should also be noted that the proposed solutions in our study do not cover all types of threats, therefore a section is provided to discuss the types of threats that are addressed in our study.

3.2 The Need for User-Centric Security Monitoring in Clouds

In this section we provide our view points on why user-centric design is required for security monitoring SLAs. In addition, the need for cooperation between tenants and providers, objectives of our study and challenges on developing user-centric security monitoring SLAs are presented.

One of the characteristics of clouds which helped to its wider adoption is its flexibility. A tenant can provision whatever amount of resources she/he wants, at any time and pays only for what she/he used. In such a cloud service model tenants deploy different types of service on the acquired infrastructure. It is also very difficult to find one security monitoring solution which can fit the needs of all tenants. Moreover, the multi-tenancy feature of cloud computing, which makes resource utilization very efficient, becomes a bottleneck for security. This is because multi-tenancy creates an environment where an attacker and a target (or two competing companies) can be co-located in the same physical machine, which introduces a new kind of vulnerability.

For example, assume there are two companies, company *A and B. Company A* provides a dynamic web service (including a back-end database and server side programs) and *company B* provides a data storage service. Company B includes an FTP service where port 20 is open while Company A needs port 20 to be closed (e.g there is no service running on that port). Both companies use a cloud infrastructure to run their services and their clients access the service through the Internet. Company A is more interested to be protected from dynamic attacks like SQL injection and cross site scripting while Company B is more interested in keeping the data integrity (not being tampered). Such requirements are not always mutually exclusive, there can be some intersections like both companies are interested on keeping their service available as long as possible.

Even if there can be intersection, the requirement of one tenant is somehow different from others or even conflicting with others. If security requirements are different it may be required to apply different monitoring solutions or the same monitoring method but with different operating configuration depending on the requirement. In the above example the requirements of company A could be addressed by using NIDSs with appropriate detection rules while the requirements of company B can be achieved efficiently with cryptographic techniques. This shows that, since the cloud computing model is user-centric by its nature, security monitoring in such an environment should follow the same design and operating process.

3.2.1 Cooperation Between Providers and Clients

As shown in Section 2.1.3, in different service models tenants manage some portion of the information system tiers. For example in IaaS clouds tenants own virtual resources (e.g virtual server or storage) running on top of the hypervisor. The security of tenants environments is not only dependent on their virtual infrastructure but also on the security of all the underlying components (hypervisor, OS if the hypervisor is running on top of an operating system, physical infrastructure and so on). For example if the underlying OS or hypervisor is vulnerable for side channel attacks as shown in [109, 188] an attacker could get cryptographic information from the target VM regardless of the type of security implemented in the target VM. Even worse, if the underlying hardware is vulnerable to Meltdown [129] and Spectre [121] types of vulnerability the OS would be compromised making both the hypervisor and VMs running on top vulnerable, again regardless of the security mechanism inside VMs. It shows that having only strong security inside a VM is not enough to be sure of its security.

On the other hand service providers cannot perform full security monitoring as they do not have knowledge or access to the services running inside tenants' environments. For example a tenant may provide an email service where port number 25 is open or an FTP service where port 20 is open. Without having knowledge of tenants' environments it is burdensome to perform full security monitoring.

To consolidate these two complementary features and to have an end-to-end security monitoring ability in a given environment there should be *cooperation between providers and tenants.* Such cooperation

will help both providers and tenants to get information that they could not have otherwise. Tenants will share necessary information about services running in their environment (e.g name and version of applications running in the environment). In return providers will give necessary log files and underlying infrastructure information for tenants (e.g. logs or output of monitoring devices). This way an end-to-end security monitoring, where both tenants and providers are participant, can be achieved. Economic incentive will be a drive for service providers and tenants will get better guaranteed service.

3.2.2 Objectives

In our work we aim to give guarantees on the performance of security monitoring devices for tenants in the cloud while those devices are configured to monitor vulnerabilities selected by tenants. Such guarantee follows other cloud service models in a way that it will be tailored per tenant. Tenants will be able to describe what they want, e.g which vulnerabilities to be monitored, accepted level of performance (described with relevant metrics as KPI) and be able to interact with what a provider offers. Tenants should also be able to compare offers from different service providers.

We also aim to provide tenants with tools and methods to check the validity of a given assurance. Such guaranteeing technique where tenants participate in the process of security monitoring for their outsourced information system adds trust, even though they do not have a full control on the physical infrastructure. By giving such a service we also aim to reduce the human component in security monitoring process (on-demand self-service) as much as possible, as it is one of the characteristics of cloud computing (see Section 2.1.1).

,

3.2.3 Challenges

In this section we present challenges for designing, implementing and providing user-centric security monitoring service through SLAs. The challenges result from different factors, including the properties of SLAs, the nature of cloud environments and the difficulties of security monitoring (e.g selecting appropriate mechanisms, measuring performance of monitoring devices, ...). This section presents the challenges that we identified and describes the ones addressed in our work. In Chapter 7 we discuss on how to address the remaining challenges. We identified the following challenges:

1. Lack of method to describe security monitoring requirements and services. To get user-centric security monitoring SLAs, tenants need to describe their needs and providers need to promote their security monitoring services. To automate and easily compare different offers such description should be standardized. Requirements and service descriptions should also be understandable for users (not too much technical) and also realistic for implementation (not in a very high level description).

2. Lack of automated and efficient security policy enforcement mechanisms. This can be related with the previous challenge, lack of standardized description creates a gap for implementation. Before the advent of clouds, in a traditional outsourcing to a managed service provider, a provider might well agree to follow the users' security policy. However, in clouds it is difficult, if not impossible, for providers to comply with all tenants separate security policies, with possibly different, even conflicting, requirements.

3. Measuring the performance of monitoring devices is not straightforward. The performance of such devices not only depends on the configuration and available resources but also on the outside environment where they are operating. Moreover, in most cases the outside factors are nondeterministic events (e.g probability of occurrence of an attack), they are hard to measure or predict. Sharing a monitoring device with multiple tenants also affects the performance. Assuming security monitoring devices in clouds where multiple tenants share the devices, measuring the effect of a single tenant on the performance of the device is an open problem, as there will be the effect of others at the same time.

4. Lack of methodologies to measure the performance in a way that is open for both tenants and service providers. We call this property *transparency*, operation is described in such a way that it is easy for others to see what actions are performed.

5. To perform dynamic evaluation of a given vulnerability, a practical attack (tool to exploit the vulnerability) is required. Usually when a vulnerability is discovered, unless the founder has negative intent, there is no incentive to publish the attack. Specially if the vulnerable product is owned by a commercial entity, there is an incentive not to disclose attacks. Hence, performing dynamic evaluation becomes challenging.

6. Most of the time tenants have no detailed knowledge on their security requirements. Regular software or product users may have knowledge on functional aspects of a system but describing security vulnerabilities requires relatively detailed knowledge. To make an agreement with providers it requires at least to describe vulnerabilities in a way that is understandable for providers.

7. After measurement is performed the result should be stored without any bias. Since violation of a term in an SLA has a consequence, participants get incentivized to tamper the result in their favor. Detecting and preventing such modifications is challenging specially in the cloud where ownership of a system is divided between different parties.

8. The dynamic nature of clouds makes security monitoring challenging. In IaaS clouds VMs are created and deleted frequently. In addition, VMs migrate from one host to others for different purposes (for resource optimization, maintenance ...). The security monitoring service should adapt to dynamic events in the cloud.

In Chapter 4, we address the challenges of describing security monitoring SLAs for specific security monitoring devices, NIDSs (Challenge 1). Our SLA description method expects tenants to describe their requirements in terms of vulnerabilities, which is not always easy as described in the challenges above (Challenge 6). Providing a simpler way to describe security requirements is out of the scope of our work. The metrics that are used in our study take the operation environment into account (Challenge 3). Specifically, the rate of occurrence of an attack is considered.

In Chapter 5, we present a transparent performance evaluation method for NIDSs which can be used by both providers and tenants(Challenge 4). We also present a distributed storage mechanism which can be used as a trusted storage platform in Chapter 6. Such a storage system is used to store security property verification results, specifically data integrity checking results without any bias (Challenge 7).

Our performance evaluation method partially takes into account the dynamic nature of clouds (Challenge 8). Chapter 5 describes the events that are taken into account. We used a very specific SLA enforcement method. We did not address the different enforcement mechanisms for the security monitoring SLAs (Challenge 2); more details on these issues are presented in Chapter 7. In addition, we used a limited set of real attacks for our experiments. Collecting and organizing vulnerable applications and the attacks to exploit those vulnerabilities (Challenge 5) is out of the scope of our work.

In the next section we present the system model and threats that we take into consideration for the contributions presented in the next chapters of this thesis.

3.3 System Model: Outsourced Information System Resource

In this thesis we consider an information system in a cloud where responsibilities are divided between tenants and providers, hence the relationship is bound by an SLA. The cloud service model can be either an IaaS or PaaS with one of the deployment methods listed in Section 2.1.2.

Out of many properties to be monitored in an outsourced information system, we are interested in security aspects. In our study an outsourced resource can be infrastructure (e.g virtual servers) or data. We assume IaaS clouds where providers own the physical infrastructure and tenants own virtual resources, like VMs, internal configurations of VMs and services that are running inside VMs. Providers are responsible to select host machines on which a VM will start (also known as *compute node*) and migrate VMs to other hosts when there is not enough resources. A provider infrastructure consists of multiple compute nodes, grouped into data centers and distributed over different countries. A data center is subjected to regulations in the residing country. Tenants may have an option to choose a country where the resources will be hosted (e.g "Regions" in Amazon web service).

Almost all the time a single host is shared between multiple tenants. Providers are responsible to provision and configure servers and manage isolation between tenants. The same way network physical infrastructure is shared between multiple tenants and providers are responsible to configure the network

connection of the shared environment. Providers use virtual switches (vSwitches) to provide connectivity and use VLAN and tunneling capabilities of the virtual switches to logically divide the network into smaller, isolated networks. There can be different layers of virtual switches e.g vSwitches to divide the network between different tenants or vSwitches dividing network for different VMs. Providers own and configure switches that are used to divide networks between tenants. Tenants manage the network in their environment and they can further divide their environment using different techniques (e.g security groups in Openstack).

In practice two IP addresses are assigned for every VM, one is part of the internal subnet used to communicate with other VMs under the same tenant while the other is used to communicate with the outside world. Tenants can deploy any application on top of VMs and configure the incoming connection as required. Any monitoring device located outside of a VM is managed by service providers while anything configured inside a VM is managed by tenants.

We also consider PaaS clouds in some part of our study where providers own the platform and tenants are responsible for the data processed in the platform. In such systems tenants have no control on the configuration of VMs, providers are responsible for managing the infrastructure where the platform is running. Tenants can perform any operation over the data as long as the operation is supported by the platform. Security monitoring SLAs on such systems address security of the data rather than the system.

3.4 Threat Model

We aim at securing digital assets against software based threats. Hence, we do not consider protecting infrastructure from physical theft. We assume service providers are trusted i.e they are neither malicious nor compromised. However, a provider may have an economic incentive to violate (not fulfill) some agreement terms hence, the requirement for transparent verification process. In general we assume attacks that can be detected by IDSs, specifically by NIDSs. However, there is no specific implementation detail which restricts adaptation to HIDSs hence, our method can easily be adapted to HIDSs. An attack can be launched from outside the cloud (from tenant input to the cloud API or from anywhere outside the cloud infrastructure) or from inside (from other virtual machines). An attacker could be inside the cloud co-located with the target VM hence a co-located tenant can be a threat.

We assume attacks on different layers of the network. Application layer attacks like SQL injection, cross site scripting (XSS) or application layer DDoS attacks are considered. Self-propagating programs like worms spreading through a network, brute forcing, port scanning techniques that are used to probe a system remotely and determine what TCP/UPD ports are open (and vulnerable to attack) are also considered. In the network layer different DOS attacks (like ICMP, SYN flooding), NIDS detectable IP spoofing (forging or falsifying the source IP addresses in network packets) and IP fragmentation-based attacks, man in the middle attacks (an attacker secretly relays and possibly alters the communication between two parties who believe they are directly communicating with each other) are considered.

We do not consider attacks that are targeted against the monitoring device itself, like IDSs evasion techniques presented in Section 2.4.3. Also, hardware attacks that can affect other co-located VMs are not considered. If there is a specific attack that is not covered by a proposed method, it will be specified in the corresponding chapter. From a practicality point of view, we believe that this is a reasonable model. Tenants care more about direct threats on the higher level of the cloud tier (closer to their service) than low level system attacks. Moreover, considering attacks on the higher level of the cloud tier helps the providers to offer more user-tailored security monitoring SLAs.

3.5 Including Users in the Security Monitoring Process Through an SLA

As presented in Section 3.1, user-centric security monitoring is achieved through SLAs. In this section we present how users are included in the process of security monitoring.

3.5.1 SLA-Based Users Participation

Users participation in the process of security monitoring through SLAs means allowing users to be part of the SLA life-cycle. As shown in Section 2.5.3, we presented the three phases of the SLA life-cycle,

SLA definition and negotiation, SLA enforcement and SLA verification. In this section we elaborate each step by describing tasks and actions of a user in the process for security monitoring SLAs.

Security monitoring SLA definition provides a quantitative measurement for the security monitoring setup. In the definition of such SLAs, users need to describe their needs and requirements in order to make the monitoring relevant for themselves. Taking users' requirements into account is the core of being a user-centric service. With an interactive process users can decline an offer if it does not satisfy their needs and demand an updated offer. This way users can participate in the negotiation process.

Tenants are expected to describe their security monitoring requirements in terms of vulnerabilities. For security monitoring SLAs the agreement will cover *only known vulnerabilities.* Adding unknown vulnerabilities into SLAs is very difficult mainly because it is challenging, if not impossible, to determine the performance of a monitoring device for an unknown vulnerability. This affects both service providers and tenants on SLA definition and verification. Covering only known vulnerabilities will help to bridge the gap between tenants knowledge of vulnerabilities and actual implementation offered by providers.

Enforcing an SLA will follow after a provider and tenant agreed on terms. Providers are responsible to configure and provision the requested service according to the agreed terms to achieve described SLO. Once the SLA is enforced there should be a verification mechanism to check its satisfaction. Both tenants and providers can perform verification (continuous verification of an SLA is call *monitoring an SLA*). There can be different types of verification and monitoring techniques. We present a verification technique that a tenant or a provider can perform at any given time and a monitoring technique where tenants and a provider participate in the SLA monitoring process using distributed algorithms. The penalty described in the agreement will be applied for any violation of SLA term(s). In the next sections we detail the three phases of security monitoring SLA life-cycle.

3.5.2 Defining Security Monitoring SLAs

Tenants have a security policy describing what is allowed or not in their information system infrastructure. The security monitoring policy is part of this security policy. When outsourcing their information system to a service provider, tenants aim to achieve an equivalent security policy. Hence, monitoring requirements will be specified which are aligned with the ones described in the policy.

Usually a security SLA (also called SecSLA [186]) covers security properties like confidentiality, integrity and availability. As described in Section 2.5.5 these properties (specifically integrity and confidentiality) are hard to measure and they are described with parameters like: "how often should the integrity be checked?" And "what type of data encryption technique should be used?" In our study we provide another way to include security in SLAs. That is by including the performance of security monitoring devices into SLAs. Such devices monitor general vulnerability of a software regardless of the effect on a specific security property.

Security monitoring SLAs, in addition to the components described in Section 2.5.2, include a list of vulnerabilities to be monitored and the relationship between vulnerabilities, softwares and infrastructure where the software is running. To formalize the definition of SLAs, we proposed an extension to the CSLA language (see Section 2.5.4) and we call it *extended CSLA (ECSLA).* ECSLA, as an extension of CSLA, inherits the fuzziness and confidence ratio properties of an SLO. As a result of the dynamic nature of the cloud, usability of such additional margin increases while dealing with security. ECSLA also introduces new features including the addition of a new generic service, a structure to define security vulnerabilities, and a definition of security monitoring service.

The generic service introduced in ECSLA can be extended to other type of services which can not be defined using the original CSLA language. The vulnerability description is not tied with any type of monitoring mechanism. Hence, it can be used, as part, to describe security monitoring SLAs for different types of mechanisms. ECSLA describes a security monitoring service as a correlation between infrastructure, product (software) and vulnerability. The structure to define infrastructure and product is described in the original CSLA description.

There are thousands of known vulnerabilities and providers need to prepare for any request from tenants. A tenant may request a single vulnerability or combination of multiple vulnerabilities to be monitored. Hence, providers should know the performance of their monitoring tools on a single or any combination of vulnerabilities in advance. We propose an efficient way to prepare SLO templates by building a knowledge base on the performance of monitoring devices. The knowledge base is built by running performance evaluation tests. The relation between the number of required evaluation and number of vulnerabilities is in factorial (i.e for n vulnerabilities, the required number of evaluation test is around $n!$). Hence, for few thousands of vulnerabilities the required number of evaluation exceeds

the acceptable amount and becomes practically infeasible. To overcome such increase in the number of required evaluation we propose a clustering technique that groups vulnerabilities based on heuristics. By performing evaluations per group we can reduce the required number of evaluations.

In the definition phase of a security monitoring SLA cooperation is required between tenants and providers. For example, a tenant should disclose the required amount of details about software running inside the infrastructure and providers should disclose the required amount of detail about the internals of the monitoring process. Chapter 4 provides detailed description, implementation and experimental results of security monitoring SLA definition.

3.5.3 Enforcing SLA Terms

Once a tenant and provider agreed on SLA terms the next step is to implement them i.e to configure the infrastructure accordingly and make the monitoring service available for tenants. Since, service providers own the underlying infrastructure they are responsible for the enforcement process. Enforcing a security monitoring SLA is specific to the monitoring device used. This is a challenge to have a generalized description of requirements independent from the monitoring mechanisms. We defined a language which can be used to describe requirements in terms of vulnerability and it is independent from specific monitoring mechanisms but as described in Section 3.2.3 these is not an optimal solution as tenants may not have knowledge about security vulnerabilities of a system.

In our study we considered few scenarios of enforcement mechanisms, specifically enforcements using NIDSs and cryptographic hash computation for data integrity. Even for the case of NIDS, we used a direct and simple approach. Detailed and complex enforcement mechanisms (e.g if multiple NIDSs are configured to work together,) requires more study. Service providers and tenants should cooperate together in this phase by preparing the necessary configuration setup in order to allow easy verification for tenants. We describe the different types of enforcement mechanisms that were studied in this thesis in Chapters 4 and 6.

3.5.4 Verification of SLO Fulfillment

In order to check the fulfillment of the SLA terms and detect any violation, a verification procedure is required. A user-centric approach in the verification process should allow users to perform the procedure at any time or according to the frequency defined in the SLA. Providers can also conduct verification at any time. In fact, verification is testing a configuration, in our case the configuration of a monitoring device, to see if it achieves the expected performance. The method used to check SLA fulfillment is a procedure of testing an NIDS configuration, and it can be applied for other purposes, e.g. a provider can use the same process to prepare SLO templates in advance. Chapter 4 presents how a provider can use the procedure to prepare SLA templates.

In the SLA definition phase, every detail about the verification including the procedure used to verify an SLO, the expected prerequisite, exceptions where the SLO is exempted, and related issues should be specified. An SLA term is violated if the verification shows dissatisfaction (e.g. performance lower than the promised value). For every violation of a term, a penalty as it is defined in the SLA, will be applied. If the violation continues or if there is a threshold and the maximum number of violations is reached, other actions like renegotiating the SLA, reconfiguration of the information system to respect the SLO or even termination of the SLA can follow the penalty. The actions after detecting a violation are out of the scope of this thesis.

In our work we studied two different types of verification mechanisms. First, an independent algorithm that can be used by both providers and tenants to perform verification is presented. In this verification process a controlled, dynamic attack campaign is performed. As this is a verification process, the objective is to see how the monitoring device performs in the event of incidents and check if it reaches the expected performance as defined in the SLO. To successfully, i.e without damaging the production service, perform such campaign, cooperation is required between the tenant and provider. A provider needs to prepare a network setup so that the attack does not spread to production services. In addition, providers are expected and trusted to give back untampered outputs (i.e logs and/or alerts) of monitoring devices for tenants. Tenants need to keep services running in order for providers to perform verification. Chapter 5 presents more details on this type of verification process.

Second, we present a technique which uses a distributed algorithm where a provider and tenants need to participate in the process to securely and transparently keep verification results. In this process no trust is required between the provider and tenants. The distributed algorithm is designed in a way that,

data stored in the system cannot be changed as long as the majority of the participants are cooperating to keep the value. The distributed system is composed of a service provider, tenants and potentially third party entities. We assume that the majority of the participants in the system are not controlled by the provider.

3.6 Summary

In this chapter we have introduced what does user-centric security monitoring means and showed the need to have tailored security monitoring services in clouds. We presented one way to achieve such service using SLAs. By its nature an SLA links the provider and tenants thus it helps to reconcile between the requirements of tenants and capabilities of provider. We studied a novel security SLA which incorporates performance of security monitoring devices rather than security properties (confidentiality, integrity and availability) terms that are used in traditional security SLAs. This way we can circumvent the challenges of quantifying security properties, specifically confidentiality and integrity. However, it should be noted that our proposed method of addressing security SLAs is not to completely replace traditional SecSLAs rather it should be taken as a complementary mechanism in achieving the ultimate security SLA.

Security monitoring devices are used to oversee the security status of a given system and as a mechanism to enforce security policies. Security incidents are reported using such monitoring tools. Therefore, guaranteeing the performance of security monitoring devices indirectly guarantees security properties. Tenants are included in the process of the SLA life-cycle, namely in SLA definition, enforcement, and verification processes. We also pointed out the need for cooperation between tenants and providers in order to have an end-to-end security monitoring.

Challenges of designing and implementing user-centric security monitoring SLAs are also discussed. The challenges revolve around difficulties of describing and enforcing security requirements, measuring performance and storing results without bias, and the dynamic nature of the cloud. We did not address all challenges in this work, specifically

- Users are expected to describe their requirements in terms of software vulnerabilities. We did not provide a higher level description to express tenants' requirements. To overcome such a limitation service providers can offer a separate *vulnerability assessment* service. Using such service will reduce the burden for tenants and also help to find vulnerabilities that are not publicly available. Specially if a tenant is deploying custom software (developed in house and not available in public either commercially or as open source) vulnerability assessment should be performed.

- Regarding SLA enforcement, security monitoring SLA enforcement is specific to the monitoring probe used. In our study we focused on NIDSs and we used a simple and straightforward enforcement mechanism. A more complex enforcement mechanism (e.g if multiple NIDSs are configured to work together) requires more study.

It should be noted that our SLA definition and verification techniques are independent of the enforcement process. Hence, any kind of mechanism can be easily integrated. In this chapter we also presented the system and threat models assumed in our work. The need to have a transparent verification method to check the fulfillment of SLOs is presented. Such verification method should allow both tenants and providers to do the procedure.

Chapter 4 describes the definition of security monitoring SLAs terms taking a specific monitoring device as an example. Chapter 5 presents a technique used to verify the fulfillment of the objectives that are promised in a security monitoring SLA. Chapter 6 presents a method where users directly participate in keeping monitoring records secure and that provides evidence in the case of SLA violation incidents.

Chapter 4

Defining Security Monitoring SLAs

4.1 Introduction

In the previous chapter, we presented a general and high-level view of how to achieve user-centric security monitoring in clouds. Specifically, we have shown that cloud SLA can help to realize user-centric security monitoring. Tenants can be part of the cloud SLA life-cycle in order to give them control, more choices and more flexibility in the process. We have also separated the SLA lifecycle in three phases namely *SLA definition and negotiation, SLA enforcement and SLA verification*. In this chapter, we describe in detail how we can achieve the SLA definition and negotiation phase.

As explained in Section 2.5.3 SLA definition is a pre-negotiation phase where service providers draft SLA templates according to the services they provide. SLA definition happens before any contact with a tenant, providers assess their performance and prepare templates to offer for potential tenants. The negotiation process starts after a tenant discovers a provider and shows an interest in the provided service. Activities in the SLA definition phase include describing and preparing quantifiable objectives for the provided service. In this phase the service description should clearly address the type of service(s) covered under that SLA and providers precisely measure their performance and generate a *Key Performance Indicator* (KPI).

From the provider perspective, SLA definition is an advertisement for the provided service. It should be clear, precise and it should attract targeted customers. Service providers publish and advertise their services using the draft template SLAs. Tenants are also expected to perform some activities before signing off an agreement with a service provider. In [179] the Cloud Standard Customer Council (CSCC) describes ten steps that help to compare multiple cloud providers or to negotiate terms with a selected provider.

As presented in Section 2.5.2 SLAs can address the different cloud service models. In this chapter we assume an IaaS system. Chapter 3 described the lack of security monitoring terms in SLAs. The majority of existing SLAs addresses system availability but fails to give a guarantee on other aspects of security. We approach this problem by including security monitoring terms into SLAs. Security monitoring is a process (see Section 2.4) and different tools are used to perform this process (see Section 2.4.2). In our work *Network Intrusion Detection Systems* (NIDSs) are used as a monitoring device and security monitoring SLAs are defined for NIDSs.

In our context, security monitoring SLA terms refer to SLA terms that are designed to give guarantees on the performance of NIDSs. Our work follows a user-centric approach in the sense that it gives more choices and flexibility for tenants. Tenants can specify which vulnerabilities to be monitored, can verify whether a specified objective is reached and can take actions if there is an occurrence of SLA violation.

In the next section, we describe the objectives and problems that are addressed in this chapter. Specifically, how can we formally describe security monitoring SLA terms? What are the relevant metrics to describe the performance of an NIDS and how to include these metrics in security monitoring SLAs? In the presence of tens of thousands of vulnerabilities how a provider can prepare tenant-tailored SLA. To this end, we show how to determine the performance of an NIDS on a subset from tens of thousands of vulnerabilities and the effect on the performance of an NIDS regarding an increased number of vulnerabilities. Additionally, we present our approach to address the issue of defining security monitoring SLA terms with appropriate NIDS metrics. In this chapter, we also provide experimental results to validate the proposed ideas.

4.2 Objectives and Problem Description

In this section, we present the objectives of this chapter and we describe problems that we address. SLAs are fundamental components of the cloud computing model. As a result, clearly defined SLAs facilitates the regulation process throughout the service life-cycle. In this chapter, we aim to achieve the following objectives.

4.2.1 Objectives

In general, the objective of this chapter is to provide a mechanism which enables to define security monitoring SLA terms. More fine grained objectives of this chapter include:

- The mechanism should follow a user-centric approach, i.e. users should be able to participate in the process, have choices and flexibility on what type of vulnerabilities should be addressed.
- For a tenant to describe her/his needs, the mechanism should require fairly little knowledge about the details of the system which is outside their own environment, i.e it should not require technical knowledge about the system controlled by the provider.
- From providers the perspective, the mechanism should enable them to prepare custom tailored SLA for each tenant according to their requirements.
- The mechanism should allow describing the performance of monitoring devices with fine-grained, relevant metrics. It should also allow describing the process of computing composite metrics.

In order to achieve these objectives, we need to tackle some problems. We have identified the following problems.

4.2.2 Problem Description

The cloud comes with a characteristic of pay per use which magnifies the significant variation in consumer needs. Hence, SLAs have to be created for each tenant by a negotiation process. The communication between the provider and tenants should be in a standard language. This provides assistance for tenants to communicate and compare SLA offers from different service providers without the need to adapt another communication standards or language.

SLA definition should incorporate components listed in Section 2.5.2. For security monitoring terms the service description should be able to express *vulnerabilities* as they represent the finest granular concept in our SLA description. In addition, a vulnerability should be related to other services like products (software) where the vulnerability exists and infrastructure where the product is running. Currently, to the best of our knowledge, there is no SLA language that can satisfy these requirements.

The KPI describing the objectives of a monitoring device should be relevant for the specific device used, in our case for NIDSs. As described in Section 2.4.4 two characteristics of a metric were set as prerequisites before being used as SLO for terms describing NIDSs performance. These are (i) the metric should take *base rates* into consideration. Axelsson [21] showed the base rate fallacy problem for NIDSs and indicated the importance of base rate. (ii) it should be a single valued metric i.e not a combination of two or more metrics. Comparing coupled metric values from different providers requires analysis and finding trade-offs. Hence, from tenants perspective, it is easier to have a single unified metric. The non-deterministic nature of the base rate makes it challenging to define security monitoring SLAs. An SLA should not be defined for specific values of the base rate because, most of the time, every occurrence of an attack has a different rate, i.e. different base rate.

As described in Section 2.5.4 many languages have been developed to define SLAs for different types of services. Our discussion includes languages that were developed for web services before the cloud and DSLs specifically designed for clouds. However, none of these languages can describe security monitoring terms with the requirements discussed above.

From the providers perspective, there are two facts which make the SLA definition process challenging and needs consolidation. The first one is the fact that tenants need custom tailored security monitoring services. This is directly related to the nature of cloud services. By design the cloud is flexible, and tenants can configure it as they want. As a result, no one security monitoring configuration can fit all tenants needs.

The second challenge is that there are thousands of vulnerabilities and almost all tenants are interested in a few of these vulnerabilities. One tenant can be interested in three vulnerabilities and another tenant can be concerned with twenty vulnerabilities. This is intuitive considering tenants are running different services on various types of configuration. Service providers should know the performance of their monitoring device on every subset of vulnerabilities. But performing a direct measurement for every subset of vulnerabilities is not practical because it requires a large number of computations.

Moreover, in practice the number of vulnerabilities may vary from time to time. For example, a patch is available for a given vulnerability and monitoring that vulnerability will no longer be necessary. Alternatively, the other way round, a new vulnerability concerning a tenant may be discovered and the tenant requests monitoring against this vulnerability. In our work, we do not address a dynamic change in the SLA, Section 4.7.4 describes how to deal with such changes.

To perform the monitoring task using NIDSs, rules should be crafted and added to the configuration of an NIDS. The number of rules configured in an NIDS affects its performance, i.e as the number of rules increases the performance decreases. To make an NIDS custom tailored means to configure a tenant's NIDS with rules only concerning that tenant. Hence, service providers should know the performance of their monitoring device for every possible combination of available vulnerabilities. Given the high number of vulnerabilities it is not practical to do the evaluation task for every combination.

4.2.3 Contributions

In summary in this chapter we present the following contributions:

- We propose a security monitoring service description with relevant KPIs, specifically an SLO for NIDSs with a single unified metric which takes the base rate into account is used to describe the performance of an NIDS. Section 4.3 describes the KPI used in our SLA and other components of security monitoring SLAs.
- To address the lack of a standard language for security monitoring SLAs, specifically to be able to describe vulnerabilities and their relation with other services, we propose an extension to *CSLA* [123], a DSL used to describe cloud SLAs. Section 4.4 describes the extension in detail.
- Defining a KPI which relies on the base rate is challenging. This is because measuring the value of the base rate before the occurrence of an attack is difficult, if not impossible. We propose an interpolation-based mechanism which takes performance points from known base rates as an input, build a model and estimate the expected performance using that model for new base rate values. Section 4.5 presents the method in detail.
- In order to prepare SLO templates, we propose a method which builds a knowledge base for NIDS performance for a large number of vulnerabilities. The method aims to reduce the number of required performance evaluations since evaluating the monitoring device over all possible combination of vulnerabilities is not practical. The method is described in Section 4.6.

The next section presents the components of security monitoring SLAs.

4.3 Components of Security Monitoring SLAs

Like any other type of cloud SLAs, security monitoring SLAs contain the components described in Section 2.5.2. Besides the anatomy similarity with other clouds SLAs the content of security monitoring SLAs differs from others in few ways. In this section we present components of security monitoring SLAs, classes of SLO that we consider and the KPI used to describe the performance of NIDSs. Security monitoring SLA components include:

- *Service Description:* The security monitoring service is used to monitor existing vulnerabilities in a product. Hence, the service definition correlates three components (i) product: a software running in tenants infrastructure that contains vulnerability (ii) vulnerabilities: as a product may contain more than one vulnerability it should be specified which vulnerabilities are covered in the agreement. (iii) infrastructure: where the product is running. NIDSs require information about the infrastructure where they perform the monitoring task. Specifically, an NIDS needs IP addresses of machines to be monitored.

Application	Version	Attacks
Apache	Apache/2.4.7 (Ubuntu)	Denial of service (DoS) and port scan
Mysql	14.14 Distrib 5.6.31	Brute force access
WordPress(WP)	V. 4.4.5	None
Instalinker (WP plugin)	V. 1.1.1	Cross site scripting (XSS)
Custom Contact Forms	V. 5.1.0.2	SQL injection

Table 4.1: Example of service list and attack types figuring in an SLO

- *Parties:* are participants in the agreement. Primarily, it includes providers and tenants. Supporting parties (e.g auditor) can be included depending on the need.

- *Penalties:* apply on violating party in cases of SLO violation. In our SLA definition a penalty can be defined in two forms. (i) Fixed type penalty which applies a fixed amount of retribution per occurrence of violation (ii) Function type where the penalty is based on a predefined function.

- *Service Level Objectives (SLO):* describe the guaranteed level of performance. For an NIDS, it shows how much a given NIDS is effective in detecting the vulnerabilities listed in the service description. The performance of an NIDS is described using C_{ID}. The next subsections provide details on why the C_{ID} is used and class of SLOs that are considered in our work.

4.3.1 KPI for Security Monitoring SLO

As described in Section 4.2.2, for a metric to be used in our SLO two features are required, namely a single unifying metric and the base rate. We saw in Chapter 2 that the Intrusion Detection Capability (C_{ID}) satisfies both of these features. The C_{ID} measures the ratio of the mutual information shared between the inputs and outputs of an NIDS and the entropy of the input. Mutual information measures the uncertainty in the input (i.e whether a given input packet is part of an attack or not) after knowing the NIDS output. Normalizing this value with the initial uncertainty of the input (i.e entropy) produces C_{ID}.

The definition of SLA describes how to compute the expected C_{ID} value from basic metrics (TPR and FPR) and a base rate. Since it is not possible to know the values of base rate in advance, SLA definition does not contain exact values of base rate. Section 4.5 presents how providers can offer SLO based on unknown base rate values.

4.3.2 Class of SLOs Considered

In the SLA definition phase providers prepare SLO templates. These templates describe the expected effectiveness of NIDSs using the C_{ID} metric while they are configured to monitor vulnerabilities. Providers prepare templates taking only *known* vulnerabilities into account. Tenants may require to be monitored against new or emerging vulnerabilities, but from the providers perspective, it is risky to promise the performance of monitoring devices for unknown vulnerabilities. Section 4.7.4 describes the issues related to unknown vulnerabilities.

Tenants describe their needs by listing vulnerabilities of their interest. The final SLA contains the list of vulnerabilities that are at the intersection between tenants requirements and security monitoring services proposed by the provider. The agreement describes the list of services to be monitored, the list of known vulnerabilities, and expected performance. Services are described with an application(s) that are used to provide that service. In addition, vulnerabilities are related to the applications version. Once the agreement is signed tenants' infrastructures are monitored against attacks which can exploit the listed vulnerabilities.

Example of services considered to be included in an SLA are described in Table 4.1. The table shows a list of vulnerabilities and attacks to be monitored including a Denial of Service (DoS) and a port scan against the Apache web server, a brute force access to the Mysql database server, a cross-site scripting (XSS) attack against Instalinker WordPress plug-in and an SQL injection against Custom Contact Forms (also a WordPress plug-in). An example SLA including this service is presented in the next section.

The next section presents a formal language to describe cloud security monitoring SLAs.

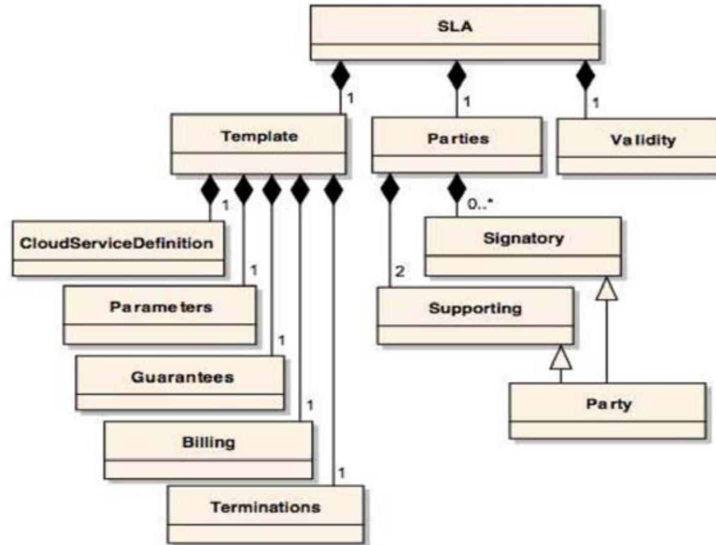


Figure 4.1: CSLA meta model [123]

4.4 Extended Cloud Service Level Agreement (ECSLA)

As presented in Section 4.2.2, one of the problems for realizing security monitoring SLAs is the lack of formal language to define an SLA. We discussed existing SLA languages in Section 2.5.4. Some of these languages were designed before the cloud era, and they do not address the distinctive characteristics of clouds. Languages designed specifically for clouds fail to include functionalities to describe security features, with the exception of availability. Most of the languages e.g SLAC [214] and CSLA [123] target functionality of a service like the response time.

To define security monitoring SLAs we need a language capable of describing vulnerabilities to be monitored and the relationship between vulnerabilities and other services, i.e. the relationship between a vulnerability and a software, showing on which software the vulnerability is residing, and relationship between a vulnerability and infrastructure, showing on which infrastructure the vulnerable software is running. To achieve this we propose an extension to a cloud Domain Specific Language (DSL), CSLA [123]. Before describing the proposed extension, we first present the CSLA language and the reason why we selected CSLA. An example SLA is presented at the end of this section.

4.4.1 CSLA

CSLA [123] is a DSL designed to address the needs of cloud SLA. Specifically, CSLA tries to take into account the dynamic nature of clouds by introducing SLA properties to tolerate violation. It also adopts a dynamic penalty model described in [110]. These properties allow service providers to tolerate fluctuation in SLO.

Figure 4.1 shows the meta-model of CSLA. SLA in CSLA contains three sections. These are *parties, validity* and *template*. *Validity* describes how long the agreement is valid and *parties* describe who is bound by the agreement. In CSLA there are two types of parties *Signatory parties* (service provider and service customer) and *Supporting parties* (e.g., trusted third party). *Templates* are structural models for SLA. A template contains five sections namely Services definition, Parameters, Guarantees, Billing and Terminations.

- Services definition: describe services following the standard cloud service models (SaaS, PaaS or IaaS). It uses the Open Cloud Computing Interface (OCCI) standard [91] for IaaS services definition.

- Parameters: used to define variables that are relevant to describe *Metric*, *Monitoring* and *Schedule* elements in other sections of the agreement.

- Guarantees: are elements containing the expected objectives. It contains four elements *Scope, Requirements, Terms and Penalties.*

  - *Scope*: specify services from the agreement that are covered by this guarantee.
  - *Requirements*: conditions that are expected to be fulfilled in order to achieve the objectives.
  - *Terms*: contain a set of guarantee terms connected by *'And'* or *'Or'* terms. Each term contains objective(s) that are defined by *expression* and *precondition.* A *priority* is also defined for each objective, and users have the option to set their preferences. An *expression* is characterized by different properties including a *metric*, a *comparator* and, a *threshold. Monitoring* and *Schedules* are also defined to specify how and when to evaluate the metrics. *Fuzziness* and *confidence* values are defined under the *expression.* As described in Section 2.5.4, these are parameters describing service degradation to deal with unpredictable environments.
  - *Penalties*: are applied in cases of SLA violation. *Constant* and *function* based penalties can be applied.

- Billing: describes the billing method for the provided service.

- Terminations: describes the termination procedure. These procedures will be followed depending on the *validity* section, specifically according to the date on *effectiveUntil* parameter of *validity* section.

The full syntax of CSLA is presented in [122].

4.4.2 ECSLA

The service description section of CSLA describes SaaS, PaaS or IaaS services. This is not enough to describe security monitoring services. We propose a generic extension to the service description section. Our extension helps to enable definitions for other types of services. In our case, the generic service definition is used to define the *SecurityMonitoring* service.

Figure 4.2 shows the model diagram of ECSLA. The dotted rectangular box shows the part from original CSLA. The *MacroService* definition represents a generic service class which can be extended to any type of class. The concept of using a generic class as part of the SLA structure is used in other languages like in SLA* [222]. The *SecurityMonitoring* service contains three sections namely *product, vulnerability,* and *infrastructure.* The *product* and *infrastructure* are defined in the original CSLA while the *vulnerability* is a new feature in ECSLA. A *product* is a general class to described software and platforms. Some properties of a product include *name, version, distribution, price* etc. Two types of cloud infrastructures are defined namely *Compute and Storage.* A *vulnerability* is characterized by *Id, Common Vulnerabilities and Exposures (CVE)* and *description.* The CVE is a list of vulnerability database entries, each containing an identification number, a description, and at least one public reference for publicly known cybersecurity vulnerabilities. CVE entries are used in numerous cybersecurity products and services from around the world [56]. We use the *CVE ID* to characterize vulnerabilities in ECSLA. However, some vulnerabilities may have no matching CVE, i.e when the vulnerability is not added to the database list. In that case, in ECSLA the vulnerability is characterized by *Id* and *description.*

In addition, in the original CSLA, the *parameter* is used to define variables that are required to describe metrics. The *metric* is described with attributes *id, name, unit,* and *description.* In the original CSLA, if a complex metric (computed from basic metrics) is used to describe an SLO, there is no way to define its formulation. Describing the process to compute a complex metrics is important in order to clarify misunderstandings. Thus, we extended the metrics definition of CSLA in order to include such a feature. Figure 4.2(C) shows the proposed extension in metrics definition. The *type* and *formulation* attributes are added. The *type* can be either *simple* or *complex.* The *formulation* formally describes how a *complex* type metric is computed. If a metric is of a *complex* type, then *formulation* shows how to compute that metric using basic metrics. The *formulation* can contain the actual formal definition or a reference to a formal definition. Appendix A shows the formal XML model for ECSLA.

4.4.3 Example Security Monitoring SLA

In this section, we provide an example of a security monitoring SLA showing the concepts described above. We consider the services described in Section 4.3.2 (see Table 4.1).

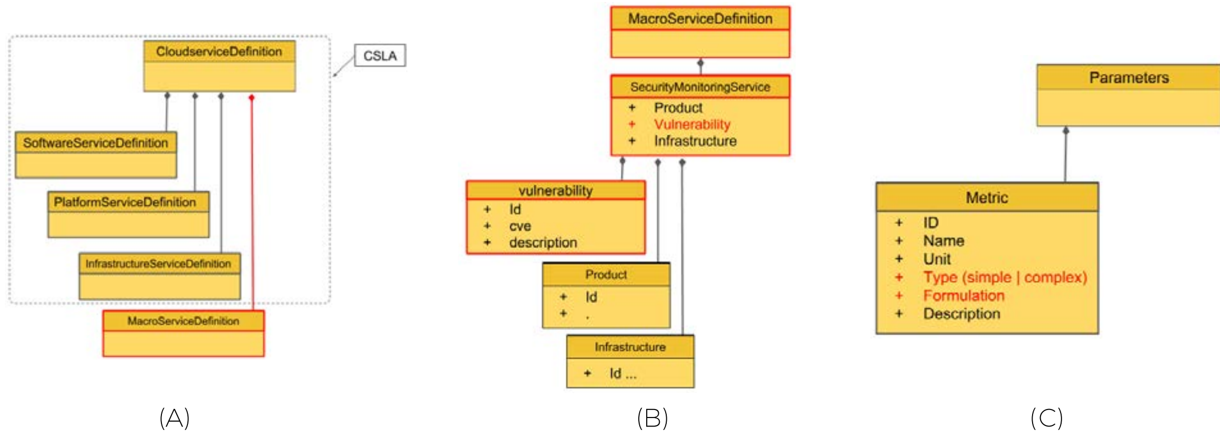


Figure 4.2: ECSLA model diagram (A) Macro service extension, (B) Security monitoring services and (C) Metrics

Service description

Listing 4.1 shows the service definition section for security monitoring SLAs. Four security monitoring services are defined; these are services to monitor vulnerabilities in Mysql, Apache, InstaLinker and Custom Contact Form. Each service contains one vulnerability except Apache, which is monitored for two vulnerabilities. The services are running in three different servers. The infrastructure section shows detail about the servers. A Web server, database server and content management server are used. According to the SLA, Apache is running on the web server, Mysql is running on the database server and WordPress plug-ins are running on the content management server.

```
<cloudServices>
<cloudService>
<macro>
<securityMonitoring id="Mysql-SM-ID" description="service definition for brute force login monitoring">
  <software id="Mysql-ID" name="Mysql" version="14.14"
        distribution="5.6.31" license="GPL" mode="mode" />
  <vulnerabilities>
    <vulnerability id="Mysql-V-1" name="Brute force access" cve_Id=""
             description="A token comparison based authentication is vulnerable to attacks of guessing (an
                  attacker can try as many times as possible to find the correct token)" />
  </vulnerabilities>
  <infrastructure id="DBserver-ID" description="Database server ">
    <compute id="DB-VM-1" name="DBserver" architecture="" hostname="" cores="" speed="" memory="" />
  </infrastructure>
</securityMonitoring>

<securityMonitoring id="Apache-SM-ID" description="">
  <software id="Apache-ID" name="Apache" version="4.4.11"
        distribution="" license=" Apache License" mode="mode" />
  <vulnerabilities>
    <vulnerability id="Apache-V-1" name="Port scanning" cve_Id=""
               description="scanning ports to detect available services on each port" />
    <vulnerability id="Apache-V-2" name="Denial of service" cve_Id=""
             description="congesting a network to alter the availability of a service" />
  </vulnerabilities>
  <infrastructure id="Websever-ID" description="Web server ">
    <compute id="WEB-VM-1" name="Webserver" architecture="" hostname="" cores="" speed="" memory="" />
  </infrastructure>
</securityMonitoring>

<securityMonitoring id="IL-SM-ID" description="">
  <software id="IL-ID" name="InstaLinker" version=" &lt;= 1.1.1"
        distribution="" license="GPLv2" mode="mode" />
  <vulnerabilities>
    <vulnerability id="IL-V-1" name="Cross-Site Scripting (XSS)" cve_Id="8382 in WPVDB_ID" description="Due
           to a lack of input sanitization in some file, it is possible to utilise a reflected XSS vector to
           run a script in the target user's browser and potentially compromise the WordPress installation."
           />
```

```
35   </vulnerabilities>
36   <infrastructure id="CM-ID" description="Content managnment server ">
37     <compute id="CM-VM-1" name="Cmserver" architecture="" hostname="" cores="" speed="" memory="" />
38   </infrastructure>
39 </securityMonitoring>
40
41 <securityMonitoring id="CCF-SM-ID" description="">
42   <software id="CCF-ID" name="Custom Contact Forms" version=" &lt;= 5.0.0.1"
43        distribution="" license="GPLv2" mode="mode" />
44   <vulnerabilities>
45     <vulnerability id="IL-V-1" name="SQL injection" cve_Id="7542 in WPVDB_ID" description="unauthenticated
          users to download a SQL dump of the plugins database tables. It's also possible to upload files
          containing SQL statements which will be executed." />
46   </vulnerabilities>
47   <infrastructure id="CM-ID" description="Content managnment server ">
48     <compute id="CM-VM-1" name="Cmserver" architecture="" hostname="" cores="" speed="" memory="" />
49   </infrastructure>
50 </securityMonitoring>
51
52 </macro>
53 </cloudService>
54 </cloudServices>
```

Listing 4.1: Security monitoring service description in ECSLA

Lines (3 -13) define monitoring of a brute force login in Mysql version 14.14 database. Lines (15 - 27) define a service for Apache to be monitored against port scanning and DoS. Lines (29 - 38) and (40 - 49) define security monitoring services for InstaLinker and Custom Contact Forms respectively.

Parameters

Listing 4.2 shows a list of parameters that are used to define the metrics, monitoring, and schedule. The SLO can be verified according to these definitions. Four simple metrics namely *TP, FP, TN, FN*, and a complex metric C_{ID} are defined as parameters, lines (2 - 24). For the formulation of C_{ID}, a reference is added to its definition in Section 2.4.4. In practice, the service provider can build a resource file describing a metric computation process, and the SLA definition can refer to this document for metric computation. Such practice (referring to other official documents for support) is not uncommon. For example, currently Amazon SLA [5] refers to a customer agreement document to exclude some cases from the SLA. The Parameter section also specifies monitoring and schedule, lines (26 - 27). Chapter 5 discusses how to choose the timing for verification. However, it is important to note that this is an agreement; every part can be negotiated and is set to values satisfying the participants.

In our example, the verification can be done three times in 24 hrs, and the minimum value should satisfy the expected SLO. The example also specifies a schedule with a start and end time to perform verification.

```
1   <parameters>
2     <metric id="TP" name="True positive" unit="count" type="simple">
3       <description description="The number of correctly detected attacks"> </description>
4     </metric>
5
6     <metric id="FP" name="False positive" unit="count" type="simple">
7       <description description="The number of legitimate inputs mistakenly classified as attacks"> </
            description>
8     </metric>
9
10    <metric id="FN" name="False negative" unit="count" type="simple">
11      <description description="The number of attacks that are not detected"> </description>
12    </metric>
13
14    <metric id="TN" name="True negative" unit="count" type="simple">
15      <description description="The number of legitimate that are classified as legitimate"> </description>
16    </metric>
17
18
19    <metric id="CID" name="Intrusion Detection Capability" unit="" type="complex">
20      <description description="The ratio of the mutual information between input and output to the entropy
             of the input
21                     (Single metrics to describe the effectiveness of an NIDS, computed from TPR,FPR and B)
                            ">
22        </description>
```

```
23        <formulation> As described in Section 2.4.4 </formulation>
24      </metric>
25
26      <monitoring id ="Mon-1" statistic="min" window="24 hrs" frequency="3"/>
27      <schedule id ="Sch-1" start="8:00pm" end="8:00am"/>
28    </parameters>
```

Listing 4.2: Example parameters in ECSLA

Guarantees

Listing 4.3 shows the guarantee part which contains three sections. The *scope*, lines (3 - 8), shows services that are covered under this guarantee. In our example, the guarantee addresses all the four services. The *requirements* section, lines (10 - 14), defines base rate boundaries. The guarantee is for base rate values greater than 10^{-7} and less than 10^{-1}. If the base rate is not in this range, the SLO may not be achieved and such incidents are not SLO violations. The *terms* section, lines (16 - 31), contains only one term, a term describing expected C_{ID} value. It describes that the expected C_{ID} value should be computed using the formula presented in Section 4.5. The next section describes why a formula is given, rather than an actual C_{ID} value. The section also presents the process used for generating such a formula. To show an example usage for such a formula, let us assume the NIDS is evaluated (or an attack occurs) with a base rate value of 7×10^{-2}, putting this value into the formula gives an estimated value of $TPR \approx 0.71046$ and $FPR \approx 0$, and from these values we can compute the expected C_{ID} value, $C_{ID} \approx 0.7162$.

The fuzziness value of 0.05 (5%) and confidence ratio of 95% are also defined in the term section. This is interpreted as: from 100 verification tests with a base rate value of 7×10^{-2}, in at least 95 of the tests, the configured NIDS must perform with $C_{ID} >= 0.7162$ and the remaining tests must perform with $C_{ID} <= 0.7162$ but $C_{ID} >= 0.6562$. As a result of fuzziness and confidence ratio, 5% of the verification tests are allowed to perform below the guaranteed performance level without violating the SLO. These margins allow for the SLO to fluctuate to some extent. This is an exciting feature, especially for security monitoring SLO because it gives some level of freedom in achieving the expected SLO. In addition, the monitoring frequency and schedule, which are a reference to the parameter section, are defined in the terms section.

```
1  <guarantees>
2    <guarantee id="G-1">
3      <scope id="Sc1">
4        <service id="Mysql-SM-ID" subid="Mysql-SM-ID-mode"/>
5        <service id="Apache-SM-ID" subid="Mysql-SM-ID-mode"/>
6        <service id="IL-SM-ID" subid="Mysql-SM-ID-mode"/>
7        <service id="CCF-SM-ID" subid="Mysql-SM-ID-mode"/>
8      </scope>
9
10     <requirements>
11       <Requirement id="R1">
12         <Specification id="Sp1" policy="Required"> Base Rate(B), B >= 10^(-7) and B <= 0.1 </Specification>
13       </Requirement>
14     </requirements>
15
16     <terms>
17       <term id="T1" operator="">
18         <item id="CIDTerm"/>
19       </term>
20
21       <objective id="CIDTerm" priority="1" actor="provider">
22         <precondition policy="Required">
23           <description> The threshold value should be computed with function as described in Section 4.5
24           </description>
25         </precondition>
26
27         <expression metric="CID" comparator="gt" threshold="" unit="" monitoring="Mon-1" schedule="Sch-1"
28                 confidence="95" fuzziness_value="0,05" fuzziness_percentage="5"/>
29       </objective>
30
31     </terms>
32   </guarantee>
33 </guarantees>
```

Listing 4.3: Example guarantee in ECSLA

Term	Proba. Representation	Description
FPR (α)	$P(A \mid \neg I)$	The probability that there is an alert, when there is no intrusion
TNR ($1-\alpha$)	$P(\neg A \mid \neg I)$	The probability that there is no alert, when there is no intrusion
FNR (β)	$P(\neg A \mid I)$	The probability that there is no alert, when there is intrusion
TPR ($1-\beta$)	$P(A \mid I)$	The probability that there is an alert, when there is intrusion

Table 4.2: NIDS metrics with probabilistic description

By now it should be clear that we are using the C_{ID} metric to describe the performance of an NIDS in SLAs. However, the guarantee definition contains a function (a model) to calculate the expected C_{ID} value rather than an actual number. The reason is that it is not possible to know how often attacks will occur i.e it is impossible to know the base rate value before the occurrence of attacks, hence we cannot calculate the C_{ID} value in advance while defining an SLA. Moreover, using a single base rate value in the SLA definition makes the SLA very strict, i.e the SLO will be achieved if attacks occur only with that specific rate. Using a model allows estimating the expected performance for previously unknown base rate values. Additionally, if there are cases where the model cannot predict well, the cases can be categorically excluded from the SLA. For example, in Listing 4.3 $B < 10^{-7}$ and $B > 10^{-1}$ are excluded.

4.5 Including Unknown Base Rate Values in SLO

In this section, we present a method used to generate a model. The SLA definition uses the model to guarantee the performance of a monitoring device. The model helps to make the SLA inclusive, i.e. to give guarantees even for previously unknown base rate values.

In Section 2.4.4 we have presented the base rate fallacy problem and Axelsson [21] showed the importance of the base rate in evaluating the performance of NIDSs. The base rate measures the prior probability of intrusion in the input data examined by the NIDS. Predicting or calculating the exact value of the base rate in advance is difficult if not impossible. As a result, the SLA definition will not include base rate values to describe the performance of security monitoring devices. Instead, the SLA will contain function(s) which takes the base rate as an input to calculate the expected TPR and FPR values. This helps to avoid an SLO which is defined for a specific base rate and be respected if an attack happens only on that rate.

As described in Section 2.4.4, C_{ID} is a function of TPR, FPR, and B. Any factor that can change the values of TPR and FPR indicates an effect on the performance of the NIDS. The factors affecting the performance of an NIDS can be grouped into two categories. First, *external factors* which includes mainly the rate of the inputs (throughput), the base rate and available resources for the NIDS. Second, *internal factors* which includes mainly the number of rules and the number of services that are configured to be monitored by that NIDS. Before describing the model generation method, we present the assumptions and challenges for designing such a method.

4.5.1 Assumptions and Challenges

If we are given a fixed number of services and vulnerabilities to be monitored, it is safe to assume that the internal factors affecting the performance of the NIDS are constant values. Indeed, since we have a fixed number of vulnerabilities to be monitored, we assume that the rules configured in an NIDS are constant. In addition, we assume there are enough resources that are required by the NIDS to perform the monitoring task. Hence, the change in the rate of the inputs (throughput) will not affect the performance of the NIDS by creating resource scarcity. The remaining factor affecting the performance of the NIDS is the attack rate or base rate.

The four basic metrics that can be counted from the NIDS output are (TP, TN, FP, FN). Using these basic metrics, we can calculate TPR, FPR, TNR, and FNR. These values can be described in probabilistic terms as shown in Table 4.2. Using these values and the base rate, C_{ID} can be computed as described in Section 2.4.4.

In realistic production sites the value of the base rate is very small (i.e close to zero). That means an attack packet happens very rarely compared to legitimate packets. Gu *et al* [99] assumed base rates in the range of $\{1 \times 10^{-2} - 1 \times 10^{-6}\}$. In 1999 Axelsson [21] supposed that the maximum value was $\{2 \times 10^{-5}\}$ (2 intrusions per day from $1,000,000$ records, an intrusion affecting 10 records in average).

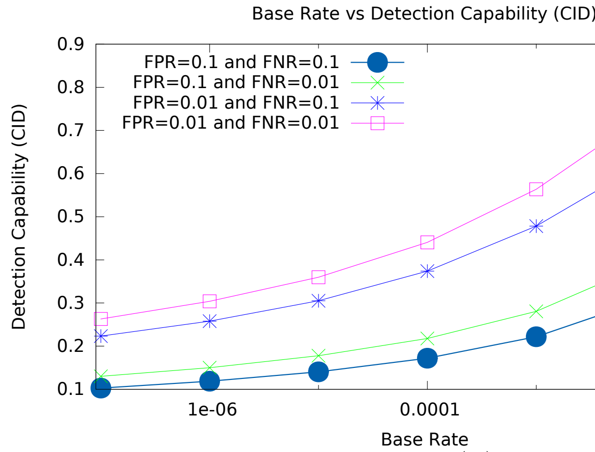


Figure 4.3: Base rate (B) vs C_{ID}

While doing an experiment, achieving such a very low base rate is challenging and it requires an enormous amount of resources. In our work, we evaluate an NIDS by injecting real attacks. The actual attack injection algorithm which takes the base rate as an input is presented in the next chapter. Here we present the challenges in a high-level description. While performing the attack injection, packets and related information are logged and analyzed later with the output of NIDS for metrics computation. To achieve low base rates, attacks are injected very rarely and to have a statistically sound result the injection is performed not only once but multiple times. Hence, the experiment takes a long time. In addition, logging packets for such a long time requires large disk space. The challenge exacerbates when assuming the experiment needs to be performed for a large number of vulnerabilities.

Additionally, we assume the SLA specifies the lowest guaranteed base rate value. i.e SLA will not be violated for attack occurrences below the specified base rate. However, such lower bound values should be as realistic as possible. Finally, we assume the provider can perform the evaluation at the lowest guaranteed base rate value at least once. Section 4.7.4 describes the drawbacks of our estimation as a result of these assumptions.

4.5.2 Metrics Estimation Method

Figure 4.3 shows the correlation between B and C_{ID} for different TPR and FPR values. The plot assumes TPR and FPR as constant values. A close look at Figure 4.3 shows a trend between the value of C_{ID} and different B values. Nevertheless, in practice configuring an NIDS to generate constant TPR and FPR values is difficult. This is because in a realistic operational environment the values of TPR and FPR are affected not only by its internal configuration but also by external non-deterministic factors, e.g. the base rate.

Following this observation, we performed experiments to identify if there is a correlation between B and (TPR, FPR). If such correlation exists, we can have a model showing the correlation from known values of B and (TPR, FPR), then we can use that model to estimate TPR and FPR values for unknown base rates. The estimated values can be used to compute C_{ID} for those unknown base rates.

To generate the model we propose an interpolation-based method. The method takes known values of TPR and FPR which are computed on different B values. These values are used to generate a fitting function that can approximate the points. The generated function(s) takes B as an input and produces TPR and/or FPR values. The function is used in SLAs to estimate the TPR and FPR values for previously unknown B value. In practice, such an estimation method may not provide the exact value, but it gives an approximation to the exact value. Moreover, the margins described in ECSLA (fuzziness and confidence ratio) helps to tolerate some degree of variation in the SLO. Given a security monitoring configuration, to generate a representative formula which can be used to estimate the TPR and FPR values, we follow the following steps:

- Compute the performance of the NIDS on a given configuration taking a known base rate value as an input. The procedure to do the evaluation is presented in the next chapter. The performance evaluation should include the lowest guaranteed base rate and as many other points as possible. Increasing the number of evaluation points results in an increase in the accuracy of the model.

- Using results from the performance test, find a correlation (function f) between TPR and/or FPR and B, i.e. f is a function of B and evaluating f at some B value produces TPR and/or FPR.

This way we can generate an equation to be used in the SLA definition. The function f can be used to estimate the expected TPR and FPR values for new B values. The function f may not represent the exact relationship, but it is derived from the best information available. Section 4.7.1 presents an experimental evaluation showing the feasibility of our metrics estimation method.

4.6 NIDS Performance with a Large Number of Vulnerabilities

By now it should be clear that we aim at defining SLAs to guarantee the performance of signature-based NIDSs. Service providers need to prepare SLA templates that will be offered to potential tenants at the start of the negotiation process. To achieve this, service providers need to build a knowledge base on the performance of their security monitoring ability. As described in Section 4.2.2 one of the challenges to prepare SLO templates is the fact that there are thousands of vulnerabilities and a tenant may choose any combination of those vulnerabilities. That means a service provider needs to prepare SLA templates based on all combinations of vulnerabilities. This is not practical as it requires the service providers to perform a huge amount of tests just to prepare template SLOs.

Let us assume there are n vulnerabilities that a service provider can monitor and offer to provide security monitoring service. A tenant is interested in k of these vulnerabilities, where usually $k \ll n$. To prepare an SLO template based on every combination means to prepare an SLO for a combination of n and k, $\binom{n}{k}$ for all k in (1 ... n). For example, if we have a thousand vulnerabilities, preparing an SLO for every combination of vulnerabilities require more than tens of millions of operations. This is very tedious and not efficient.

In addition, the number of services that are monitored under a given security monitoring configuration affects the effectiveness of the monitoring process. This consequence is intuitive, as having more services to monitor means having additional tasks and more input, hence there is an effect on the performance of the monitoring device. If we retake the above example, let us assume there are two tenants, the first one selects k vulnerabilities and the second one selects m vulnerabilities where $m > k$ and $k \in m$ (m includes all k vulnerabilities). The security monitoring service will perform better for the first tenant. The second tenant will have more tasks to be done; as a result, the performance will degrade.

In practice, NIDSs use rule(s) to monitor a vulnerability and more vulnerabilities means more rules to be added in NIDS. Thus, the inputs will be evaluated against more rules. We refer to the effect of a rule on other rules as *interference*, i.e when a rule interferes with the functioning of another rule.

Quantifying the interference is useful to compute the performance of an NIDS which is configured with more than one rule. Given an NIDS with such a configuration, i.e. an NIDS configured with more than one rule, we propose a performance evaluation method which uses the interference value between rules. Let us assume the NIDS is configured with n rules; for simplicity, we assume one rule is used to monitor a single vulnerability. For a given rule, the collection of the interferences between that rule and the remaining other $(n-1)$ rules form a *vector*. For all n rules, the collection of their interference vector forms a *matrix*. However, in practice, building such vectors and matrices requires a large number of computations. To reduce the required number of computations, we propose a clustering method which groups rules based on a given criterion. A formal description of interferences is presented in the next section.

In the next sections we address the problems described above, i.e we want to answer questions like, can we model and evaluate the effect of having more vulnerabilities, and thus more interferences between rules? Can we provide a better way for service providers to build a knowledge base while having a large number of vulnerabilities?

4.6.1 Modeling Rule Interference and its Effect on NIDS Performance

In this section, we want to address the problem of interference between vulnerabilities. Understanding the effect of a vulnerability on other vulnerabilities is essential in order to prepare custom-tailored SLA templates efficiently. To show the importance of measuring interference, let us assume we have n vulnerabilities and corresponding rules used in an NIDS. The performance of an NIDS varies depending on the number of configured rules. Assuming all other factors affecting the performance of an NIDS are constant, the performance and number of rules are inversely proportional, i.e the performance decreases as the number of rules increases. Moreover, it is not enough to know the existence of such interferences

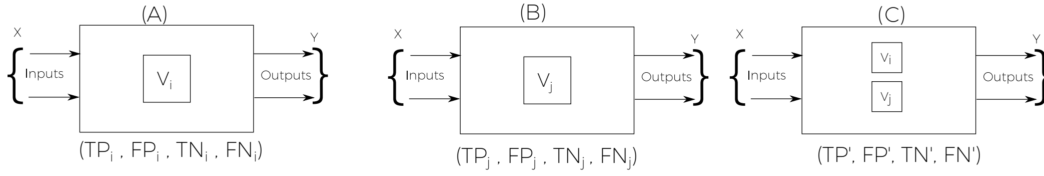


Figure 4.4: NIDS outputs

between vulnerabilities; we need to measure and *quantify* the level of interference in order to use it in the process of SLA definition. If a tenant selects k vulnerabilities out of n, the SLO offered for that tenant is the performance of the NIDS regarding those k vulnerabilities.

While selecting k out of n vulnerabilities, if we don't have a quantitative measure of the interference between the k vulnerabilities (i.e between the rules to monitor those vulnerabilities) then the provider is expected to run a test in order to measure the performance of its NIDS on k vulnerabilities. This process happens for every possible set of k rules. Having quantitative knowledge of the interference helps to do statical analysis rather than running a dynamic test. Therefore, it reduces the number of performance tests to prepare SLO templates. We present a formal definition of the interference and how to perform the static analysis in the next section.

4.6.2 Rule Interference in NIDS

As described above, the interference between rules refers to the effect of one rule on another rule (alternatively, it can be seen as the effect of a vulnerability on another vulnerability, assuming a single rule is used to monitor a single vulnerability). Before formally defining the interference between rules, we describe the assumptions taken in the formulation of the interference.

Assumptions

NIDSs take rules as an input. The rules are mechanisms to tell the NIDS what to look in the input packets. The detection engine of an NIDS applies the rules on each packet. If the packet matches a rule, the specified action of that rule is taken, and log(s) and/or alert(s) will be generated. However, if the packet matches with more than one rule, the NIDS behaves in one of the following ways:

- It generates an alert for all the matchings ;
- It generates an alert for few of the matchings based on some heuristics (e.g. the first match, the most severe).

If the NIDS follows the second option, in some cases it may stop further processing after finding the heuristic (e.g. after finding the first match). Usually, the second option is the default property, but even if it is not optimal, NIDSs can be configured to behave like the first option. While formally defining rule interferences we assume an NIDS configured to generate an alert for all matchings (the first case). i.e. if a packet matches with more than one rule, it generates an alert for every match. Assuming this we define interference as follows.

Formal definition of interference

Let us assume we have two vulnerabilities to be monitored (V_i and V_j) with a set of rules for each of them to be configured in an NIDS. For a given base rate (B), an NIDS configured to monitor V_i will generate (TP_i, FP_i, TN_i and FN_i) the same for V_j, it generates (TP_j, FP_j, TN_j and FN_j). These are the basic metrics that are presented in Section 2.4.4. They are measured by counting the number of attacks (or legitimate traffic) that are correctly (or incorrectly) classified by the NIDS from a given input. From these basic metrics we can calculate TPR and FPR as:

$$TPR_i = \frac{TP_i}{TP_i + FN_i} \quad , \quad FPR_i = \frac{FP_i}{FP_i + TN_i} \quad , \quad TPR_j = \frac{TP_j}{TP_j + FN_j} \; and \; FPR_j = \frac{FP_j}{FP_j + TN_j} \tag{4.1}$$

Vule.	V_1	V_2	V_3	......	V_n
V_1	0	$(FP, FN)_{1,2}$	$(FP, FN)_{1,3}$	......	$(FP, FN)_{1,n}$
V_2		0	$(FP, FN)_{2,3}$	......	$(FP, FN)_{2,n}$
V_3			0	......	$(FP, FN)_{3,n}$
......				......	......
V_n				......	0

Table 4.3: Interference Matrix (IM)

Interference between V_i and V_j is an event that can happen when both are configured together in one NIDS. Figure 4.4 (A and B) shows a box diagram, representation of an NIDS configured to monitor V_i and V_j separately. Moreover, Figure 4.4(C) shows an NIDS configured to monitor both V_i and V_j. As we have NIDS outputs for V_i and V_j separately, an NIDS configured to monitor both vulnerabilities together generates (TP', FP', TN' and FN').

We define the interference as the change in values between TP' and $TP_i + TP_j$ (respectively FN' and $FN_i + FN_j$) and between FP' and $FP_i + FP_j$ (respectively TN' and $TN_i + TN_j$). Formally we can define an interference as follows:

Definition 4.1. There is an interference between two vulnerabilities V_i and V_j if an NIDS configured with V_i gives a quadruple value (TP_i, FP_i, TN_i and FN_i), an NIDS configured with V_j gives a quadruple value (TP_j, FP_j, TN_j and FN_j), an NIDS configure to monitor both V_i and V_j gives a quadruple value (TP', FP', TN' and FN') and

$$TP' \neq TP_i + TP_j \;\; or \;\; FP' \neq FP_i + FP_j \;\; or \;\; TN' \neq TN_i + TN_j \;\; or \;\; FN' \neq FN_i + FN_j \tag{4.2}$$

In other words, if an NIDS is configured to monitor both vulnerabilities (V_i & V_j) and the output is not equal to the sum of separate outputs from V_i and V_j then we can say there is an interference. Note that interference is just the change in values it can be an increase or decrease of (TP', FP', TN' and FN') from the sum of (TP_i, FP_i, TN_i, and FN_i) and (TP_j, FP_j, TN_j, and FN_j). However, an increase in TP' or TN' is a *positive interference*. It indicates better effectiveness of an NIDS and such type of changes increases the C_{ID} value. It is unlikely for this event to happen.

Therefore, we are interested in *negative interferences* which indicate a degradation in the effectiveness of an NIDS and decrease the C_{ID} value. In other words, an increase either in the value of FP or FN means an increase of false classification or errors made by the NIDS. We can represent the interference between V_i and V_j as $(FP, FN)_{ij}$. The change in FP and FN can be described as:

$$FP_{ij} = FP' - (FP_i + FP_j) \;\; and \;\; FN_{ij} = FN' - (FN_i + FN_j) \tag{4.3}$$

Moreover, (TP', FP', TN', and FN') can be expressed as follows. Note that an increase in FP shows a decrease in TN and the same way and an increase in FN shows a decrease in TP.

$$\begin{aligned} TP' &= TP_i + TP_j - FN_{ij}, \;\; FN' = FN_i + FN_j + FN_{ij} \\ FP' &= FP_i + FP_j + FP_{ij}, \;\; TN' = TN_i + TN_j - FP_{ij} \end{aligned} \tag{4.4}$$

Interference vector (IV) and interference matrix (IM)

If there are n vulnerabilities, the interference between V_i and the other $n-1$ vulnerabilities can be described with an *interference vector (IV)*

interference vector (IV), $V_i = \{(FP, FN)_{i1}, (FP, FN)_{i2}, ...(FP, FN)_{in}\}$

Interference vectors of n vulnerabilities will form an *interference matrix* as shown in Table 4.3. An entry in the table shows an interference between the vulnerabilities in the corresponding column and row. The matrix is triangular and there is no interference for a vulnerability with itself.

Given such quantitative values of an interference, the next question is how we can compute the expected C_{ID} value for an NIDS configured to monitor both V_i and V_j? As we know C_{ID} is a function of TPR, FPR and B. Using Equation 4.1 to compute the TPR and FPR values for (TP', FP', TN', and

FN') may not be possible if there is interference between the vulnerabilities V_i and V_j. The equation needs to take into account the changes created as a result of putting multiple vulnerabilities together. In the next section, we present how to compute aggregated metrics, TPR and FPR values while aggregating multiple vulnerabilities in a given NIDS.

Computing aggregated metrics

Assume we have two vulnerabilities V_i and V_j, and an NIDS configured to monitor both V_i and V_j. The TPR' and FPR' values for this NIDS is represented as TPR_{agg} and FPR_{agg}. They are values computed from (TP_i, FP_i, TN_i, and FN_i), (TP_j, FP_j, TN_j, and FN_j) and $(FP, FN)_{ij}$.

If there is no interference between the two vulnerabilities, that means there is no change between the output of the NIDS and the sum of the separate outputs from V_i and V_j (i.e $(FP, FN)_{ij} = (0, 0)_{ij}$). From Equation 4.4 we have

$$TP' = TP_i + TP_j, \;\; FN' = FN_i + FN_j, \;\; FP' = FP_i + FP_j \;\; and \;\; TN' = TN_i + TN_j \tag{4.5}$$

TPR_{agg} and FPR_{agg} can be computed as follows:

$$\begin{aligned} TPR_{agg} &= \frac{TP'}{TP' + FN'} = \frac{TP_i + TP_j}{TP_i + TP_j + FN_i + FN_j} \;\; and \\ FPR_{agg} &= \frac{TP'}{TP' + FN'} = \frac{FP_i + FP_j}{FP_i + FP_j + TN_i + TN_j} \end{aligned} \tag{4.6}$$

A TPR is computed as the ratio between *correctly detected attacks* and the *total number of attacks*. Equation 4.6 can be interpreted as, since there is no interference between V_i and V_j the number of correctly detected attacks are the sum of TPs from V_i and V_j. The total number of attacks is the sum of attacks from V_i and V_j, i.e let us assume we use the packet flow f_i to evaluate the NIDS configured with V_i which contains a total number of $(TP_i + FN_i)$ attacks. Also, we use the flow f_j to evaluate the NIDS configured with V_j which contains a total number of $(TP_j + FN_j)$ attacks. Then to evaluate the NIDS configured with both V_i and V_j we use both f_i and f_j which results in a total number of $(TP_i + FN_i + TP_j + FN_j)$ attacks.

To keep the evaluation consistent f_i and f_j are the same. However, the flow contains packets that can trigger alerts for both of the vulnerabilities. Also, in some cases a packet may match more than one rule; as a result, the NIDS generates alerts for every match, as described in Section 4.6.2. Equation 4.6 can be generalized for n vulnerabilities as:

$$TPR_{agg} = \frac{\sum_{i=1}^{n} TP_i}{\sum_{i=1}^{n} TP_i + \sum_{i=1}^{n} FN_i} \;\; and \;\; FPR_{agg} = \frac{\sum_{i=1}^{n} FP_i}{\sum_{i=1}^{n} FP_i + \sum_{i=1}^{n} TN_i} \tag{4.7}$$

If there is an interference between V_i and V_j then there is a difference between the output of an NIDS configured to monitor both V_i and V_j and the sum of the separate outputs from V_i and V_j (i.e $(FP, FN)_{ij} \neq (0, 0)_{ij}$). Taking Equation 4.4, we can calculate TPR_{agg} and FPR_{agg} as follows:

$$\begin{aligned} TPR_{agg} &= \frac{TP'}{TP' + FN'} = \frac{TP_i + TP_j - FN_{ij}}{TP_i + TP_j + FN_i + FN_j} \;\; and \\ FPR_{agg} &= \frac{FP'}{FP' + TN'} = \frac{FP_i + FP_j + FP_{ij}}{FP_i + FP_j + TN_i + TN_j} \end{aligned} \tag{4.8}$$

If we have n vulnerabilities, we can generalize TPR_{agg} and FPR_{agg} as shown in Equation 4.9. In the equation, in addition to the assumptions described above, we assume that interferences are limited to pairwise interferences, i.e. the only interferences in the group are than the ones in the pairs of rules.

$$TPR_{agg} = \frac{\sum_{i=1}^{n} TP_i - \sum_{i=1,j=1}^{n} FN_{ij}}{\sum_{i=1}^{n} TP_i + \sum_{i=1}^{n} FN_i} \;\; and \;\; FPR_{agg} = \frac{\sum_{i=1}^{n} FP_i + \sum_{i=1,j=1}^{n} FP_{ij}}{\sum_{i=1}^{n} FP_i + \sum_{i=1}^{n} TN_i} \tag{4.9}$$

Given the performance of an NIDS on n vulnerabilities separately and the interference between them for some base rate value B, we can compute the expected C_{ID} value for that NIDS by doing only static

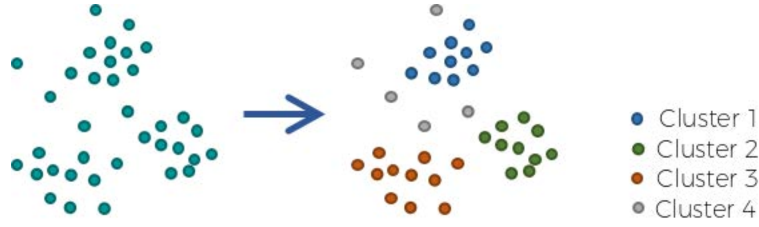


Figure 4.5: Clustering NIDS rules

analysis. This is done by computing TPR_{agg} and FPR_{agg} as shown above and use those values with B to calculate C_{ID}.

It is important to note that, we still have the issue with a large number of vulnerabilities. The method described above may reduce the number of required tests to prepare SLO templates to some degree, but it still requires to perform at least a test for every couple of vulnerabilities, i.e if we have n vulnerabilities the above method requires at least $\binom{n}{2}$ (combination of n and 2) performance tests. In practice when n is very large, the method may not be practical.

In the next section, we present our proposed solution to reduce the required number of evaluations in order to prepare SLO templates. Our goal is to have a knowledge base on the performance of an NIDS in order to prepare template SLAs and offer the templates to a potential tenant.

4.6.3 Building a Knowledge Base by Clustering Vulnerabilities

In order to realize a security monitoring SLA, the conflict between having a large number of vulnerabilities and the need to have custom tailored SLOs needs to be solved. Otherwise, running numerous performance tests just for building a knowledge base is not practical. Moreover, this process is not a one time task; it may be required to do the test on different occasions. For example, as new vulnerabilities are discovered the NIDS needs to be tested on those vulnerabilities. Hence, having an efficient method, which can reduce the required number of performance tests significantly affects the practicality of a security monitoring SLAs. In order to achieve this, we propose a method based on clustering vulnerabilities.

The idea is to perform the evaluation test per group rather than for each vulnerability. The clustering mechanism is applied to the NIDS rules based on heuristics, and the resulting groups are used to build a knowledge base for the performance of NIDSs. Clustering, as the name indicates, is an act of grouping elements to some kind of classes called *clusters* (see Figure 4.5). After constructing such groups the evaluation can be done per group, i.e configuring an NIDS to monitor all vulnerabilities in one group and measure the performance. If the interference between the vulnerabilities in a group is a *negative interference*, then the result of this evaluation is a *lower bound* for the performance of an NIDS configured with any subset of the group. However, even if the chance of having a *positive interference* is very small, the provider should be careful while setting the lower bound values. If the interference in a group is a *positive interference*, it may not represent a lower bound for a subset of the vulnerabilities in that group.

Let us take for example that we have one hundred vulnerabilities, and by clustering, we formed ten groups $(G_0, G_1...G_9)$ each G_i containing ten vulnerabilities $(V_0, V_1...V_9)$. Assume an NIDS is configured to monitor all the vulnerabilities in G_i. Let us say evaluating the performance of that NIDS for a given base rate value B results in C_{ID} value of x. If the interference between the vulnerabilities in G_i is a negative interference, then we can say that x is a lower bound on the performance of the NIDS for any subset of vulnerabilities from G_i. This is following Equation 4.9, assuming negative interferences, the FPR_{agg} for all V_is in G_i is higher than V_is for any subset in G_i. Hence, the C_{ID} will be the lower bound while taking all vulnerabilities in G_i. However, if the interference between the vulnerabilities in G_i is a positive interference, we can not set x as a lower bound for that group. In other words, if the interference is positive, then there may be a subset of vulnerabilities that could result in a worse performance than putting all the V_is in G_i together. It is unlikely for such an event to happen, but it is essential to consider the case while drafting an SLO.

Subsequently, Table 4.3 will have a smaller height and width. The columns and rows will represent groups of vulnerabilities (G_i) instead of single vulnerabilities (V_i). Service providers can choose appropriate heuristics to group vulnerabilities. Some examples of heuristics are described below. These are example heuristics, a provider needs to consider the available resources and the grouping criteria. Having a minimal number of groups results in meager SLO offer and having a large number of groups requires

Groups	G_1	G_2	......	G_n
G_1	...	...	......	...
G_2	...	...	......	...
...	...	...	......	...
G_n	...	...	......	...

Groups	G'_1	G'_2	......	G'_n
G'_1	...	...	......	...
G'_2	...	...	......	...
...	...	...	......	...
G'_n	...	...	......	...

Table 4.4: Multiple interference matrix from groups using different heuristics

huge amounts of computations. Providers need to select the heuristics considering available resources. Some examples of heuristics include:

- Grouping vulnerabilities based on related application, for example, OS vulnerabilities (vulnerabilities to monitor different OSs), browser vulnerabilities, ...
- Grouping vulnerabilities based on related applications that are used together, for example, grouping vulnerabilities in the LAMP stack (Linux, Apache, Mysql, and PHP) applications in one group,
- Grouping based on the nodes and services they are providing or used for, e.g. grouping vulnerabilities for login nodes together, storage nodes,
- Grouping based on the threat or severity of the vulnerabilities, for example, grouping less severe vulnerabilities together.

Having such groups and a minimized interference matrix, it is possible to compute a minimum expected performance for any given vulnerability. When a tenant needs to be monitored and selects the vulnerabilities from these groups, there are two cases. First, if the selected vulnerabilities are in the same group, then the performance of the NIDS on that group is offered as an SLO. Second, if the selected vulnerabilities are in different groups, then we can calculate the expected C_{ID} value using the method described in the previous section.

It is possible to group vulnerabilities based on multiple heuristics. Each heuristic produces a table as shown in Table 4.4. For a given set of vulnerabilities, the table which produces a better performance can be used to offer the SLOs.

In the next section, we present evaluations performed to validate the ideas proposed in this chapter.

4.7 Evaluation

We performed experiments to validate the proposed solution for the metrics estimation and rule clustering methods.The actual procedure for the experiment is presented in the next chapter. In this section, we present the setup and results from the experiment. The results show the basic metrics (TP, FN, FP, TN) of a given NIDS while configured to monitor a list of services. The results are shown for different base rate values. We also present how we managed to group NIDS rules based on some heuristics.

We aim to validate the C_{ID} approximation for unknown base rates and the efficiency of using the interference matrices, i.e to validate the efficiency of our clustering method to reduce the number of measurements required when preparing SLO templates. To make our experiment as realistic as possible, we perform a dynamic, real attack injection. By injecting real attacks we can see how a given NIDS performs and behaves on a specific configuration. This measuring technique is used to verify the correctness of an NIDS configuration. *Verification of SLO* compliance is the main topic of Chapter 5. Full details of the procedures are presented in the next chapter. Here we present an overview of the experiment setup and the result, counted as basic metrics, to show the process of SLO template creation.

4.7.1 Experimental Setup

Grid5000 [22] testbed infrastructure is used to run our experiments. We build a cloud infrastructure using OpenStack [163] and Open vSwitch(OvS) [165] as a virtual switch. Figure 4.6 shows a high-level architecture of the experimental setup. The tenant infrastructure is configured to run the services described in Table 4.1. For the experiment, three production virtual machines (VMs) are running the Apache, Mysql server and WordPress content management system. A fourth VM is also instantiated and used as a *target* for the injected attacks. The target VM exhibits similar properties as the production

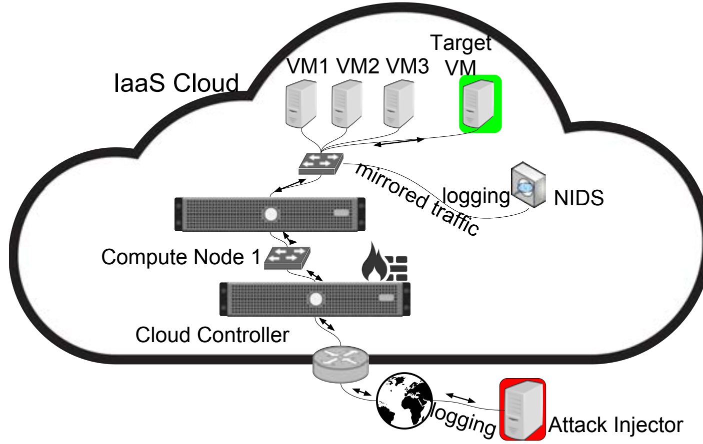


Figure 4.6: Experimental setup

VMs by running all the three services. The virtual switch is configured to forward all attack packets only to the target VM. Hence, the attacks are prevented from disrupting the production VMs.

Snort [13], the most widely deployed IDS, is used as NIDS. It is deployed on a separate physical node, and it is connected to the virtual switch through a mirror port. All packets passing through that switch are also mirrored to the Snort node for analysis. Snort is configured to monitor the services listed in Table 4.1 i.e to look for an instance of attacks listed in the table and to output an alert for each matching rule. An attacker machine is located outside of the cloud and it is used to inject attacks i.e. to perform a dynamic attack injection campaign. We use real attacks to exploit vulnerabilities in the target VM. The attacks are interlaced with legitimate traffic according to the base rate. In addition, we know the number of packets sent by each legitimate and attack request.

To measure the performance of a configured NIDS, we record all the communications between the attacker and target VM, i.e. we record all the injected packets. Snort gets all these packets as an input and generates an alert(s) when it detects an attack. At the end of the attack campaign, we get the output of the NIDS. Using recorded inputs and the output alerts, we count the number of inputs that are correctly (wrongly) classified as attacks (legitimate requests) by the NIDS. This way we can count the (TP, TN, FP, and FN) values. More details about the setup and its justification are presented in the next chapter.

4.7.2 Collecting Data Points and Generating an Estimation Model

Using the setup described in the previous section we run experiments to measure the performance of the configured NIDS. Table 4.5 shows the TPR, FPR, TNR, and FNR values, these are calculated from basic metrics (TP, TN, FP, and FN) using the formulas described in Section 2.4.4. The table also shows the calculated C_{ID} value for a given B. The values shown are averages over three rounds for each computation.

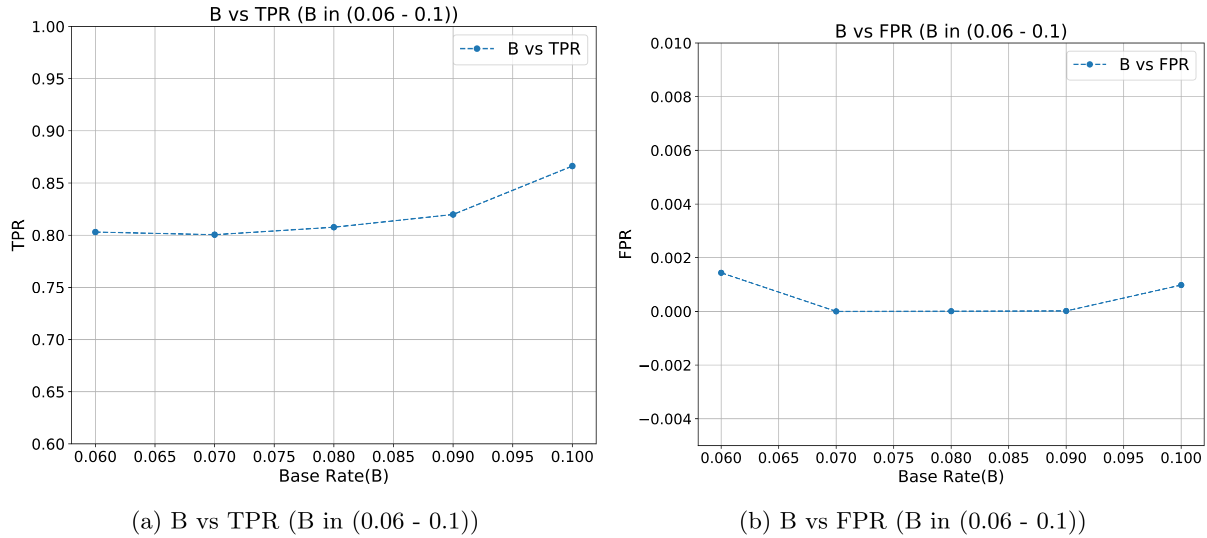
The experiment is performed for two sets of base rate values, for the smaller B values in $(0.001-0.01)$ and a slightly higher B values in $(0.06-0.1)$. For this experiment, we assume an SLA which guarantees the performance of an NIDS with a lower bound base rate value of 10^{-3}, i.e the SLA will not be violated if the NIDS underperformed for an occurrence of attack with $B < 10^{-3}$. It is important to note that in practice this value (10^{-3}) is a relatively large lower bound, i.e. attacks usually occur with a base rate value $B < 10^{-3}$. As described in the previous section, the actual value of the base rate is very small. However, in our environment, $B = 10^{-3}$ is the lowest achievable base rate value. Section 4.7.4 presents a detailed explanation on the limitation of our experiment.

Figure 4.7 shows the plot of B vs TPR and B vs FPR for B in $(0.06-0.1)$. We use Table 4.5 and the corresponding graphs to model the relationship between B and (TPR, FPR). From the table and the plots, we observe that the FPR values are very small (close to zero). This is because the rules in the NIDS are carefully crafted for the set of considered attacks. As a result, false positives generated by these rules are very small. This property is consistent for different base rate values. Other studies [183] showed that given the attacks to be monitored, it is possible to craft an NIDS rules with a minimal FP value. Taking these into consideration, we can use a constant $FPR = 0$ value in the SLA. However, for

	B	TPR	FPR	TNR	FNR	CID
Small B values	0.001	0.7005733333	0.000035352	0.9999566667	0.2994266667	0.6453533333
	0.003	0.7022433333	1.23E-07	0.9999966667	0.2977566667	0.6493066667
	0.005	0.7167533333	0.00000026	0.9999966667	0.26614	0.66004
	0.007	0.7278733333	8.63E-07	0.9999966667	0.2721233333	0.66845
	0.009	0.7567866667	0	1	0.2432133333	0.6968933333
	0.01	0.7338044444	8.12E-07	0.999996667	0.23843	0.700916667
Slightly higher B values	0.06	0.80297	0.001438933	0.99856	0.197023333	0.704433333
	0.07	0.80046	0	1	0.1995366667	0.71623
	0.08	0.8076033333	6.53E-06	0.9999966667	0.19239	0.73025
	0.09	0.8198066667	1.59E-05	0.9999833333	0.18019	0.76141
	0.1	0.8661833333	0.0009814231	0.9990166667	0.13381	0.7737933333

Table 4.5: TPR, FPR, TNR, FNR values of an NIDS from our experiment and calculated C_{ID} value for varying B

our SLA we take the highest FPR value, as C_{ID} is sensitive for changes in FPR. In the context of SLA, taking the highest FPR means promising a lower C_{ID} value, which puts the provider in a better position for not violating the SLA.



(a) B vs TPR (B in (0.06 - 0.1))

(b) B vs FPR (B in (0.06 - 0.1))

Figure 4.7: Plot of B vs TPR and B vs FPR for B in $(0.06 - 0.1)$

For the TPR, we can observe that its value is increasing with the B value. To model their relationship, we use the TPR values for B in $(0.06-0.1)$, and a TPR at the lowest guaranteed base rate, i.e $B = 10^{-3}$. We fit these points using a quadratic polynomial function, because there is only one local extremum value in the given range of B. The quadratic function f which best approximates these points is shown in Equation 4.10

$$f(B) = 0.7008827 + 1.357906 * B + 1.447843 * B^2 \tag{4.10}$$

Using this function, we can estimate the values of the expected TPR for other base rate values. As an example, Table 4.8 shows the estimated and actual TPR values for B in $(0.003 - 0.009)$. Figure 4.9 shows the plot of expected TPR in comparison with actual values. From this data, we can observe that our metric estimation method produce results which are close to the actual values.

Table 4.8 can be used as an input to drive the fuzziness and confidence ratio values. In addition to such a table, a provider may take other assumptions like the available resources to drive the appropriate fuzziness and confidence ratio values.

B	Estimated TPR	Actual TPR
0.003	0.7049694	0.7022433333
0.005	0.7077084	0.7167533333
0.007	0.710459	0.7278733333
0.009	0.7132211	0.7567866667

Figure 4.8: Estimated and actual TPR values

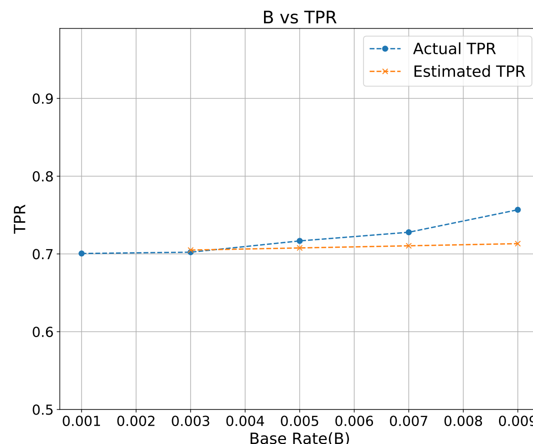


Figure 4.9: Plot of the estimated and actual TPR values

4.7.3 Clustering NIDS Rules

In Section 4.6.3 we presented a clustering technique that can be used to reduce the width and height of an interference matrix, thus reducing the number of required evaluation tests to prepare the template SLOs. In this section, we present an example using rules from Snort NIDS.

In the official snort rule repository, there are three categories of rule sets:

- Subscriber: The most updated set of rules which is available for paying customers and it is regularly updated.
- Registered: This rule set is 30 days behind the Subscriber rule set and it is distributed free of charge.
- Community: A set of rules written by the community and verified by the company maintaining Snort. It is also distributed free of charge and it does not contain the subscribers rule set.

We used the registered rule set version 2.9.8.0 to demonstrate the clustering technique described above. It contains a total of 10628 rules in 53 files. We used several properties to group the rules including properties that are part of the rule syntax. The properties used include the *application type*, the *attack class type*, the *severity of the vulnerability*, and *applications working together*.

Grouping based on application type

Another method is grouping applications with a similar type of functionalities into the same group. For example, grouping vulnerabilities in the browsers or operating systems together in one group. By default, rules in the Snort repository are grouped per application, e.g. Firefox rules, Internet Explorer rules, office file rules, etc. This default grouping can be used for our clustering purpose and our grouping method reduces further the number of groups. Table 4.6 shows an example of grouping created using the application type.

Grouping based on attack type and severity

An attack type (class type) is part of the rule options used in Snort syntax. Rules in the Snort NIDS are divided into two sections, the *rule header* and *rule options*; where the header contains information like the rule's action, protocol, source and destination IP/port. The option contains the alert messages and information about which part of the packet should be used to generate the alert message.

The keyword '*classtype*' is used to indicate that a rule is detecting general kind of attack. We can use this keyword to classify NIDS rules for building the interference matrix. A few default *classtypes* are defined, but it is possible to define a custom *classtype* in the *classification.config* file. Table 4.7 shows classification of rules based on the *classtypes* property.

Similar to this, Snort rules have a *priority* parameter which allows defining the severity with integer values. A *priority* is also defined in *classification.config* file. For example, the default configuration contains four priorities. A priority of 1 (high) is the most severe and 4 (very low) is the least severe.

Application Group	Applications (number of rules)	Total
Browsers	Firefox (15), Internet explorer (1286), Webkit (2), Others (8), Plugins (31)	1339
Files	Executable (17), Flash (1506), image (125), java (110), multimedia (55), office (462), pdf (506), others (331), file-identifier (979)	4091
OS	Windows (388), Solaris (3), mobile os (3), others (36), netbios (22), Linux (15)	467
Protocols	imap (2), scada (10), telnet (3), dns (1), pop (1), snmp (1), voip (2), others(4)	24
Servers	Apache (27), mail (9), mysql (1), oracle (2), samba (6), web-apps(273), mssql(1), others (223)	542
Potentially Unwanted Application (PUA)	Adware(30), p2p(3), toolbars (4), others (21)	58
Malware	Backdoor (110), CNC (3053), malware-tools (14), others (336)	3513
Others	app-detector (2), exploit-kit (492), indicator-compromise (30), indicator-scan (2), policy-social (2), policy-others (15), sql (16), deleted(3), indicator-obfuscation (32)	594

Table 4.6: Rule classification based on application types

Grouping based on applications working together

It is very likely that an application is used in collaboration with other applications. To give a simple example, an operating system is used to run almost all applications. Such a relationship can be used to group vulnerabilities. Such grouping is natural, as applications are used together to provide the expected service. Moreover, it can result in a higher performance value, as aggregation from different groups will be minimized.

Examples of such method include grouping Linux, Apache, Mysql, PHP (LAMP) and WordPress in one group, Windows, Microsoft Office products and Internet Explorer in another group.

Using such grouping, we can reduce the number of required evaluations. If we take the example rule set to build the interference matrix using each rule, it requires more than 54 million tests (combination of 10628 by 2). Using the default grouping by application, it requires 1378 tests (combination of 53 by 2) and using application type requires 28 tests (combination of 8 by 2). It is important to note that there is a tradeoff between the number of required tests and the performance of an NIDS (or its C_{ID} value). Less number of groups means less number of tests, but it also produces smaller C_{ID} values than a large number of groups. This is because when having a small number of groups, each group will contain more vulnerabilities than when having a larger number of groups. The provider should take into account available resources for the tests and select the clustering method accordingly. Multiple grouping methods can be used to build multiple interference matrices (see Table 4.4) and offer SLOs from the one which produces the best result.

4.7.4 Discussion

This chapter presents a feasible mechanism to define SLAs guaranteeing the performance of NIDSs. The previous section presented the experimental evaluation showing the validity of the proposed method. In this section, we present some issues that are related to the proposed method.

In ECSLA, the formal language used to describe security monitoring SLAs, tenants specify their security requirements using vulnerabilities for a given application. This level of abstraction is not ideal; it is not easy to know the existing vulnerabilities in an application for regular users. This creates a problem to make the security monitoring SLA life-cycle straightforward. As described in Chapter 3, service providers can cover this issue by offering a separate *vulnerability assessment* service. By using results from such a service, a provider can offer security monitoring SLAs with a higher level of abstraction.

The SLA definition described in this chapter uses a fixed set of vulnerabilities, i.e the set of vulnerabilities addressed in a given SLA is fixed. However, in a real situation the number of vulnerabilities that concerns a tenant may increase or decrease. An increase happens when new vulnerabilities concerning a tenant are discovered and a decrease happens when a patch is available for a vulnerability. In our SLA life-cycle both cases are not automated. Notably, the discovery of a new vulnerability could

Class Type	Description
attempted-user (4332)	Attempted User Privilege Gain
protocol-command-decode (46)	Generic Protocol Command Decode
denial-of-service (38)	Detection of a Denial of Service Attack
default-login-attempt (2)	Attempt to login by a default username and password
misc-activity (1049)	Misc activity
suspicious-filename-detect (1)	A suspicious filename was detected
attempted-dos (117)	Attempted Denial of Service
attempted admin (728)	Attempted administrator privilege gain
trojan-activity(3904)	A network Trojan was detected
string-detect(1)	A suspicious string was detected
bad-unknown (3)	Potentially Bad Traffic
network-scan (1)	Detection of a Network Scan
misc-attack (37)	Misc Attack
attempted-recon (146)	Attempted Information Leak
policy-violation (57)	Potential Corporate Privacy Violation
successful-recon-limited (12)	Information Leak
unsuccessful-user (1)	Unsuccessful User Privilege Gain
web-application-attack (150)	Web Application Attack
successful-user (4)	Successful User Privilege Gain

Table 4.7: Rule classification based class type

change tenants requirement. Such events are addressed by renegotiating and restarting the process of SLA life-cycle. To facilitate this, it is possible to include termination conditions in the SLA, describing the possibility of renegotiating when new vulnerabilities are discovered.

Moreover, our SLA definition addresses only known vulnerabilities. Monitoring an unknown vulnerability is usually performed by using anomaly-based NIDSs. With a few modifications, our SLA definition could be extended to be used for such a monitoring device. However, as it is in the current state, it cannot be applied to anomaly-based NIDSs because our service description requires describing the vulnerabilities; this is not practical for currently unknown vulnerabilities. Moreover, guaranteeing the performance of an NIDS covering unknown vulnerabilities is challenging and the risk of not meeting an objective is very high.

While computing the relationship between B and (TPR, FPR), we assumed the resources available for the NIDS are fixed. However, as described in [149], in the cloud environment resources for an NIDS may vary due to elasticity. Our assumption can be interpreted as the minimum amount of resource that is needed to perform the monitoring task. The available resources for an NIDS may increase but will not decrease below the level used for testing. Furthermore, an increase in resources should result in better performance. Hence, an increase in the available resources will not lead to an SLA violation.

The metric estimation method assumes that the provider can conduct a performance test using the lowest guaranteed base rate. In practice, this task is not straightforward. As described in Section 4.5.1, performing a test using a small base rate value takes a long time and requires large disk space. For example, to measure the performance an NIDS which is configured to monitor the services listed in Table 4.1 using $B = 10^{-2}$ takes around 42 minutes. Performing the same experiment by changing only the base rate to $B = 10^{-3}$, the experiment takes around four hours. Chapter 5 presents the actual procedure to perform a test and optimizations to reduce the required time by increasing the degree of parallelism.

To estimate the performance of an NIDS using real attack injection, it requires to have the attack which exploits a given vulnerability. As presented in Section 3.2.3, getting an attack to exploit a vulnerability is challenging. It is because, usually there is no incentive to publish an attack, especially for commercially owned products. Some issues related to the type of attacks used to perform the test are addressed in Chapter 5.

We conclude the chapter by presenting a summary in the next section.

4.8 Summary

In this chapter, we have discussed the problem of defining a security monitoring SLA. Specifically, we studied SLAs describing the performance of security monitoring probes. To show the proposed SLA definition method, the network security monitoring device we used is a signature-based NIDS.

We started by stating the objective, which is to provide a mechanism that enables the definition of security monitoring SLAs. We continued by listing the problems which need to be addressed in order to meet the objectives and to realize the security monitoring SLAs. The lack of a formal language to define SLAs, finding a relevant KPI i.e. a single unified metric which takes the base rate into account, and reconciling the fact that there are lots of vulnerabilities and the need for a custom-tailored SLAs are presented as problems.

In order to address the lack of a formal language in the SLA definition process, we proposed an extension to CSLA [123]. CSLA is a domain specific language specially designed to describe cloud SLAs. Our extension, called ECSLA, adds the ability to define a security monitoring service which contains the users' requirements description as a list of vulnerabilities. The extension also enables to define a complex parameter which is computed from basic metrics.

To describe the performance of an NIDS we used the C_{ID} as a parameter. C_{ID} meets both of our requirements: it is a single metrics and it takes the base rate into account. Taking the base rate into consideration in SLA comes with two challenges, (i) at what base rate value should a provider offer an SLA as the C_{ID} varies depending on the input base rate ? (ii) in practice measuring the value of the base rate before the occurrence of an attack is very difficult.

To address both of these issues, our SLO definition uses a *model* which takes the base rate as an input. Defining a model addresses both problems: first, it removes the need to use a specific base rate value in the SLA definition; second, the model can estimate the performance metrics for previously unknown base rate values. The model takes a value of B as an input and outputs the TPR and FPR values. The model is generated by testing the NIDS using known base rate values. Using the results from such tests, and an interpolation-based method, we generate the expected model. The final model can be used as the SLO description in the SLA.

The other issue addressed in this chapter is the effect of monitoring multiple vulnerabilities with a single monitoring device. Increasing the number of vulnerabilities in an NIDS results in lower performance. A quantitative measure of the effect between vulnerabilities helps to estimate the performance of the NIDS while aggregating those vulnerabilities together. We introduced the *interference vector and matrix* to describe the effect of vulnerabilities between each other. Using the interference values and the performance for individual vulnerabilities, we presented how to calculate the aggregated metrics. These metrics describe the performance of an NIDS while configured with all the vulnerabilities together.

Having a formal language and the performance estimation method for any group of vulnerabilities can be enough to prepare SLA templates. However, having thousands of vulnerabilities makes it impractical to build the interference matrix. Hence it hinders a custom-tailored SLA template preparation. To address this issue, we proposed a clustering mechanism which groups the vulnerabilities based on some heuristics and performs the interference test per group. The performance of an NIDS on the group indicates the worst performance; hence it gives a lower bound for any subgroup of vulnerabilities from that group. By this approach, it is possible to reduce the dimensions of the interference matrix and to make it practical.

Finally, we showed an experimental evaluation on how to prepare the model that can be used in the security monitoring SLOs. In our experiment, we showed the feasibility of the metrics estimation process using an interpolation-based method. We also showed an example of the clustering method using the Snorts Registered rule set. The providers can perform clustering depending on the available resources. A tradeoff should be maintained between the size of the interference matrix and the number of vulnerabilities per group. Indeed, the performance of an NIDS on a group reduces as the number of vulnerabilities increases.

Chapter 5

Monitoring an SLA Describing the Performance of an NIDS

5.1 Introduction

In the previous chapter we presented the first step in the life-cycle of an SLA, i.e a mechanism used to define security monitoring SLAs. Specifically, an SLA guaranteeing the performance of an NIDS in the cloud is studied. In this chapter we present a monitoring mechanism to check the satisfaction of the SLO defined in the agreement. Note that there is another step between defining an SLA and monitoring it, that is the enforcement stage. As described in Chapter 3, we use very simple and straightforward SLA enforcement method in our work. In this chapter we present the enforcement mechanism that we use for SLOs related to NIDS performance. A more general enforcement mechanism is left as a future work.

Verifiability is one of the characteristics of SLAs that are described in Section 2.5.1. Assume a provider promises to deliver a service with some quantifiable expected performance, without any method to check this measurement one can quickly realize that the promise is useless. To remove any ambiguity in the process of checking SLA validity, the procedure of measuring *key performance indicators (KPIs)* should be clearly described. But in practice verifying an SLO is challenging due to different reasons. One of the main issues arises from the fact that in clouds, the responsibility on different tiers is shared between tenants and providers. The main challenges in SLA verification are discussed in Section 5.2.

SLA monitoring is a *process of continuously checking the validity of an SLA*. In theory performing a test as many times as possible could provide the most accurate information on the state of SLA. But in practice this may not be possible and usually the SLA includes the description of the monitoring mechanism with a schedule. Monitoring mechanisms are highly related to the properties addressed by the SLA. An SLA covering service availability does not have the same monitoring mechanisms as an SLA covering the performance of security monitoring devices. The schedule is also an important factor, specially if the verification process has an effect on the monitored service.

In general the SLA verification process should allow to compute relevant metrics and it should evaluate the NIDS in realistic operational environment (dynamic or on-line evaluation as described in Section 2.4.4). A realistic process gives more accurate results than simulation. The process should not be too invasive, i.e the effect of a verification process on a production environment should be as minimum as possible. Moreover, any participant in the agreement should be able to perform the verification process. Specially, as we are aiming to have a user-centric security monitoring service, users should be allowed to verify that the SLA is respected.

As for security monitoring SLAs, a realistic verification can be done by performing a dynamic attack campaign. It is a method where the test is done by injecting real attacks in a given configuration. Such a verification process comes with some drawbacks. Putting additional packets in the network introduces overhead on the performance of the production service. Hence, one of the goals while designing a verification method for security monitoring SLAs is to minimize the overhead in a production environment. More detailed drawbacks are presented in Section 5.8.

As described in Section 3.1, the other design goal is to have a transparent (non-secretive) mechanism that can be used by both parties. To achieve such verification mechanism the need for cooperation was also presented in Section 3.5.4. Different resources are owned by different entities hence to have a transparent process cooperation is necessary. Considering current SLAs, specially availability SLAs,

the trend is to outsource the monitoring task to third parties like Cloudstatus [53] and Site24x7 [174]. Alternatively, tenants perform SLA monitoring by using tools which are offered by their provider (e.g Amazon CloudWatch) but not intended for SLA monitoring.

The goal of monitoring an SLA is to detect SLA violation. An SLA is violated if the result of a verification does not match what is expected as defined in the SLA. Penalties are applied for any violation and this is a challenge for transparent and unbiased cooperation. Since a penalty is applied for SLA violation, participants have incentive not to violate SLAs. In our verification process we try to minimize the dependency of a participant on other participants.

It should also be noted that, verification and the template preparation processes are both based on a same method which consists in running attack campaigns and computing the resulting NIDS performance metrics. As presented in the previous chapter, in order to offer SLOs providers perform tests to measure how their security monitoring device performs on different sets of vulnerabilities. These tests follow the same procedure as the verification process. Hence, when we describe verification it also refers to the procedure to prepare SLO templates.

In the next section we describe a simplified SLA enforcement mechanism for NIDS. Section 5.2 presents a detailed description of the problems and challenges that are addressed in this chapter. A unique attack injection mechanism is presented in Section 5.4. Section 5.5 describes the process after an attack champaign is performed, i.e how the NIDS output is analyzed to compute the KPIs. Finally, we present an experimental evaluation of the proposed SLA verification method with correctness and security analysis.

5.1.1 Simplified SLA Enforcement Mechanisms for NIDS

SLA enforcement is one of the phases of SLA life-cycle, in this section we present the enforcement mechanism used in our work. One of the characteristics of SLA is *attainability.* Providers will not (should not) offer an SLA which is not achievable. Not achieving an objective has consequences, the consequences can be either tangible like paying penalties, intangible like getting bad reputation or both. Tenants receive only statement credits, which don't help to recoup any potential asset or revenue lost during an SLA violation. In either cases it is mostly "no-win (lose-lose)" situation for both tenants and providers.

As presented in the previous chapter, in the SLA definition phase tenants specify their requirements in terms of vulnerabilities. The ECSLA language allows describing vulnerabilities using three properties namely *ID, CVE* and *Description.* The *CVE* refers a unique ID in vulnerability database. We also know that providers offer security monitoring services for sets of known vulnerabilities, hence they have a knowledge base on the matching between NIDS rules and related vulnerabilities. Using this knowledge when users request for security monitoring of a given vulnerability, the corresponding rules are added to the NIDS.

This is a straightforward enforcement of security monitoring SLAs for NIDSs. However, by no means it is the optimal way, specially assuming a more complex security monitoring setup. For example if there is more than one NIDS (e.g distributed IDSs), the enforcement may not be as straightforward as in our case. In Chapter 7 SLA enforcement is described as one of the future works.

5.2 Problem Description and Challenges of SLA Verification

This section presents problems and challenges that are faced while performing SLA verification. SLAs in our context describe terms guarantying the performance of NIDSs. Some problem and challenge descriptions are more general and some are specific to security monitoring SLAs. We distinguish these two classes in this section.

5.2.1 General Problems

Negotiated agreements must be verified against their structure for fulfillments of SLOs. Violation of an SLO may happen intentionally on unintentionally. To give an unintentionally SLO violation example, in 2017 the Northern Virginia region of Amazon Simple Storage Service (S3) was disrupted [9]. It was caused by an administrator error in command line input which removed a larger set of servers than intended. Interdependency between different services caused the whole region to be out of service for more than four hours. Many services were affected including Amazon's S3 dashboard, GitHub, Adobe's cloud and many more. The incident resulted in an SLO violation, SLO was promised at 99.9% availability

which means around 48 minutes of outage per month was tolerated. Since four hours is greater than 48 minutes, it indicates an SLO violation. Hence, verification is not only to detect intentional violation but also unintentionally events that could potentially lead to SLO violation.

The main challenge in doing verification is related to the nature of cloud computing service models. In Section 2.1.3 we presented responsibilities of providers and tenants in managing various tiers of the cloud. The responsibilities differ from one model to another. In IaaS clouds providers are responsible up to the virtualization layer, VMs and everything inside VMs are under tenants' responsibility. Performing verification in such an environment includes checking status of a tier which does not belong to the verifier.

If we take an example, verifying system availability SLA by a tenant means checking whether the infrastructure providing the running environment for VM(s), an environment which is not controlled by the tenant, is correctly configured and responds on time. In some cases tenants may need outputs (logs) from the provider. For example, in case of security monitoring SLAs, where monitoring devices are controlled by providers, a tenant may require outputs of the monitoring devices, which is again not controlled by the tenant.

5.2.2 Specific Problems for Security Monitoring SLAs

The other challenge in security monitoring SLA verification is the lack of methodology to evaluate a security setup in the cloud. An efficient method to evaluate a given security monitoring setup (e.g as the one presented in Chapter 4) can be easily adapted to be used for SLA verification. As described in Section 2.4.4, there have been both on-line and off-line methodologies that are used to evaluate the performance of security monitoring tools and setups before the cloud. However, we lack such a method in the cloud, i.e a transparent method which can be used in context where there are different owners in the system tier and the system itself is dynamic.

The dynamic nature of cloud infrastructures is another challenge. Virtualized infrastructures are dynamic and malleable, since creation, deletion and migration of VMs are frequent. Security monitoring SLA verification must anticipate such changes. The occurrence of such events should not affect or the effect should be as minimum as possible. Moreover, automatic adaption to changes in the infrastructure can help to keep the verification process consistent.

In addition, the verification mechanism should allow to compute relevant metrics for the measured property. In general, the verification process for security monitoring SLAs should be able to take base rate into account and specifically in our case, it should allow to compute the C_{ID} metrics.

In summary, in this chapter we address the problem of evaluating security monitoring setups in the cloud environment. A process which can be used by either the provider or tenant to check the performance of configured NIDS is presented. It takes the base rate into account and can be used to calculate any metrics which are computed from the basics (TP, FP, TN, FN) and base rate. The process is applied to perform SLO verification by both parties. The same procedure is used by the providers for SLA template preparation.

5.3 Threat Model

In this section we present the threat model that is assumed while performing SLA verification. The threat model is a subset of the one described in the previous chapter, Section 3.4. We do not take into account hardware attacks. The cloud provider is assumed honest and the provider infrastructure is assumed not compromised. Thus attacks can only originate from tenant input to the cloud API, from virtual machines (VMs), and from outside the cloud infrastructure.

In particular no low-level network attacks can reach the virtual switches and change their behavior, since traffic from outside the cloud arrives as IP packets that are routed towards the VMs by an edge router, and traffic from VMs is encapsulated in virtual LANs.

It is important to note that the negotiation of SLAs covers only known types of attacks. Indeed, service providers will not commit for unknown types of attacks. Thus, only those known attacks are used in the verification process. The list of monitored attacks can be regularly updated in order to include newly discovered vulnerabilities and attacks but it may require renegotiating or restarting the process of SLA life-cycle as described in Section 4.7.4.

Note that although the provider is assumed honest, SLAs need to be verified because (i) by definition SLAs must be verifiable and (ii) an SLO violation may be unintended and result from provider-implemented heuristics being imperfect.

In the next section we present a description of the proposed SLO verification process.

5.4 SLO Verification Method

In this section we propose a verification method that both the cloud provider and tenant can use to verify security monitoring SLOs for an NIDS. In addition, service providers can use the same process to prepare SLA templates in a pre-negotiation phase as described in Chapter 4. We present first an architectural view of our method which contains different components to perform a dynamic attack injection campaign and second a unique attack injection algorithm which respects a given base rate. Before diving into our SLO verification process, we present the unit of measure considered in our verification process.

5.4.1 Unit of Measure

For NIDS the unit of measure indicates the granularity at which events are counted e.g packet or flow level. To perform KPI computation for a given security monitoring setup, after running an attack campaign, incidents in the setup are counted and a specially designed Metric Evaluator (described later in this section) is used to do the computation.

Our unit of measure in processing inputs for the Metric Evaluator and counting incidents in the setup is the packet. For NIDS the unit of measure is related to its functionality. For example, for Snort [13] the unit of analysis is a packet. In such case the base rate indicates the rate of malicious packets in a given input. For flow-based NIDSs (e.g Bro [210]) the base rate indicates the rate of malicious flows in an input.

5.4.2 Architecture

The verification mechanism performs attack campaigns against a given NIDS configuration. We set three main design goals while preparing the verification method. First, to make the verification process as realistic as possible the attack campaign should be done dynamically (See Section 2.4.4 for mechanisms to measure NIDSs). This helps to measure the performance of NIDS exactly as it behaves in a real occurrence of an attack.

The second goal while performing the verification is not to damage the production environment. Performing a dynamic attack campaign means injecting attacks into the production environment to observe the behavior of the configured NIDS. At the same time the attacks should not damage the production VMs.

Thirdly, the overhead in the environment should be minimized. Since the process injects additional packets to the environment, the overhead in the network as a result of verification can not be avoided but should be as minimum as possible.

As presented, the first two goals seems contradictory, i.e we want to inject real attacks but without damaging the production environment. To resolve these paradoxical goals we use a *target virtual machine*, which is a properly crafted VM and located in strategic position to be used as a target for the injected attacks. Target VMs are described later in this section.

An example of the attack running environment is shown in Figure 5.1. The production infrastructure is composed of tenant production Virtual Machines (VM1, VM2, and VM3), hosted on cloud compute nodes (physical servers), and connected to virtual switches. An NIDS is connected to a mirroring port of the virtual switches and analyzes all packets passing through the virtual switches. Given such an environment to evaluate the performance of NIDS, we extend the infrastructure with two components:

- An *Attack Injector*, that injects traffic, containing attacks, to be analyzed by the NIDS,
- A *Target Virtual Machine*, to which the injected traffic is redirected, and that exhibits the network behavior of the production VMs

Attack Injector

The Attack Injector is a physical or virtual machine located inside or outside the cloud network and it is used to simulate an attacker. The Attack Injector must be able to reach the Target VM, which behaves as the production VMs. The switch which connects VMs in a tenant environment is configured to forward all incoming packets from the Attack Injector to the Target VM. Hence, injected attacks should

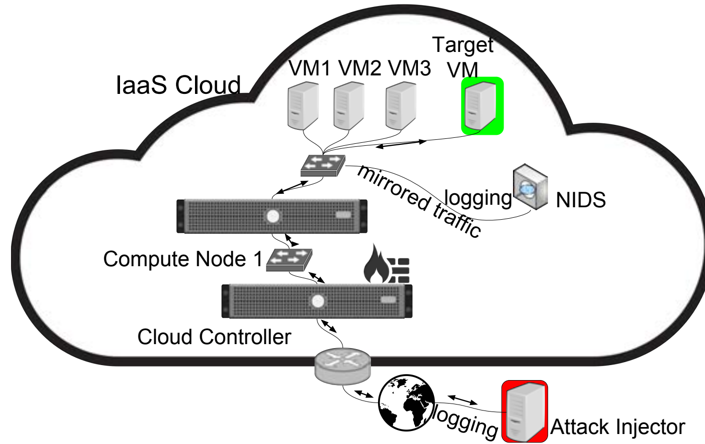


Figure 5.1: Example attack running setup

not affect services running on production VMs. In practice, if the switches are controlled by service providers tenants can not access and configure the switch. Thus, cooperation between the provider and tenant is required at this step. Providers are responsible to configure the switches.

As presented in Chapter 4, we use C_{ID} as a KPI to describe the NIDS performance and the C_{ID} value depends on the base rate during the attack campaign phase. However, in real production environments the base rate is likely to be very low (i.e attack packets arrive very rarely compared to legitimate packets) and it is impossible to measure its value in advance before the attack happens. Hence, the base rate used in the verification phase should be based on trade-offs between statistically observed data (as described in Section 4.5), performance (see Section 5.7.2), and accuracy when inferring KPI values based on a realistic base rate from verified values.

The novelty of the traffic injection algorithm we propose is to dynamically control the base rate. The algorithm takes three inputs: a set of attacks, a set of legitimate requests, and a target base rate. The base rate in the attack campaign is dynamically controlled using the knowledge of the number of packets sent by each attack and each legitimate request. The algorithm runs two loops with two different sets of processes in parallel. The first group injects an attack and the second group sends legitimate requests with some distributed inter-arrival times. The attacks and legitimate requests inter-arrival time distributions are selected to achieve the target base rate. Section 5.4.3 describes the attack injection algorithm.

Target VM

The Target VM is used as a victim machine to which the verification traffic is redirected. This machine simulates the behavior of production VMs (i.e services running in the production VMs). Specifically target VM simulates the network behavior of production VMs. In our SLA verification process we are interested in the network behavior of an application, we are not interested in the data processed by the production service. Any mechanism that can simulate the network behavior of an application could be used in our methodology. In [180] a finite-state automaton is used to model the network exchanges of an application with a legitimate and a malicious user. It should be noted that more realistic simulations produce more accurate results.

One way to simulate the production services in the target VM is by running the same software on the target VM as in the production VMs. This task (installing the software) is done once while preparing the target VM, which is performed in the SLA enforcement phase. It is not tedious to do it in practice. As long as there are no changes or updates on software in production VMs, the target VM should be usable. In some cases, one target machine may not be enough to simulate the behavior of production services. For example if different operating systems are running in the production environment, it may not be possible to simulate both at the same time using one machine. Multiple target VMs could be used in case a single VM is unable to exhibit all the required behaviors.

Attack campaigns are performed using the setup and algorithm described in the next section. Let's recall that the attack is injected from an attacker machine to a target VM. While performing the attack

campaign all the communications between the attack injector and target VM are recorded. i.e. all the requests (both attack and legitimate) are recorded. This is necessary to compute performance metrics in the next step. In addition, when the attack campaign is finished an output is generated from the NIDS. Section 5.5 presents how to compute the required metrics using this information.

5.4.3 Attack Injection Algorithm

In this section we present the attack injection algorithm used to perform attack campaigns. Algorithm 1 shows the pseudo code of the main function for the attack injecting process. The function takes a set of legitimate requests, a set of attacks, a target base rate and an integer value variable ("np") representing the number of processes in a group. The value of "np" should be selected considering available resources for the attack campaign. A lager value of "np" helps to run the verification in a short time but it also increases the overhead on the network. We present the effect of different values of "np" in Section 5.7.2.

Both attacks and legitimate requests are given with the number of packets that they generate, i.e when a legitimate or attack request is sent the algorithm knows how many packets are sent by that request. This is required, because the unit of analysis for the NIDS used in our experiment is the packet. See Section 5.4.1 for more detail about the unit of analysis.

The algorithm runs two sets of processes, each containing "np" number of processes ($np \geq 1$). In the main function "np" number of processes are started (lines 2-4 and lines 5-7) and tasks are assigned to each process. For the first group of processes (lines 2-4) they are assigned the task of sending legitimate requests and the second group (lines 5-7) are assigned the task of injecting attacks. The main function waits until the attack injecting processes finish their task.

Algorithm 1: Main function for attack injection

Input: attacks and legitimate requests with number of packets, base rate (B), request sending interval (r), number of processes (np), pool size for legitimate requests (LPS) and pool size for attacks (APS)

Result: send both legitimate requests and attacks with an average attack packet rate of B

```
1  Function main (attacks, legitimate, B)
       /* 'np', number of processes that are in one group and running in parallel.
          Start 'np' processes to send legitimate requests                      */
2      for p ← 0 to np do
3          start a process with (task=Algorithm 2, input=(legitimate requests, target IP, LPS, r));
4      end
       /* start 'np' processes to send attack requests                          */
5      for p ← 0 to np do
6          start a process with (task=Algorithm 3, input=(attack requests, target IP, B, APS,
           rounds, r));
7      end
8      wait for all child processes running attacks to exit ;
9      exit ;
```

Algorithm 2 shows the pseudo code for legitimate request execution. Legitimate requests are sent following a uniform distribution. A process is selected from a pool of processes and a randomly selected legitimate request is sent. Then the next request is sent after the specified waiting time. The requests are running asynchronously, hence the second process doesn't necessarily wait for the previous process to finish. The actual waiting time is related with the number of processes in the pool and maximum time required for a request. If the pool size is not selected carefully, there is a chance to have a scarce of processes in the pool which results from requests taking a longer time than the waiting time. In our experiment since the requests take less than one second, we used a waiting time between (0,1] second.

Each process in the group of legitimate requests follows Algorithm 2. Each process has a pool of processes to execute tasks asynchronously. The pool size is related to the maximum time required to send a request and interval at which requests are sent. There should not be scarce of process in the pool when it is the time to send a request. The algorithm first initializes processes in the pool (line 2) and schedules the next request by setting a timer to raise an alert after some time "t" (line 8-9). Then the algorithm selects randomly one legitimate request, it executes the request using one process from the pool and it logs data about the request (line 10-12). Finally, the algorithm sets a listener to capture the

scheduled alert and wait for the alert to execute the next round (line 3-5). The main process waits and executes *sendLegitimate()* whenever an alert is raised. It terminates when the parent process exits.

```
Algorithm 2: Legitimate request sending algorithm
   Input: legitimate requests, target IP, pool size for legitimate requests (LPS), r
   Result: send requests
 1 initialize a pool containing LPS number of processes;
   /* to be executed when a signal is received                                  */
 2 set signal handler with function sendLegitimate ;
 3 sendLegitimate (legitimate requests, target IP);
 4 while True do
   |  /* this process wakes up when the signal is fired                         */
 5 |  sleep()
 6 end
 7 Function sendLegitimate (legitimate requests, target IP)
   |  /* Schedule the next request                                              */
 8 |  next = (select random number between (1, r))/ r;
 9 |  set a timer to raise a signal after next seconds;
   |  /* Send request of this round                                             */
10 |  select randomly one request from legitimate requests;
11 |  execute the request asynchronously with one process from the pool;
12 |  log data (request sent and time);
```

The second group of processes is used to send attack requests. Algorithm 3 shows the pseudo code for sending attack requests. It takes the base rate, pool size and counter variable as an input. The counter variable is to control the number of rounds for attack injection. The value of this counter variable needs to be set taking available resources into account. For higher value better results are obtained, i.e an increased number of rounds gives more statistically sound results.

After initializing the process pool and counter variable (line 1- 2), the algorithm computes the average number of packet ratio between legitimate and attack packets (line 3). Attacks and legitimate requests are given with their corresponding number of packets and these values are used to compute the *"packet ratio"* between legitimate and attack requests. To get the *packet ratio* in an attack campaign, we first compute the average number of packets for both attacks and legitimate requests and then compute the ratio between the two average values, i.e

$$packet\ ratio = \frac{average\ number\ of\ legitimate\ packets}{average\ number\ of\ attack\ packets} \tag{5.1}$$

where *average number of legitimate packets* and *average number of attack packets* are computed based on the prior information about the legitimate requests and attacks used in the experiment. For example, let us assume we have three legitimate requests (*LR1, LR2*, and *LR3*) and three attacks (*A1, A2*, and *A3*) to be used in our experiment. With a few experiments we can know the average number of packets sent by each of the legitimate requests and attacks. Let (*lr1, lr2* and *lr3*) and (*a1, a2* and *a3*) represent the average number of packets sent by each of the legitimate requests and attacks respectively. The *average number of legitimate packets* and *average number of attack packets* are computed respectively as $\frac{lr1+lr2+lr3}{3}$ and $\frac{a1+a2+a3}{3}$.

Before executing an attack for the current round,the algorithm schedule the next round by setting an *"alert"* to be raised after time *"t"* (line 15 - 17). The time *"t"* is related to the base rate and the packet ratio computed previously. A random waiting time is selected as described in the previous algorithm. For the attack injection algorithm the goal is to respect a given base rate (B) value. For simplicity let us assume the packet ratio is 1, i.e the average number of legitimate and attack packets are the same. In that case, the random value is normalized by the base rate value. Assuming a packet ratio different from 1, the normalizing factor becomes base rate times packet ratio (line 16).

After setting an *"alert"* to be raised after time "*t*", the algorithm executes an attack for this round, logs data about the injected attack and increments the counter (line 18 - 21). Each process keeps its own counter variable. These procedures are repeated until the required amount of rounds are reached. If the number of attacks reached the expected rounds, the algorithm closes the pool not to instantiate any new job and waits for other attack processes to finish. Finally, the attack sending process exits and

returns to the parent process. The main process forces the other group of processes (legitimate injecting processes) to be halted and finish the campaign.

Algorithm 3: Attack injection algorithm

```
   Input: attack requests, target IP, Set base rate (B), pool size for attacks (APS), rounds, r
   Result: send attack requests
 1 initialize a pool containing APS number of processes;
 2 count ← counter for the number of rounds;
   /* Compute the average number of packet ratio between legitimate and attack
      campaign                                                                   */
 3 packet_ratio ← average number of legitimate packets / average number of attack packets ;
 4 set signal handler with function sendAttack ;
 5 sendAttack(attack requests, target IP);
 6 while True do
       /* this process wakes up when the signal is fired                         */
 7     sleep();
 8 end
 9 Function sendAttack (attack requests, target IP)
10     continue = True;
11     if count > rounds then
12         continue = False ;
13     else
14         rn = (select random number between (1, r))/ r ;
15         next = rn/(B * packet_ratio);
16         set a timer to raise a signal after next seconds;
17         select randomly one request from attacks;
18         execute the attack asynchronously with one process from the pool;
19         log data (request sent and time);
20         count = count + 1;
21         if count > rounds then
22             continue = False ;
               /* tell the pool not to accept any new job and wait other processes to
                  finish                                                         */
23             close the pool;
24     end
25     if not continue then
26         exit();
27     end
```

5.5 KPI Computation

The goal of KPI computation process is to compute the C_{ID} metrics using NIDS outputs and injected traffics. To calculate C_{ID} we need (TP, FP, TN, FN) and base rate values. Hence, first we need to count the detected and missed attacks (i.e TP and FN) by comparing the inputs and the outputs from the NIDS. At this stage, we have a predefined set of attacks and legitimate requests with their corresponding number of packets sent, target base rate used in the attack campaign, dumped packets from the communications between the Attack Injector and the Target VM and events list from the NIDS output.

Using the available information we can compute any metrics which is based on (TP, FP, TN, FN) and base rate. In our case we use this information to compute the NIDS performance in terms of the C_{ID} metrics. The comparison between inputs to the attack injection algorithm and the output of NIDS helps to see whether the NIDS detected attacks sent from the Attack Injector and to differentiate between true and false positives. Figure 5.2 shows the architecture of the *Metrics Evaluator*.

To see whether the configured NIDS detected an injected attack, we need to compare packets from the attack campaign with the output of the NIDS. We have to match packets associated to alerts in the output of the NIDS with packets from the communications between the Attack Injector and Target VM. The output of an NIDS is a set of events which are triggered when packets match at least one rule in the

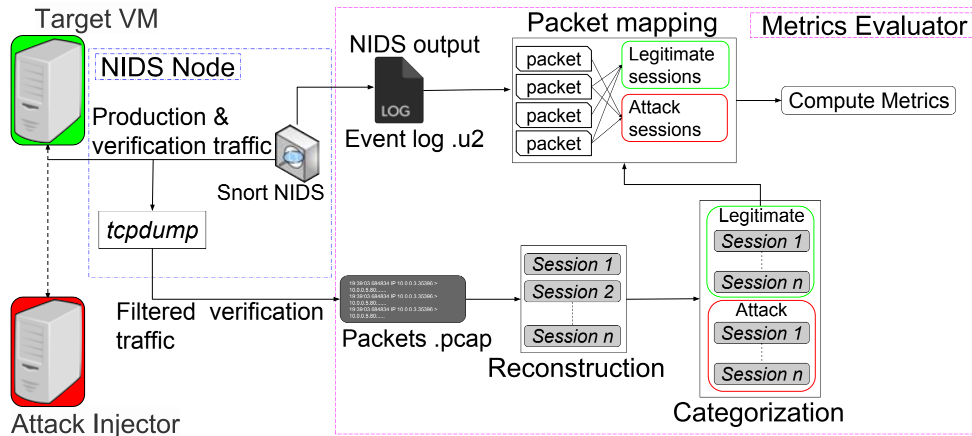


Figure 5.2: Architecture for metrics evaluation

NIDS rule set or when monitored traffic exhibit abnormal behavior in the case of anomaly-based NIDSs. An event includes an event ID, source and destination IPs and ports, a time stamp, a set of packets which triggered the event and (if applicable) the rule which generated the event. If an injected attack packet matches with packets from the NIDS output event list it indicates correct detection of an attack (TP). On the other hand if the legitimate input matches with packets from the NIDS output event list it indicates false positives (FP).

Figure 5.2 shows the communication between the Attack Injector (machine with a red background) and Target VM (machine with a green background) including mirrored traffic to an NIDS. The communication between the Attack Injector and Target VM is recorded in a location close to NIDS. Being as close as possible to the NIDS helps to have the same set of packets that are processed by the NIDS and see the same level of packet fragmentation. The Metric Evaluator takes the recorded packets and the output of the NIDS as an input.

The computation starts by reconstructing sessions from the dumped set of packets. Since the recorded events are bunches of packets it is required to organize them in order to identify which packet belongs to which request. Then sessions are categorized into legitimate and attack sessions. This categorization is based on the knowledge about the injected packets. From the event list (i.e output of NIDS) packets are extracted and used in the mapping stage. Packets from the events are mapped to the constructed sessions to get the values of TP and FP. If a packet matches a legitimate session then it indicates the NIDS wrongly classified a legitimate request as an attack, which is FP. On the other hand, if a packet matches an attack session then it indicates the NIDS detected an injected attack, i.e TP. This mapping shows specifically which type of attack or legitimate request triggered the events, hence we can count the values of FP and TP.

Since we have the knowledge of the total number of attacks and legitimate requests that are injected we can compute the remaining basic metrics, TN and FN. Note that the verification method does not depend on the NIDS software used. Our method can be adapted to different types of NIDS by adapting the parser of the NIDS output.

We now detail each metrics computation step.

- *Dumping Packets*: The NIDS is connected to a location where it is able to see every packet passing through the monitored environment. We use that link to record the communication between the Attack Injector and Target VM. Dumping from such a link allows seeing all packets entering in the NIDS. As we are in the production environment the link may contain packets from the production traffic. For our use case, we want to reconstruct sessions only between the Attack Injector and the Target VM, we use filters to separate those packets.

- *Session Reconstruction*: The output from the previous step is a list of packets. In order to differentiate between legitimate and attack packets from this list we re-assemble the packets in network sessions, which are easier to match with the traffic that is injected.

- *Session Categorization*: Once the packets are assembled into sessions, using our prior knowledge about the injected traffic, we can categorize them into their respective types, i.e either legitimate or

attack. To this end, since we know the requests prior to injection, we use some of their characteristic attributes like port numbers used, packet payload and number of packets per session.

- *Packet Mapping* This is the last step before doing the actual metrics computation. Packets from the NIDS output are mapped to the sessions from the previous step. If a packet matches with an attack session it indicates that the IDS detected that attack. On the other hand if it maps to a legitimate session, the NIDS issued a false positive. From this we can calculate the different metrics described in Section 2.4.4 including the C_{ID}.

The required metrics can be computed following this procedure. The Metric Evaluator can be located anywhere inside or outside the cloud hence, can be used by either tenants or providers. However, cooperation is still required at this stage, the NIDS is controlled by the provider. The output of the NIDS should be provided by the provider to tenants. Section 5.7.3 discusses more about cooperation between the provider and tenants.

In the next section we present tools and technologies used to implement the attack injection campaign and Metrics Evaluator.

5.6 Implementation

In this section we present tools that are used to implement what is described in the previous sections. A prototype of the attack injection algorithm is implemented in Python, using the "*multiprocess*" module to implement functionalities related to parallel processing, like parallel execution and pool of processes. We used signals and signal handlers from "*signal*" module to raise scheduled alerts. In the attack injection algorithms (Algorithm 2 and 3) after calculating the waiting time "t", a signal is scheduled to be fired after "t" amount of time and a pre-configured handler starts the next task when receiving the signal. For the attack campaign we used the attack and legitimate requests listed in Table 5.1. The services used in our setup are the same as the ones listed in the previous chapter Section 4.3.2, Table 4.1.

Both legitimate and attack requests are implemented using Python. But some attack requests are dependent on other tools. For example the DDOS attack script uses *Torshammer* [211], the port scanning one uses *Nmap* [159] and SQL injection uses *Metasploit* [145] in the back. All are wrapped using Python for consistent automation.

The Metric Evaluator is implemented using different tools. Dumping packets was performed using *tcpdump* [209] on the machine where the NIDS is running. We used tcpdump's filtering features to dump only the communications between the Attack Injector and the Target VM.

We used *tcpflow* [87] to perform session reconstruction. It reconstructs the data stream and stores each session in a separate file. Tcpflow understands sequence numbers and correctly reconstructs regardless of retransmissions or out-of-order delivery. However, tcpflow does not handle neither IP fragments nor protocols other than TCP. Both *tcpdump* and *tcpflow* are selected for ease of use, similar functionalities can be achieved using other tools. Section 5.7.3 describes the drawbacks of these softwares and presents alternative tools.

Snort [13] is used as NIDS in our implementation. The NIDS can be configured to give output in different formats but the binary event list (*unified2* format) is the most efficient way for the NIDS to generate the outputs. For example Snort [13] can be used with different output modules. In our implementation Snort is configured to give output in the Unified2 format. To extract packets and related information the binary file needs to be parsed. For this purpose we used a tool called *u2spewfoo* [1] which can dump the contents of Unified2 log files to stdout.

Categorization of sessions and mapping of packets to sessions is performed using tools that we developed. In addition, a tool to compute C_{ID} value was developed in Java.

In the next section we present an evaluation of the proposed SLO verification method. We show experimental results with correctness, usefulness and security analysis.

5.7 Evaluation

In this section we present the experimental evaluation and analysis of correctness, usefulness and security for the proposed verification method. In the experiment we aim to measure the effect of verification on a production network infrastructure and to study trade-offs between verification time and resource usage.

To do the evaluation we configured a cloud environment with different services and an NIDS to monitor specific vulnerabilities. We injected real attacks to measure the performance of the configured NIDS using the method presented in previous sections.

First we present the setup used to run experiments. Then we present observed results from the experiment. Finally, we discuss the correctness, usefulness and security analysis.

5.7.1 Experimental Setup

We set up an infrastructure as shown in Figure 5.1 on the Grid'5000 [22] testbed. Each physical machine (node thereafter) featured two Intel Xeon X5570 processors (6M Cache, 2.93 GHz, 4 cores) with 24GB memory. We used OpenStack [163], a mainstream IaaS cloud management system, with one controller and one compute node. Open vSwitch [165] is used as a virtual switch in our cloud setup. The NIDS is hosted on a third node and is connected to the virtual switches mirroring ports through generic routing encapsulation (GRE) tunnels. The Attack Injector is hosted on a fourth node outside the cloud network.

Three production VMs ran respectively a web server, a database server and a content management server, as shown in the previous Chapter, Section 4.3.2, Table 4.1. We added a Target VM that exhibits the behavior of production VMs by running all three services. All VMs ran Ubuntu server 14.04 with the OpenStack m1.medium flavor (2 VCPUs, 4GB memory and 40GB disk).

For this evaluation we used the example SLA presented in the previous Chapter, Section 4.4.3. To this end we collected the vulnerable applications and collected and/or developed the attacks (listed in Table 5.1) to exploit vulnerabilities listed in the example SLA. As mentioned in Section 5.4.2 attacks are injected in parallel with legitimate requests. To this end, we collected a set of legitimate requests: loading a web page, login into Mysql, uploading a file and login into a WordPress account.

Finally, we used the Snort [13] mainstream open source network IDS. We collected or wrote Snort rules for the attacks listed in Table 5.1.

5.7.2 Performance Impact

Since our SLA verification method runs in the production network infrastructure but does not involve production VMs, the most expected performance impact is network overhead. There is no CPU or memory overhead related to the production environment because the attack campaign phase only uses the production network infrastructure.The metric computation process can be performed outside of the cloud.

The network overhead is measured as the difference in response time of a given request to a production VM with and without SLA verification running in parallel. First a request is sent and the response time is measured. Then the verification process starts and the same request is sent while the verification is ongoing. The difference in response time with and without the verification process running shows the overhead in the network as a result of SLA verification.

Figure 5.3 shows a time line on how the experiment is performed with response time in y-axis. To measure the network overhead, the experiment started by simulating production traffic at time t_0. The SLA verification started later at time t_1 and finished at time t_2. The production traffic then continued until time t_3 ($t_0 < t_1 < t_2 < t_3$). This way we can compare the response time in different phases. From t_0 to t_1 the network contains only production traffic. From t_1 to t_2 the network contains both production and verification traffic. Finally, from t_2 to t_3 it goes back to the first state (i.e only production traffic). The difference in response time between "at verification" and "before verification" phases show the network overhead introduced by the verification process. This experiment was performed using base rate values $B = 10^{-2}$ and $B = 10^{-3}$.

Table 5.1 shows the average number of attacks and legitimate requests injected over ten rounds using base rate value $B = 10^{-2}$. As described in Section 5.4 there are multiple processes in each group (legitimate sending group and attack injecting group). For this experiment both attack and legitimate request injectors used three processes each to send interlaced verification traffic. Moreover, each process has a pool with two and three processes for attack and legitimate requests respectively. Each attack injection process continues until 20 injection rounds, i.e a total of 60 attacks are injected in our experiment. As presented in Section 5.4.3, we used a waiting time between (0,1] second.

The box plot in Figure 5.4 shows the mean response time for each type of legitimate request in the three time intervals: "Before" verification, "At verification", and "After" verification. The plot shows that the overhead is very small relative to the time needed for a request. We observe a maximum overhead of 6.9% (0.000165 second) in the case of "SQL Login" requests and a minimum of 2.2% (0.001755 second)

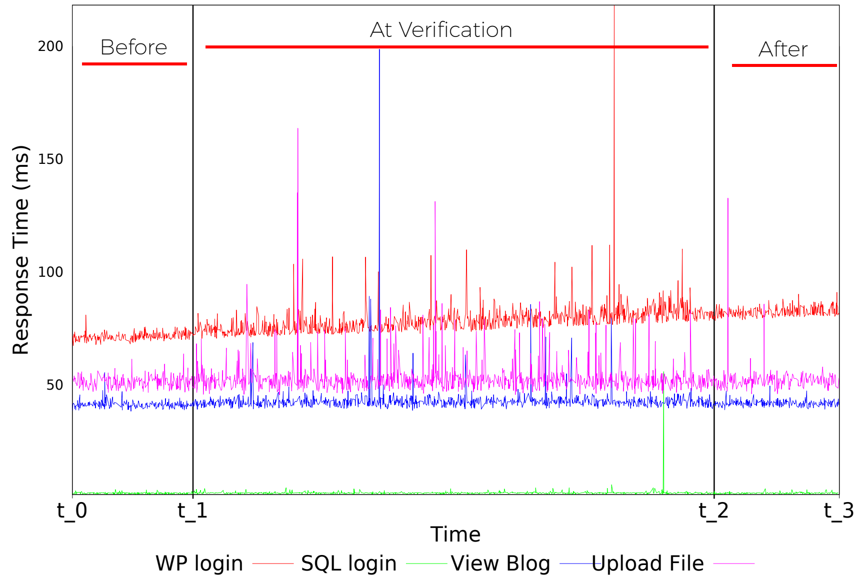


Figure 5.3: Verification time line

in the case of "WordPress Login" requests. The third box (after verification) plot for "WordPress Login" requests shows an increase in response time even after the verification (observed also in Figure 5.3, the red plot after time "t_2"). This behavior is observed even without the verification process. Overall in this experiment, the most observable impact of verification seems to be an increased dispersion of response times.

Another important factor while performing verification is the base rate. It determines the time required for verification. The experiment to produce the graph shown in Figure 5.4 took 41.6 minutes using a base rate $B = 10^{-2}$. We also did an experiment with $B = 10^{-3}$, which showed the same overhead but took a much longer time (around 4 hours). This duration is expected because as the base rate decreases, the rate of attack injection declines and it requires a longer time to get enough attack samples to compute accurate statistics.

Decreasing the verification time is possible by increasing both the attack and legitimate traffic injection rate but the overhead on the production environment increases. Figure 5.5 shows the time needed in further experiments to perform verification using different traffic injection rates. The injection rate can be altered by increasing (decreasing) the number of parallel processes. We performed three experiments using respectively 3, 12 and 30 processes for both attack and legitimate traffic. It took 41.6, 12.5 and 2.45 minutes respectively, while the overhead increased for each case respectively. In other terms there is a trade-off between the time required to perform the verification and the overhead on the production environment.

5.7.3 Correctness, Usefulness and Security Analysis

In this section, we first show the correctness of our verification process, then its usefulness from tenants and providers perspective and finally we present a detailed security analysis.

#S = average number of injected flows (attacks or legitimate requests).
#P = average number of packets per flow.

Attacks	#S	#P
Denial of Service (DoS)	12	1916
Port scan	12	1012
Brute force access	12	2000
XSS	13	6
SQL injection	12	25

Legitimate request	#S	#P
View Blog	2482	6
WP login	2498	12
Mysql login	2479	6
Upload file	2501	3111

Table 5.1: Traffic injected using $B = 10^{-2}$ and parallelism degree of 3.

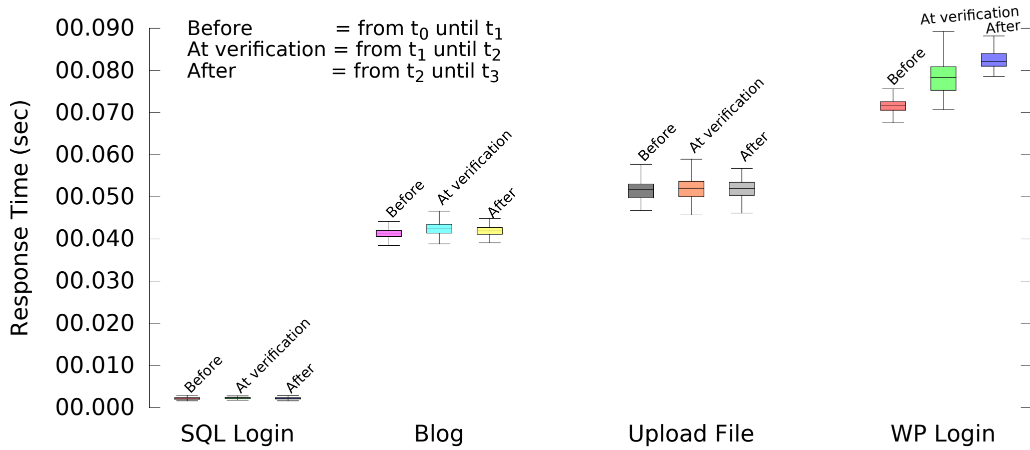


Figure 5.4: Impact of verification on the mean response time for the four types of legitimate requests

Correctness Analysis

We present the correctness analysis in two parts, first the correctness of our attack injection algorithm in reaching the expected base rate and second the overall correctness of the verification process by comparing the expected C_{ID} values with the results obtained from experiments.

Figure 5.6 shows the base rate value throughout the attack campaign. The given target base rate is $B = 10^{-2}$. Initially, at the start of attack injection the base rate is high, i.e the ratio of attack to legitimate packet is high. This is intuitive as there are not enough legitimate packets in the network at that time. The algorithm quickly adjusts the ratio of attack packets in the network to the expected base rate value (the base rate reaches the target in the boundary of $\pm 10^{-3}$). We get the most accurate approximation when running longer experiments which gives enough points for the statistics, as described in Section 5.4.3

To show the correctness of our verification process, we first manually compute the expected C_{ID} value using knowledge from the NIDS configuration, then we run the verification mechanism and check if the result matches the expected value. In the experiment Snort is configured with around 274 rules to detect the attacks listed in Table 5.1. The table also lists the actual number of attacks and legitimate requests injected in our experiment using base rate $B = 10^{-2}$.

We expect Snort to give alerts for almost all true attacks and very few (close to zero) false positives. This is a result from carefully crafted rules in the experiment setup, which simulates a real production environment. It should be noted that, in a real production environment, achieving such a low false positive rate is difficult. In our experiment we observed an average packet drop rate from Snort of 2.95%

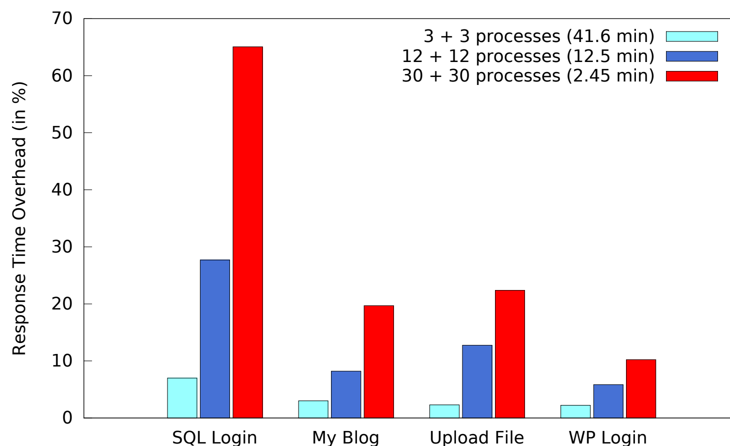


Figure 5.5: Response time overhead as a result of verification, with $B = 10^{-2}$, the four types of legitimate request and increasing traffic injection rates

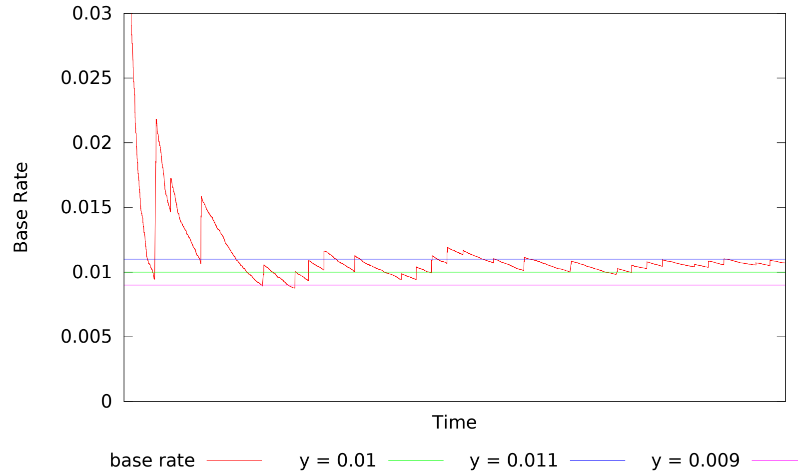


Figure 5.6: The rate of attack packets (base rate) in the network throughout the experiment targeting a base rate value of 10^{-2}

over ten rounds. Consequently, we expect false negatives ($FNR \approx 0.03$) as some attack packets are likely not processed by Snort. As a result the expected C_{ID} value should be around 0.95.

Using the data in Table 5.1, the output of Snort and the packets recorded from the attack campaign, we computed the C_{ID} value as shown in Figure 5.2. As expected the metrics evaluator output values of $TPR = 1.0$ and $FPR = 0$, i.e. all processed attacks are detected and no false alarm is generated. We have $FNR = 0.07$ and $C_{ID} = 0.90$. In our experimentation environment the verification process thus shows a very small and acceptable difference from the expected FNR value.

Usefulness Analysis

Having a transparent SLA verification mechanism, as described in this chapter, is useful for both service providers and tenants. Tenants should trust more services they consume as such SLAs guarantee compensation in case of SLO violation. For providers, in addition to increasing their clients trust, providing a guaranteed security monitoring service should give economic advantage either directly by providing a paid service model or indirectly by attracting new customers, who were previously not confident enough with the security of clouds.

Specifically, the verification process presented in this chapter allows both parties to perform verification with minimum overhead. Depending on available resources the time to do such a verification can be minimized. The proposed method requires configuring a target VM only once and preparing requests to be used for verification in cooperation with the service provider.

Security Analysis

In this section we provide an analysis of the proposed method from a security perspective. We discuss the innocuousness of the method for both tenants and providers, the factors that could affect the outcome of the process and the types of attacks that can be used for the proposed method.

Innocuousness In our verification mechanism the Target VM is the only added component intervening in the production environment. The other components (Attack Injector and Metrics Evaluator) could be located outside the production infrastructure. The Target VM is carefully prepared to simulate the behavior of production VMs and is targeted by attacks which are detected by the network IDS, i.e. attacks like cross-VM side-channel [226] (attacks that could affect other VMs in the same host) and undetectable IDS evasions [181] should not be injected in this method.

Attack packets injected into the production network are no real threats because the virtual switch is configured to forward all incoming packets from the Attack Injector to the Target VM. This guarantees that attack packets injected to the network do not reach production VMs. To configure the switch tenants must cooperate with service providers since, tenants should have no direct control on a switch shared between virtual networks from different tenants. This cooperation should be incited by the economic advantage brought by providing guaranteed security monitoring.

As a requirement for security monitoring SLA terms, tenants are required to provide lists of services to be monitored. This disclosure of services should not be a security concern for two reasons. First as we assume the provider is trustworthy, confidentiality of tenants should be respected. Second our methodology does not require knowledge about the actual data processed by the services, we are interested in the network behavior of tenants environment.

Robustness Packets from the Attack Injector are filtered and dumped, from the same virtual switch mirroring port the NIDS is attached to, for later analysis. In a network where there is very high traffic tcpdump may not perform as expected. Packet drops in this phase may affect subsequent steps. In particular if tcpdump is writing to disk the speed and disk space should be taken into account. Writing to a disk may decrease the speed and lead to packet loss. Other tools like Moloch [152] could be used to overcome such problems.

We used tcpflow to reconstruct sessions from the output of tcpdump. But tcpflow does not understand IP fragments, that is flows containing IP fragments will not be recorded properly. In cases where the NIDS and tcpflow use different IP fragments handling strategy, the reconstruction phase would also be altered. Thus attacks exploiting IP fragmentation could lead to errors in the packet mapping phase. Other tools like Wireshark [223] could be used to overcome this issue.

If there is any external TCP session (communication which is not between the Attack Injector and the Target VM) in the tcpdump output, it will be reported in the categorization phase. This could happen as a result of erroneous filtering. Getting a higher percentage of uncategorized sessions would imply poor filtering and/or reconstruction and it could lead to a false result. In such cases the verifier should observe and decide whether to continue or redo the previous steps.

Attack Coverage Finally, if an attacker performs IP spoofing types of attacks [43] to send attack packets impersonating verification traffic, this could be dumped as a communication between the Attack Injector and Target VM. This would disrupt session reconstruction and metrics evaluation afterwards. In addition, occurrences of attacks targeting the NIDS [181] during the verification phase could prevent the NIDS from performing as expected and would falsify the final result. To prevent such uncontrollable errors choosing appropriate times for verification could help, for instance during the most idle times of the production environment.

It should be noted that our solution is not "bullet proof", hence before summarizing the chapter we present shortcomings of the proposed method.

5.8 Shortcomings of the Proposed SLO Verification Method

We have explained that taking dynamicity into account is one of the design goals. The verification process presented in this chapter only takes partial dynamicity into account. The dynamic events in the cloud that may affect our verification method include VM creation, migration, and deletion. The verification process will not be affected by VM deletion. However, VM migration and VM creation affect the verification process, and protecting from the effects such events require automatic reconfiguration tasks. For example, creating a new VM (e.g. for load balancing) requires one additional step, to add IP addresses of the new VM to the monitoring list. Similarly, VM migration requires additional tasks to satisfy the required configuration as presented in Section 5.4. If automated self-adaptable method (e.g. as presented in [93]) is configured, events like addition and migration of VMs will not affect our verification process.

To perform experiments we used real attacks. But the attack set contains mostly application level attacks, network level DOS and port scanning. Even if these are the most common types of attacks, we did not use more sophisticated attacks, as explained in Section 5.7.3. The mechanism works on types of attacks that can be detected using NIDS, specifically signature-based NIDS. But not all attacks are detectable using this monitoring device (e.g. zero day vulnerabilities). Adapting the presented security monitoring SLA for other types of monitoring devices (e.g anomaly-based NIDS) is one of the future works described in Chapter 7.

The next section finalizes the chapter by providing a summary.

5.9 Summary

In this section we provide a summary for the SLA monitoring method proposed in this chapter. We started by describing what does SLA monitoring means and the need to have SLA verification mechanisms. By definition SLA should be verifiable and verification helps to detect violation of SLO by any party. It should be noted that violation can occur either deliberately or accidentally as a result of other actions. We stated the design goals needed to achieve a transparent SLA verification process: Having a mechanism which is transparent, which can be used by both parties and with minimum overhead on the production environment. In addition, the same mechanism can be used by service providers to prepare SLA templates.

We have described challenges that need to be addressed in order to reach the desired goal. Nature of the cloud including having different owners in different tiers and the dynamic nature of the infrastructure with lack of method to evaluate security monitoring setup are described as main challenges.

We proposed an in situ SLO verification method to measure the performance of an NIDS (expressed in terms of C_{ID}) dynamically using traffic injection. The injection process uses real attacks without damaging the production VMs. The novelty of our verification method is to improve the relevance of the obtained measures by two means: for the sake of trust we do not clone the production NIDS, and the base rate of the injected traffic is dynamically controlled. The experimental evaluation shows that during verification phases of 40 minutes a reasonable overhead (less than 10%) can be observed on production VMs response times. This overhead could be decreased at the price of longer verification phases.

In addition to experimental evaluations, we performed analytical evaluation of the verification method to show its feasibility in an IaaS production environment from the viewpoint of correctness, usefulness, and security. We showed the correctness of our algorithm in respecting the target base rate and correctnesses of the overall verification process by comparing manually computed, expected, value with experimental result. We showed the innocuousness of the proposed method and its usefulness for both tenants and providers.

Chapter 6

Monitoring an SLA on Data Integrity

6.1 Introduction

The work presented in this chapter was done during a six-month internship at Laurence Berkeley National Laboratory (LBNL). The internship focused on enhancing data integrity checking methods in a scientific environment. Specifically, we studied the advantages of using logical or physical secure components in the process of integrity checking for scientific data. In this chapter, we propose to adapt and use the internship work for the case of user-centric security monitoring SLAs in clouds.

In the previous chapters, we have seen one way of implementing user-centric security monitoring in the cloud. We used SLAs to include tenants in the processes of security monitoring. Specifically, allowing users to define which vulnerabilities to be monitored and check the validity of an SLO at any time (or according to a schedule defined in the agreement). In the process, we mentioned the need for cooperation between providers and tenants. In this section, we first recall the dependency between tenants and providers then we describe a proposed mechanism to reduce such dependency and finally we present the SLA that we consider in this chapter.

6.1.1 Dependency Between Tenants and Providers

Going back to the initial problem, the need to have security monitoring SLAs for clouds is because of the nature of its operation. In the cloud business model tenants outsource their information system and providers are in charge of monitoring the physical infrastructure including its security monitoring aspect. This scenario creates a trust issue between tenants and the provider, thus the need for an agreement.

As described in Chapters 3 and 5, SLOs must be verified for their fulfillment and any violation results in a penalty for the violating party. In the best case scenario, once an agreement is signed any participant should have the option to perform a test on the satisfaction of an objective independently of the other party(ies). Moreover, in case of violation, one party should be able to prove the violation for the others. As described in Chapter 2 currently detecting SLO violation is left as a responsibility for tenants. Even when tenants discover any violation, penalties do not apply automatically. Service providers perform checks on their side and the penalty is applied only if the provider discovers the violation. For example, Amazon SLA [5] describes the procedure to report a violation and states *"if the Monthly Uptime Percentage of such request is confirmed by us and is less than the Service Commitment, then we will issue the Service Credit to you ..."*.

Section 3.2.1 describes the need for cooperation between tenants and providers in order to perform end-to-end security monitoring. Chapter 5 also indicates without cooperation doing verification is difficult if not impossible. Tenants need to disclose the service(s) they are running and providers need to give an untampered output of security monitoring devices. In the context of SLAs, such dependency creates a conflict of interest and it requires trust from the other party, which goes back to the initial trust issue. Thus, we need a mechanism to reduce (remove if possible) such dependency in SLA verification.

6.1.2 Trusted Component in the SLA Verification Process

To remove such dependency on one party we need either a trusted third party or a trusted system where participants in the agreement get information and make decisions accordingly. Recently we have witnessed an increase in applications of distributed systems technology to make a secure system state

change between untrusted parties without using any third party. These technologies, referred to as *blockchains*, are widely adopted for their property of *tamper-proof evidence*. Blockchains are used as the core technology in digital currency (cryptocurrency) applications and a very wide range of applications ranging from the Internet of Things (IoT) to health, identity and security are being developed.

In this chapter, we provide a user-centric security monitoring SLA verification approach for clouds which relies on the blockchain technology. Users participate in the SLA life-cycle process. Specifically, they can perform verification at any given time. Besides, users directly participate in keeping and maintaining attestation data secure, data which will be used for verification. Users participation is possible as a result of a property of blockchains; by design, blockchains are distributed systems; i.e multiple nodes, ideally geographically distributed, form a network to build the blockchain. Tenants can be part of this network and participate in the SLA life-cycle process. In the method presented in this chapter tenants participate in the network with the provider to keep verification data secure, this allows trusting the system in general rather than a single entity. Depending on the algorithm used in the blockchain operation process, few numbers of participants can be malicious and the system can still be trusted.

6.1.3 Considered SLA

SLAs in this chapter are different from the ones defined in Chapters 3 to 5. Previously defined SLAs were guaranteeing the performance of security monitoring devices, specifically NIDSs. In this chapter, we define SLAs guaranteeing security properties of a given data stored in the cloud, specifically the *integrity* property of the data. Our goal in this chapter is to show that having a trusted component in the SLA life-cycle helps to facilitate and improve the trust level of different phases in the life-cycle of SLA. Even if we are using data integrity to show the use of trusted components in the SLA life-cycle, the method presented in this chapter can be adapted for other types of SLAs, including security monitoring SLAs as presented in Chapters 3 to 5. Section 6.8.3 describes using secure hardware components for security monitoring SLAs.

The problem discussed in Chapter 3 for IaaS clouds mostly applies to data in the cloud. There is a lack of security SLA for data stored in a cloud. This lack of security support has been a significant difficulty for the adoption of cloud services mainly for enterprises and cautious consumers. For example in Dropbox [52], a file hosting service, user metadata is stored in the company's data centers, while the actual files reside on Amazon's S3 storage service. Such a relationship between companies requires an agreement which covers the security aspect of the data. The same way as in IaaS clouds, we can address the security issues in outsourced data storage through SLAs.

Data integrity failure is a common issue in data storage systems [220, 75, 194]. There are different solutions to protect data from corruption and to recover from corruption after its occurrence. Additionally, in some fields of research and development, it is mandatory to keep data integrity and verify this property for others. For example, almost all regulatory agencies controlling medical drugs and health care products publish data integrity guidelines. Such government entities require implementing these guidelines in testing, manufacturing, packaging, distribution, and monitoring of drugs to review the quality, safety, and efficiency of the products.

There are different reports of data integrity failure in the real world IT production environments. In 2009 Facebook temporarily lost more than 10% of photos in hard drive failures [75]. Amazon S3 suffered from a data corruption issue [194] caused by a load balancer which was corrupting single bytes in the byte stream. As a result, uploaded objects processed through that malicious load balancer did not match the MD5 hash values supplied by the user. In [220] the authors examined 138 data corruption incidents reported in the bug repositories of four Hadoop projects. The study presented conclusions on the causes and impacts of data corruption and listed limitations in detection and handling of data corruption mechanisms. The effect of data corruption is not only limited to data integrity. It may lead up to service unavailability. As presented in [57] corrupted iCloud data was a cause for iOS home screen crash.

The seriousness of the issue reaffirms the need to have SLAs guaranteeing the security aspect for data stored in the cloud. Such SLAs, in addition to other properties, would be meaningless without a verification mechanism. For existing SLA metrics, monitoring is performed by tenants or third-party companies, and service providers should confirm the violation. In order to minimize the trust issue between service providers and tenants, we need an open (non-secretive) process to do verification and to store and share the result without any bias. In this chapter, we show a monitoring mechanism that can be used to check the correctness of data stored in the cloud without relying on the service provider.

As we have seen before, monitoring is a continuous verification process, and we provide a verification method for the integrity property of data hosted in a remote server without relying on the service provider. As in any security solution for clouds, our goal is to perform the verification with a limited performance impact, i.e., with minimum overhead in the production process. We start by describing the SLA life-cycle process for data in clouds and its difference with SLAs described in previous chapters.

6.2 SLA Life-cycle for Monitoring Data Integrity

The general SLA life-cycle process described in Section 2.5.3 also applies for SLAs guaranteeing data integrity, i.e., SLA life-cycle for data integrity also has three phases namely *SLA definition and negotiation, enforcement*, and *verification*.

The contents of each phase of the SLA life-cycle in this chapter is different from what was presented in the previous chapter for security monitoring SLAs as the targeted properties are not the same. In this chapter SLAs address security properties for data hosted in a cloud, specifically the integrity property. Other properties like confidentiality can be included in such type of SLAs. Section 6.8.3 describes how to incorporate the confidentiality property in the proposed method. In most existing cloud storage services (e.g., Amazon S3 [8] and DriveHQ [70]) availability is addressed in their SLA. Other properties are not addressed in SLAs. For example, Amazon S3 claims to have "extremely durable" storage with data stored redundantly across multiple devices and checks for corruption while data is at rest and in the network. However, Amazon does not guarantee these properties through an SLA.

The difficulties described in Chapter 3 related to quantitatively measuring security properties are still valid for the case of data integrity and confidentiality. It is difficult to measure integrity and confidentiality quantitatively. In our use case, tenants require checking the correctness of their data, i.e. either their data is corrupted or not. Hence, we assume the definition of SLA with an objective to keep the data uncorrupted as long as possible. Other properties like backup frequency and type of access control policies can be included in the SLA definition. Additional security properties like write-serializability (i.e. consistent among updates made by authorized users) and read freshness (read operation returns the latest update at any point) can also be integrated into SLAs.

SLA enforcement for data integrity can be achieved using different mechanisms. That is different techniques can be applied to keep the data uncorrupted, for example, replication and cryptographic methods. Data corruption can occur due to software bugs, design flaws, human errors, hardware failures, natural faults (bit flips), and malicious attacks. There exist different detection and correction mechanisms for various types of corruption sources.

Monitoring is used to check the satisfaction of SLA terms and detect any violation. Monitoring SLAs guaranteeing data integrity means verifying the correctness of data stored in the cloud. The verification can be performed either by a provider or tenant. In this chapter, we want to show a mechanism to perform verification without relying on the provider. We use a distributed system, blockchain, in which as long as the majority of the participants are cooperating to keep the value, stored data can not be changed.

Note that this chapter shows a verification process for data integrity, but the method could be easily adapted for other properties. The goal is to show the advantage of trusted components in user-centric security SLAs to reduce dependency on providers to verify SLOs. In Section 6.8.3 we discuss the possibilities in the direction of using other trusted components, like a secure hardware component (see Section 2.3.3) for SLAs presented in Chapters 3 to 5.

In this chapter, we first describe the problem that we address and give a background on the blockchain and types of blockchain. Then we describe the threat model assumed for the verification process. We describe the verification process and a prototype implementation. Finally, we present an experimental evaluation and discussion on the proposed method.

6.3 Problem Description

This section presents the problems addressed in this chapter. SLAs in this chapter describe the integrity property of data hosted in a cloud. We focus on the third phase of the SLA life-cycle: SLA monitoring. Hence, most of the general problems from Chapter 5 (i.e. problems that are not specific to NIDSs) are also challenges for data integrity SLA monitoring.

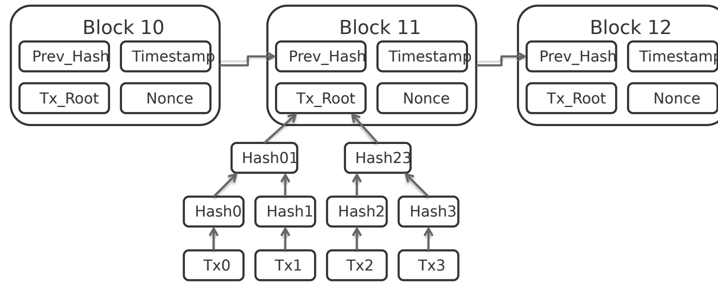


Figure 6.1: Simplified block chain representation

One of the biggest challenges in doing verification is having different owners in different tiers or layers of the cloud (as described in Section 2.1.3). Verification in such an environment means checking the status of a service from a tier which does not belong to the verifier. An SLA guaranteeing data integrity is not different, the provider infrastructure holds the data and a tenant wants to check the integrity of that data.

The verification mechanism presented in the previous chapter requires cooperation. In this chapter, we aim to reduce the need for cooperation between tenants and providers. The considered SLO is to keep data uncorrupted as long as possible. Our problem is similar to a well-known problem called remote data integrity checking, which enables a server to prove to an auditor the integrity of a stored file. However, in our case the verification is in two ways, i.e. tenants need to check data integrity in the cloud and providers need to check the correctness of SLO violation claim which can be submitted by tenants.

In summary, in this chapter, we address the problem of SLA verification for data integrity which allows both the tenant and provider to perform checks without relying (or with a minimum dependency) on the other party.

6.4 Background and Related Works

The verification mechanism proposed in this chapter is based on the blockchain technology. In this section, we introduce blockchain, its various types, and features. This section also presents related works on data integrity monitoring tools.

6.4.1 Blockchain

The blockchain is a distributed linked list data structure, list of connected blocks, where every block contains a cryptographic hash of its predecessor block, hence forming a chain (see Figure 6.1) [154]. A valid change of a block requires changing every block after that modified block until the end of a chain. Additionally, the chain is stored in a *distributed manner*, i.e. there is no central location keeping all the blocks. Every valid block in the list is available in every participant node. Adding a new block (i.e. extending the list) requires an agreement between the participants. Participants use a consensus algorithm to decide what to add next in the chain.

Blockchains are distributed, and they are relying on consensus algorithms both of which makes the system trustworthy without trusting any specific participant. It is important to note that blockchains are not 'bulletproof', as the technology is emerging and getting lots of attention, different studies are being conducted to study its property from different aspects. In general blocks in a blockchain contain at least two parts:

- *Link to a previous block*, the hash value of the previous block will be included in the current block, and it serves as a link from the previous to the current block. If any byte is changed in the previous block the link would be broken, i.e. the hash value will not match with a reference in the current block hence, it requires updating all blocks after the modified block, which creates a new line of the chain.

- *Data stored in the current block* is the information stored in the block. The data can be anything depending on the application using the blockchain. Most well known digital currency implementations store information like timestamps, transactions, and values used in the consensus process. Figure 6.1 shows a simplified snippet of Bitcoin [154], a well known digital currency blockchain.

The data section stores transactions and for an efficiency reason, they are stored in a *hash tree (Merkle tree)* data structure.

Following this, we describe types of blockchains, consensus algorithms used and existing implementations. Furthermore, we present in detail Hyperledger Fabric [18] which is the blockchain used in our work.

Types of Blockchain

Based on write permissions (permission to add blocks) blockchains are categorized into three categories [161]:

- *Public blockchain* is a type of blockchain where anyone from anywhere can join the network and participate in the consensus process to add blocks. Usually, participants have economic incentives to cooperate and when anyone wants to add a block, there is a simple verification process performed by other participants to check the correctness of the proposed block.
- *Private blockchain* is a type of blockchain where only one entity (e.g. the administrator) is allowed to add blocks. Some applications like database management may use such kind of data structure. Permission to read from the blockchain can be public, restricted or private as required.
- *Consortium blockchain* is a type of blockchain where only a pre-defined set of nodes are allowed to add blocks in the chain. It can be seen as a private blockchain having more than one authorized entity to add blocks. An example could be a group of financial institutions or a group of colleges and universities forming a consortium. As in the case of private blockchains, 'read' permissions can be set as required.

These categories could also be generalized into two groups, *permissioned and permissionless.* Consortium and Private blockchains are permissioned while Public blockchains are permissionless.

Consensus Algorithms

A blockchain is a distributed, peer-to-peer system and an agreement is needed to make any decision in the system. Consensus algorithms [185] refer to this process and in computer science, it is a well-known problem. A prominent class of such problem is the Byzantine Generals Problem [125]. It is a problem to reach an agreement in the network in the presence of potentially malicious participants. *Byzantine Fault Tolerance (BFT)* is the characteristic of an algorithm which defines a system that tolerates the class of failures that belong to the Byzantine Generals' Problem. Byzantine failures are considered the most general and most challenging class of failures to deal with. There are other types of failure models where non-malicious faults are tolerated, like crash fault tolerance.

A Byzantine failure implies no restrictions and makes no assumptions about the kind of behavior a node can have. If an algorithm can handle such a failure, it is assumed as robust. The goal of consensus algorithms in blockchains is to reach this level of fault tolerance. Moreover, in a blockchain, what is needed is not a Byzantine fault-tolerant SQL database. Instead, it is a Byzantine fault-tolerant distributed, append-only ledger, i.e. the duplicated data is not controlled by a single entity (multiple participants) and the data stored in the system is immutable.

In practice, there are different implementations of consensus algorithms. In the field of blockchains, we can generally classify these algorithms into two groups, namely *Byzantine-based* and *Proof-based* consensus algorithms.

- Byzantine-based consensus: is the traditional way of reaching an agreement between nodes in a network. The most well-known implementation in this category of algorithms is *Practical Byzantine Fault Tolerance (PBFT)* [41] which was published in 1999. After PBFT, several BFT algorithms were introduced to improve its robustness and performance. Few blockchain systems implement byzantine-based consensus. Hyperledger Fabric [18] is an example where PBFT is implemented, at least as a prototype, and used for consensus.
- Proof-based consensus: are generally called "proof of X" algorithms, where 'X' indicates evidence for having some required property. The node who poses this proof can be a leader and can propose a new block to be added. Then the block will be propagated and verified by other participants before adding it to the block list. This kind of algorithms can be seen as a probabilistic approach

for Byzantine-based consensus. Examples of proof-based consensus algorithms include Proof of Work (PoW) [154], Proof of Stake (PoS) [34], Proof of Elapsed Time (PoET) [45].

PoW is the first implementation of this category of algorithms, and it is used in major digital currency applications including Bitcoin [154] and Ethereum [33]. In the PoW consensus algorithm, to be a leader and propose a block, one must solve a cryptographic problem. The first one who gets the solution can propose a block, and other participants verify, add the block and start working on the next block. The problem to be solved is difficult, computationally intensive, and the difficulty level can be adjusted to control the rate of new block creation. Other "proof of X" algorithms also follow a similar logic but rather than having the computationally intensive problem they propose to use other parameters.

Blockchain Implementations

In early computer science studies the concept of state machine replication was used as a general method for implementing a fault-tolerant service by replicating servers and coordinating client interactions with replicas [198]. The Blockchain technology addresses similar problems and the first implementation, Bitcoin [154], showed the potential of its usage. After that, different varieties of implementations have been done. In general, there are three generations of blockchain implementations [18]:

- *First generation*: the early implementations of blockchains have few distinct properties. The ledger (blockchain) is used specifically for one application, mainly digital currency with a particular scripting language to perform operations. It uses a specific (fixed) consensus algorithm, operations are related with the currency used in the chain, and it is permissionless; hence anyone can join the network. An example of this first generation blockchain is Bitcoin [154].
- *Second generation*: this group of implementations improves upon the first generation. The second generation blockchains separate the logic and storage parts into different sections. The logic part, also known as a *smart contract*, is the only component allowed to do operations (read and write) in the storage component. The storage part, as the name indicates, is responsible for storing the actual data. This separation allowed using blockchains for applications other than digital currencies e.g. for identity management or certificate authority. As a result of this separation, the logic component can be programmed to perform *deterministic* operations. Domain-specific languages are used to enforce the deterministic property in the scripting language. This generation implementations uses a fixed consensus algorithm which forces taking the same threat model for all applications. Ethereum is a prominent example of this generation.
- *Third generation*: this group of implementations follows a different kind of architecture to remove the constraint of using deterministic languages to write smart contracts while sacrificing on being permissionless. It allows writing smart contracts using general purpose languages and consensus in the blockchain is modular, i.e. an application can plug and use any type of consensus algorithm depending on its need. Being permissioned means that there is a requirement to get permission from some entity which, some argue, makes a blockchain centralized.

It is important to note that this classification is not universally accepted; it is possible to find other classifications, like [225].

Hyperledger Fabric (HLF)

Hyperledger Fabric or *Fabric* [18] for short, is an implementation of permissioned blockchains. It was initially developed by IBM and it is now hosted by Linux Foundation project called Hyperledger, hence the name *Hyperledger Fabric (HLF)*. HLF is permissioned by design, and it is developed focusing on business use cases where participants are expected to be known in the system. HLF introduces a different architecture than the first and second generations of blockchain implementations.

The data stored in blocks are in the form of transactions, which are executed on the current state of the system and create the next state. Before HLF, other implementations follow an *order-execute* architecture, i.e the first transactions are ordered, and a consensus is reached on that order. Then every participant node executes transactions in that order and updates the ledger accordingly. However, this architecture has drawbacks including:

- *Transactions must be written in deterministic code*: Since consensus is reached before execution, a non-deterministic code may create different results on different nodes, hence breaking the consensus and creating more than one state (*fork*) for the blockchain. In previous generations of smart contracts, domain-specific languages (e.g. Solidity for Ethereum) are used to achieve a deterministic behavior.

- *Limited to sequential execution*: Since a consensus is reached before executing the transactions, every node executes transactions in a block sequentially. Sequential execution affects the throughput in the system limiting any parallelization.

HLF introduces a new type of architecture called *execute-order-validate. Fabric* tries to overcome previously described and other drawbacks from prior permissionless blockchains (e.g. hardcoded consensus and confidentiality) by introducing the highly modular *execute-order-validate* architecture.

In this model transactions are first executed (to be specific, transaction execution is simulated), all nodes are not required to do this step, and this will not affect the state of the ledger because it is a simulation on a local copy of the ledger. This feature gives the opportunity for confidentiality and parallel execution. Then transactions are ordered using a consensus algorithm, and any algorithm can be used for this process. This feature gives the opportunity for a modular blockchain design which can use different consensus algorithms depending on the application threat model. Finally, a predefined policy is used to validate transactions. Such a policy is defined per application level which allows having a flexible trust model. Furthermore, since ordering is performed after execution, the executions are not expected to be deterministic, the validation phase guarantees freshness i.e conflicting transactions will be dropped.

Fabric provides the confidentiality feature by dividing the network into different *channels*. An organization will get permission to perform some action (according to its role) only if it is a member of that channel. This is an essential feature in Hyperledger Fabric, and it is useful in cases where participants might be competitors or just when a subset of participants are not part of a project. The organization deploying an application in a channel can decide who can participate in the process of that application.

As *HLF* is a permissioned blockchain, there is *Membership Service Provider (MSP)*, which is responsible for maintaining thidentities of participants in the network. MSP is an abstraction where the back-end could be different membership standards and architectures. The interaction between entities in the network is cryptographically signed with identities provided by MSP.

As a design principle, HLF follows a highly modular design pattern. It provides different components of the blockchain network as Linux containers (containerized) service. Each organization contains different services including peers, orderer, MSP and so on.

In the next section, we present related works focusing on data integrity checking tools and practices.

6.4.2 Related Works

This section presents the related works in two parts, first related works on remote data integrity checking and second, the usage of blockchains for data integrity.

Remote data integrity checking

The concept of remote data integrity checking has been around in relation to distributed systems [167]. While the adoption of cloud increases, remote integrity checking becomes more critical because more and more sensitive data is being stored in the cloud. There have been different works on checking properties of remotely stored data, including integrity. One way to check data integrity is by completely downloading the data and compare it with the original version but this method contradicts the basic idea of the cloud, and it is impractical. Usually, the size of data stored in the cloud is huge, and users may not keep a copy of the entire data. As a result, almost all remote integrity checking works rely on cryptography methods without the need to completely download the data.

Due to its necessity, remote data integrity checking has attracted extensive research interest [101, 102, 134, 228]. Generally, we can categorize these works into two main groups, integrity checking *with and without a third party auditor (TPA)*. TPA is an entity who has expertise and capabilities for performing an integrity check and convincing both the client and the provider. It is considered to be a trusted party. TPA performs the primary role of data integrity check; it performs activities like generating a hash value for blocks received from the cloud server and compares signatures to verify whether the data stored in the cloud has been tampered with or not.

For example, TPA is used in [134]. The paper presents a remote data possession checking protocol with the support of public verifiability (i.e. anyone can perform the verification). It defines four functions namely *KeyGen, SigGen, GenProof, VerifyProof*: *KeyGen* run by users to generate a key, *SigGen* used by users to generate verification metadata, *GenProof* run by service providers to generate a proof of data storage correctness and *VerifyProof* used by TPA to verify the proof from service providers. The protocol executes in two phases, namely the *setup* and *audit* phases. In the setup phase, users generate a key using the *KeyGen* function and generate metadata for data files to be sent to the cloud. Then the user sends the data to the cloud and publishes the metadata to TPA. After that, a user can delete the local copies of the uploaded data.

The second phase is an audit phase. Upon request by a user, TPA starts the verification process. TPA formulates and sends a challenge to the provider and waits for a response. The service provider upon receiving the challenge, runs *GenProof* to generate a proof showing the data is correctly stored and sends back the proof to TPA. The TPA runs *VerifyProof* using the returned value and checks the result with the original metadata. This way TPA is used to check data integrity in the cloud.

Other works like [228, 101] follow a similar pattern, but they use different cryptographic methods to achieve properties other than integrity, e.g data dynamics, privacy against verifiers, and proof on multiple providers. The protocol presented in [228] supports data dynamics, and the work presented in [101] supports privacy against third-party verifiers. In some works the TPA is optional and the user can also be an auditor. However, in other works, the TPA is required and acts as a trusted third party (TTP), It is used to resolve disputes between the provider and tenant.

There are also some works which do not use third parties in the process. The protocol presented in [102] does not rely on a TPA. It follows a similar procedure as described above using six functions to operate, except the cryptographic procedures are different. Such protocols cannot be used for SLA verification because the process is only one way, i.e. only one party is performing the integrity check. For our use case both parties need to perform the verification. CloudProof [177] presents a storage system which provides proofs of a violation; hence neither providers nor tenants can bring a false claim of violation. Our work can be used to extend such a system in order to exploit the distributed nature of blockchains and become even more independent from other entities, like certificate authorities.

In our work, we want to reduce dependency between participants and possibly remove TTPs as it requires as much trust as providers. We thus propose to rely on a secure and distributed ledger, blockchain.

Blockchains on data integrity

About the usage of blockchains, there are different applications for data integrity services in various scenarios including IoT, database systems, data provenance tools and so on. In addition, new distributed storage systems based on blockchains (e.g Storj [66]), which include the data integrity feature by design are being developed. In [131] the authors proposed a blockchain-based data integrity framework for IoT data stored in the semi-trusted cloud. The framework incorporates data generators and data consumers and enables consumers to perform integrity verification. The framework follows a typical blockchain applications strategy, i.e it does not store all the data in the blockchain, it stores only the hash of the data and the actual data is stored in a cloud storage service.

The authors in [85] presented a blockchain-based database with strong integrity guarantees. The system uses two layers of blockchains, the first layer with a lightweight distributed consensus protocol that assures low latency and high throughput. The second layer is designed with a strong consensus to guarantee better integrity by using PoW-based algorithm. There are new upcoming companies (e.g Chainpoint [14]) offering to anchor users data to existing blockchains, which helps to verify the integrity and existence of data without relying on a trusted third party.

To use the presented studies in our work, as they are proposed they do not describe the ability to verify integrity by other parties (other than the data owner). This feature is mainly a result of the considered threat model. In other words, using these works for our use case would allow only one party to perform the verification. Having a verification method that can be used by both parties is a requirement for our verification process. In that sense, we are addressing a different kind of problem than most blockchain-based applications.

In our work, we want to show the use of a trusted component, like blockchain, in the SLA verification process in order to reduce the dependency between tenants and providers and enable independent verification for both parties. In the next section we describe the assumed threat model.

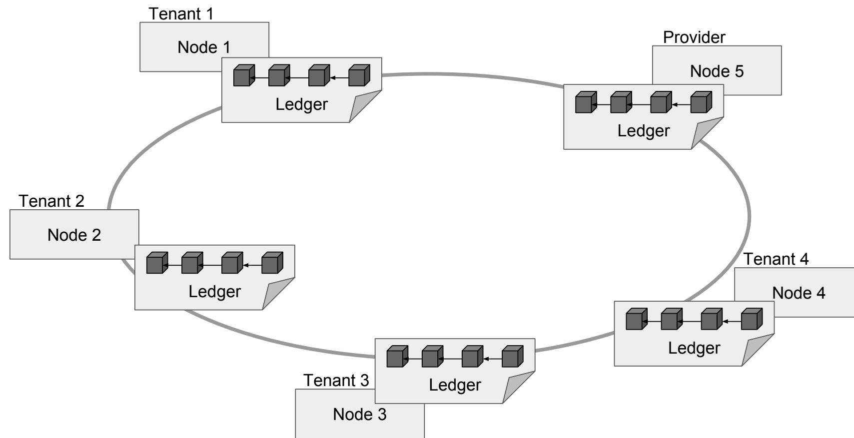


Figure 6.2: Consortium blockchain network formed by tenants and a provider

6.5 Threat Model

In this chapter, we assume that cloud service providers are semi-trusted. In a sense, a provider may return wrong data or wrong result of computation for a request from a tenant because an attacker altered the data or of some other error. However, providers are not actively trying to alter tenants' data. In addition, providers never lie when claiming that they have not accepted to store some data. They are working towards maximizing profit and errors on the stored data are unintentional. Providers also have economic incentive related to an SLA. Hence, they may lie in order not to violate an SLO. Tenants store their data in the cloud and do not keep a local copy of the whole data. Tenants may have some portion of the data and store hashes of the complete data on the ledger (blockchain) to be used later for verification.

Tenants may falsely claim a reward for a data integrity breach. Hence, providers need to do verification by themselves. However, tenants never lie when claiming that providers accepted to store some data. A tenant is interacting with a single provider, and the provider replies to requests about the data. Providers may store the data in one location or divide it into different chunks for security or any other reasons. We assume the blockchain network is not compromised; at least the minimum number of required participants are honest and are not controlled by the provider. In addition, we assume there is a secure communication network between the three entities, tenant, provider, and the ledger. Notably, the integrity of a message is respected, i.e. there is no man-in-the-middle which is actively altering the communication between the three entities.

6.6 Data Integrity Checking Process

In this section, we describe our proposed method to check integrity using a trusted, secure ledger and without relying on a third party. This method is used to perform SLA verification. Using the proposed method, a tenant can check the integrity of the data stored in the cloud, and a provider can check the correctness of SLO violation claims without relying on the other party. Our method guarantees to link the provider and the tenant with the same piece of evidence for data integrity.

As described in the previous sections, we use a blockchain to remove the trusted third party. This secure ledger is used to store a piece of evidence for attesting the correctness of outsourced data. Providers and tenants form the blockchain network. In a user-centric security monitoring context, using such a technique gives two main advantages. First, it adds trust, transparency, and security to existing integrity monitoring techniques without other parties and second users are directly participating in the process of securing their outsourced information system. Allowing users to participate in the process is the core of user-centric services and systems.

In general, given an SLA guaranteeing the integrity of users data, the verification process follows two main procedures, namely the *setup* and *verification* phases.

- *Setup phase* is where a tenant prepares *tags* for the data to be used later in the verification phase

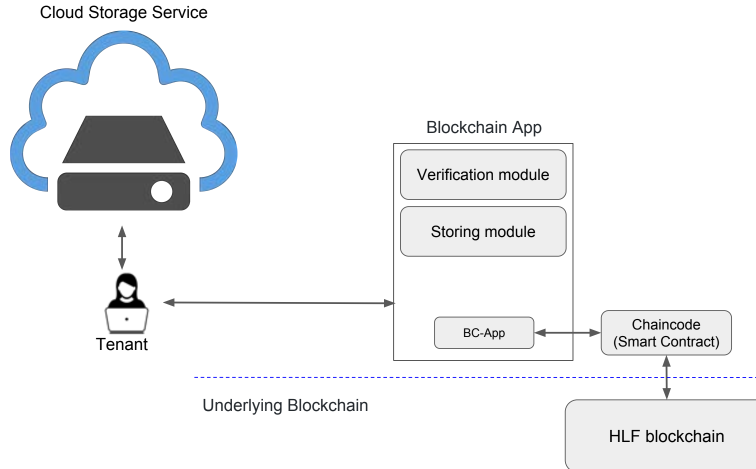


Figure 6.3: Data integrity verification architecture

and it is performed before uploading a file to the cloud. *Tags* are proofs containing different components including the hash of the data. The tags are used by a tenant to perform verification without the need to trust either the provider or any third party entity. Section 6.6.2 describes *tags* in more detail. After this process, a tenant sends the data plus hash of the data to a provider. The hash value is used by service providers to check the correctness of the initial data upload. After successfully uploading the data, both the tenant and the provider publish the hash value of the data to the ledger.

- *Verification phase* is the procedure after the data is uploaded correctly to the cloud and the hash value published in the ledger. A tenant sends a request for providers to perform integrity checking and receives the hash value performed over the current state of stored data. Using the tags generated in the previous phase a tenant can confirm the correctness of the stored data. If the data is not correct, a tenant can claim an SLO violation using the proof stored in the ledger and proceed with the next procedure as stated in the SLA.

Section 6.6.2 presents a more detailed description of each phase. While the blockchain is serving as secure storage for hash values that are used to prove integrity, the providers' infrastructure stores the actual data. In the next subsections first, we present the architecture of the integrity checking platform, and second, we describe in detail the verification process.

6.6.1 Architecture

In this section, we describe the architecture and components of the proposed integrity checking platform. Our data integrity verification platform contains five main components (see Figure 6.3). These are the *underlying blockchain, smart contract (chaincode), blockchain app, a service provider (cloud) and a user (tenant).*

- *Underlying blockchain* is the blockchain network which stores pieces of evidence (hash values) of data that are used as an anchor for integrity checking. This component is assumed to be a trusted and secure storage system. The blockchain network is formed by the tenants and providers. It is a consortium blockchain, i.e. a private blockchain having more than one authorized entity to add blocks (see Figure 6.2). As described above every participant in a blockchain network holds all valid blocks and updates the list according to the consensus algorithm used. Our SLA verification method is not dependent on any specific blockchain implementation, i.e. it can be used with any type of consortium blockchain implementation. The number of required participants in the network depends on the type of blockchain implementation used and the resiliency model requirement (e.g. using HLF with a crash fault tolerant consensus algorithm, to tolerate n number of crashes it requires $2n+1$ nodes). Section 6.7 presents the details of our implementation.
- *Smart contract (chaincode)* as described in Section 6.4.1, is the logical component of blockchains and can be programmed to perform different tasks. It is the only component which directly interacts

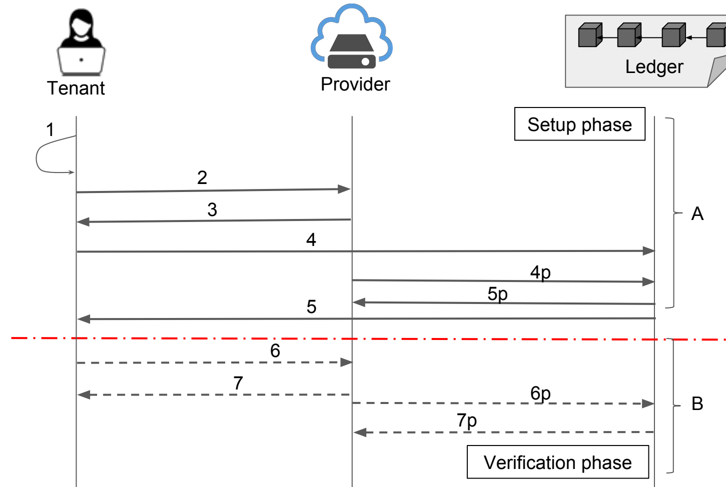


Figure 6.4: Data integrity verification process, (A) setup phase and (B) verification phase

(perform read and write operations) with the underlying blockchain. Our smart contract consists of the following functions: *initLedger()* called only once to initialize the blockchain. *addData(data)* used to add 'data' to the ledger, *queryData(data)* to check if 'data' is in the ledger and *Invoke(f, param)* used by external applications to call a function from the chaincode.

The smart contract, which performs operations on the blockchain, avoids any duplicate entry in the blockchain, i.e when receiving a write operation from a tenant or provider, the smart contract first checks if the same data exists in the blockchain. If it found the same data it returns the ID of the existing block. Otherwise it will add the requested data. As a result, even if both tenants and providers make a separate request using the same data, the data will be added only once, and both the tenant and provider will hold the same block ID.

- *Blockchain application* or client application is a module which acts on behalf of a user, i.e. the entity who wants to call functions from the smart contract. The caller can be either tenants or the provider to store or retrieve a piece of evidence for a data block. Our application contains the *storage module* used to store evidence in the ledger and the *verification module* to retrieve evidence. These modules call the *addData()* and *queryData()* functions respectively from the smart contract.
- *Service provider (cloud)* is an entity providing the storage service. The provider offer SLAs to guarantee data integrity. As described in Section 6.5 providers are not malicious, and they can respond to requests (challenges) from a tenant or client application. They can also perform verification in order to check the correctness of an SLO breach claim from tenants.
- *Users (tenants)* are owners of the data stored in the cloud and they sign an agreement with the cloud provider. Users add evidence of data integrity into the ledger and they perform verification of data integrity to check if the SLO is still valid.

6.6.2 Integrity Checking Process

The process of checking integrity contains two phases, the *setup* and *verification* phases. In the setup phase, the tenant generates and stores required information for later verification. In the verification phase, the actual checking is performed by using evidences generated in the previous phase. Figure 6.4 shows the two phases of the integrity checking process. We also provide algorithms (Algorithm 4 and 5) describing both phases. The numbers in the figure correspond to the line numbers in the algorithms. The numbers with the suffix 'p' in the figure represent the tasks performed by the provider.

Setup phase

Figure 6.4 (A) shows the setup phase and Algorithm 4 shows the procedure in the setup phase (the line numbers in the algorithm represent the arrow numbers in the figure). In this section we present a step

by step explanation of the process:

- The tenant (owner of the data) performs an operation on the data to produce evidence (tags) that will be used later for verification. In practice, this procedure is hashing the data using a secure hash algorithm e.g. SHA-1, SHA-2, xxHash... A hash function maps input data of arbitrary size to a byte string of a fixed size. Depending on the security guarantees, there are two main categories of hash algorithms, namely *cryptographic* and *non-cryptographic* hash algorithms. The cryptographic hash functions provide strong security guarantees like collision resistant, one-way function, pre-image attack resistant etc. On the other hand, non-cryptographic hash functions provide weaker guarantees in exchange for performance improvements. They just try to avoid collisions for non malicious input. Since we consider non-malicious inputs, a non-cryptographic hash algorithm is enough for our use case. Using such a function the tenant generates the required tag values.

  The generated tags include three parts, first hashing the data, *H(D)*, second generating n random strings called *nonce*, $(R_1, R_2...R_n)$, and third hashing the data concatenated with every random string, $(H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n))$. The number of random strings are determined by the length of the SLA validity period and the frequency of verification. For example, if an SLA is valid for five years and verification is performed once per day, the tenant will generate 1825 (5 * 365) different strings and performs the hash of data plus a random string, for every value. The hashes and generated random values, i.e $(H(D+R_1),\ H(D+R_2)\ ...\ H(D+R_n))$ and $(R_1, R_2...R_n)$, should not be shared with other parties, specifically with the provider. These nonce values are used to force providers into performing a fresh hash computation. Even if the method requires $(n + 1)$ hash operations, it is practical in the SLA verification context. It is feasible because of two major reasons (i) the SLA has a validity period and (ii) data integrity is not a property which is checked very often like availability. Thus, the variable 'n' is bounded.

- Tenant uploads the data with the hash value, i.e. (D, H(D)) to the cloud storage. Upon receiving the data, the service provider runs the same hash function over the data and compares the result with the provided hash value. If the value matches the provider will send a confirmation; otherwise, the provider assumes there is an error in the data transfer and notifies the tenant. In the case of such an error message, the tenant should repeat the process.

  In practice this process is not novel, Amazon S3 [8] command line tool offers a similar option. The command *'s3api put-object'* takes *'--content-md5 and --metadata'* arguments and Amazon uses this information to perform integrity checking. Amazon confirms the correctness by returning an *'Etag'* and stores the hash value with the data. The hash value can be retrieved at anytime to check the integrity of stored data. This step is done only once unless the data is changed or updated. In that case, the hash of the new data should be computed.

- Once the upload is confirmed the tenant publishes the hash value, i.e $H(D)$ in the ledger. Publishing a hash value can be achieved by using the storage module from the client application. Providers also publish $H(D)$ to the ledger. However, as described in the previous section, the smart contract prevents duplicate entries in the ledger; thus the second *addData()* operation returns the block ID of the previously added data. This way both the provider and tenant have the same block ID.

Verification phase

At this stage, the hash value of the data is published, and both the tenant and provider have a block ID referring to an address where the hash value is stored. It should be noted that random nonce values, $(R_1, R_2...R_n)$, and hash of the data with these values, $(H(D + R_1),\ H(D + R_2)\ ...\ H(D + R_n))$, are stored privately by tenants.

To perform verification, a tenant can challenge a provider to compute a hash value over the current state of the stored data. Figure 6.4 (B) shows the verification phase and Algorithm 5 shows the verification process. The steps to verify the integrity of the data are:

- A tenant selects one R_i and sends it to the provider with the file name to be checked. This value of R_i is then removed from the set of random numbers, i.e. it will be used only once.

- The provider computes the hash of the data concatenated with the nonce value (R_i) and returns the result to the tenant. The tenant compares the return value with the locally stored value and concludes about the integrity of the data stored in the cloud.

Algorithm 4: Setup phase

```
Input: Data 'D', Hash function 'H()'
Result: n Random strings (R_1, R_2...R_n), Hash of 'D' H(D) and Hash of 'D' with random strings
        (H(D + R_1), H(D + R_2) ... H(D + R_n))
/* these random strings are used later for verification. Both the random strings
   and (H(D + R_1) ...) should be kept private with tenants                      */
1 Generate n random strings, compute H(D) and (H(D + R_1), H(D + R_2) ... H(D + R_n)) ;
2 Upload the data, i.e send (D, H(D)) to the service provider ;
3 The provider computes a fresh hash of 'D', H(D)' and compares it with the received one;
  if H(D)' == H(D) then
  |  return 'success' ;
  else
  |  return 'error' ;
  end
4 if the previous step is successful then
  |  /* Both tenants and service providers execute the following function        */
  |  addData(H(D));
  |  /* the smart contract adds H(D) only once and subsequent addData(H(D))
  |     requests return the blockID of existing H(D)                              */
  else
  |  return to step two and re-upload the data;
  end
5 Get blockID, the blockID indicates where H(D) is stored in the ledger
```

In the event of a discrepancy between these two values, a tenant can claim for an SLO violation and use values from the ledger as evidence. Our integrity verification process guarantees that the tenant and the service provider will hold the same block ID value, i.e. they both refer to the same value in the ledger. It is important to note that holding the same pointer to a secured data storage location does not automatically resolve a conflict between tenants and providers. However, it can help in the process to resolve a disagreement between the two parties. One way of such usage can be in the process of legal action.

Values written in the ledger are immutable, i.e. it is secured by duplication and the consensus algorithm; hence, the ledger is serving as a secure and trusted anchor for both tenants and providers. Using a different nonce value for every verification, the tenant forces the provider to compute fresh hash values.

Service providers can check the integrity of stored data, by hashing the data over its current state and comparing the result with the one stored in the ledger. This checking process is especially helpful when there is a complaint from tenants, and a provider wants to check the validity of such claims.

Algorithm 5: Verification phase

```
Input: n Random strings (R_1, R_2...R_n) and Hash of 'D' with random strings
       (H(D + R_1), H(D + R_2) ... H(D + R_n))
Result: true (no data corruption) or false (there is data corruption)
6 Select one R_i from the set of strings and send it to the provider. Remove R_i from the set of
  strings in order not to use it again ;
  // note that the provider cannot cheat because R_i is different on every request
7 Compute H(D + R_i) and return the result ;
  if result == Previously computed H(D + R_i) then
  |  // no data corruption
  |  return 'true' ;
  else
  |  // there is data corruption
  |  return 'false';
  end
```

6.7 Implementation

We have implemented a prototype of the proposed data integrity SLA verification tool. In this section, we present the implementation details by describing each component listed in Section 6.6.1.

We used *Hyperledger Fabric* (HLF) [18] as a back-end blockchain. For our use case, the unique features of HLF provide an advantage over other implementations. We describe three advantages over other well-known implementations. HLF is permissioned, there is no native digital currency, and it allows writing a smart contract in any programming language.

As described in Section 6.4.1 HLF is one of the major implementations of permissioned blockchains. HLF allows us to create a consortium blockchain network with the tenants and provider as the participants. The fact that there is no native digital currency means that participants can perform operations on the ledger without paying additional payments for each operation. HLF allows using general purpose programming languages to write smart contracts, which helps to develop chaincode (smart contracts) easily and rapidly. Additional benefits of HLF include its modular consensus, and its active community both from academia and industry. It should be noted that other types of blockchain implementations that can be used to build a consortium blockchain network can be easily adapted for our use case.

All the other modules are implemented using python except the smart contract, which is written using the *Go* programing language. We used the *xxHash* algorithm to perform hash operations. *xxHash* is an alternative to the SHA hash algorithm families. It is a non-cryptographic hash algorithm with better speed than SHA families. The main criterion for the hash algorithm is to avoid collisions and since we consider non-malicious inputs a non-cryptographic hash algorithm is enough for our use case. Specifically, we use the xxhash.xxh64() method in our implementation. We perform incremental hashing based on a fixed block size rather than hashing the whole data at once; this helps to decrease the time needed to hash a given file.

The next section presents the evaluation performed to measure the performance of the proposed method.

6.8 Evaluation

In this section, we start by describing the setup used to perform experiments. We present the performance evaluation of the proposed method, specifically the time overhead as a result of the verification process and resources required to run the proposed verification process. We present an analysis and discussion on the performance and security of the proposed method.

6.8.1 Experimental Setup

To measure the performance of the proposed data integrity checking platform, we built a setup on the Grid5000 [22] testbed infrastructure. Three physical nodes are used to represent a user, a provider and a consortium blockchain built using Hyperledger Fabric (HLF). Each physical node contains two Intel Xeon E5-2630 v3 CPU, eight cores per CPU and 128 GB memory, all running Ubuntu version 14.04. We experiment with one tenant and one provider. However, the blockchain network contains ten participants. Our tenant and provider control one node each.

The entire blockchain network runs on a single physical node with containerized services, i.e. participants and related components are instantiated using Docker containers [68], and they communicate through virtual networks. In a real production environment participants reside in different physical nodes. The blockchain network contains ten participants, belonging to five different organizations (i.e. each organization has two participants in the network). As described in Section 6.4.1, HLF uses a structure of organizations and peers. For our experiment, one node in the network is owned and managed by a tenant and one node by a provider. The remaining eight nodes can be seen as other tenants and providers participating in the network.

In our experiment, HLF uses a Kafka-based ordering service, i.e. a consensus algorithm which is based on Kafka cluster and ZooKeeper ensemble [18] implementing a crash fault tolerant consensus algorithm. There are four nodes in the Kafka cluster and three nodes forming the ZooKeeper ensemble. In practice, since HLF is modular and our verification method is not dependent on the consensus process, any algorithm can be plugged and used, including Byzantine fault-tolerant algorithms. The network represents providers and tenants who agreed on SLA terms on guaranteeing the integrity of data outsourced to the cloud infrastructure.

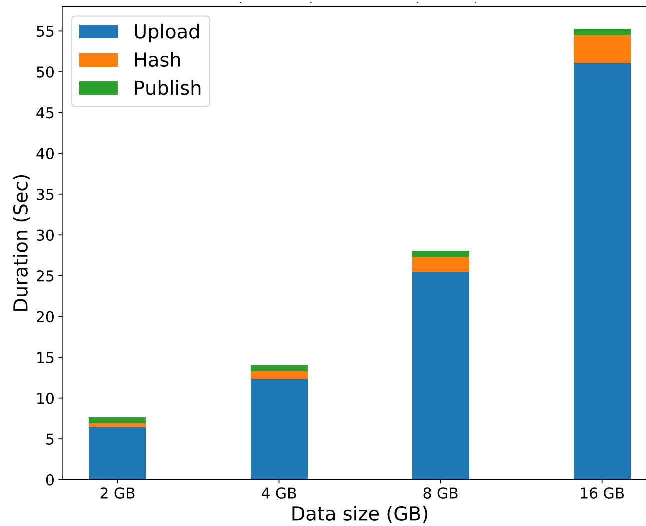


Figure 6.5: Time required for setup phase operations (upload, hash and publish)

HLF provides different components of the network as containerized services. Hence, the blockchain network is running on top of a Docker network, and each organization runs multiple Docker containers representing different components like peers, orderer and so on. A second separate physical node represents a cloud provider, specifically a storage as a service cloud provider. Users sign an SLA with the provider and submit their data after performing the setup process described in Section 6.6.2. In our experiment python Simple HTTP server, which implements the required functionalities like accepting tenants data upload requests, computing hash of submitted data and publishing hash value to the ledger is running on the provider node. A third separate physical node is used to represent a user. Users perform the setup and verification process on this node.

The data size is a significant factor in our verification process because the time needed for operations like hashing is directly related to the data size. For our experiment, we used different data sizes ranging from 2GB to 16GB. This range of data sizes is enough to show our experiment goals, but in practice, cloud users can upload tens or hundreds of gigabytes of data to the cloud. Initially, all the files are not cached i.e the cache is cold.

6.8.2 Performance Evaluation

The SLA verification method presented in this chapter requires two additional resources. First, performing the steps presented in Section 6.6.2 takes additional time and second, to participate in the blockchain network, it requires building at least one node in the HLF blockchain network. The results of the performance evaluation are structured in three parts: the time required for operations in the setup phase, the overhead of the verification phase, and additional resources required to participate in the process.

Using the setup described above, we measured the time overhead caused by the verification process. In an environment where there is no integrity checking the only task is to upload the data to the cloud provider. Adding integrity checks requires two more tasks, namely hashing the data and publishing the hash value in the ledger. There are two kinds of hash operations performed by tenants (i) hashing the data alone to be sent for the provider and be published in the ledger (ii) hashing the data with random nonce values. Figure 6.5 shows the time required for the first kind of hashing (i.e. hashing the data alone), publishing the hash value, and uploading data. Note that the time to publish a hash of the data is constant, because the output from a given hash function is always of the same size, regardless of the input size. Time to publish means the time required to write a hash value into the ledger. As shown in the figure it is relatively small (10% and 1% for 2 and 16 GB respectively) compared to the total time.

But it should be noted that our blockchain network is being simulated on a single physical machine. In a real production setup, participants in the network are physically distributed and the time to perform write operations could be higher. Section 6.8.3 describes the performance of HLF on geographically distributed participants.

We performed the experiment for ten rounds and results are reported over an average of ten rounds. Writing the output of the hash function to the ledger takes on average 0.725 seconds, almost twice

the time required for reading from the ledger. It is because writing to the ledger also performs a read operation to avoid multiple entries of the same data in the ledger.

As presented in Section 6.6.2, the setup phase requires selecting n random string values and performing the hash of the data with each of those values (second kind of hash). For example, an SLA with a validity period of five years and a frequency of one verification per day, the tenant selects 1825 random values and hashes the data with each value. In comparison, our method performs a smaller number of hash operations than other cryptographic solutions presented in Section 6.4.2. However, it takes more computation time than other solutions. It is because our method performs a hash of the whole data with each random strings while other solutions compute a hash of a few blocks out of the whole data. Section 6.8.3 presents a possible optimization technique using parallelism. Moreover, in terms of resources our method requires more resources to participate in the blockchain network.

Without considering the second type of hash, Figure 6.5 shows that the time required for uploading data is dominant over other tasks. This task is not avoidable even without performing an integrity check. For the second type of hash, if a single hash operation takes t seconds, the second type of hash operation takes $(n * t)$ seconds using a single process, where n is the number of random strings. For example, to hash a 2 GB and 16 GB files take 0.4564 and 3.4266 seconds respectively. However, this task is highly parallelizable and the time could be optimized to $\frac{(n*t)}{p}$, where p is the number of processes used for the hash operations and it is bounded by the number of CPU cores available on the tenant's machine.

If we assume a verification phase, it consists of asking providers to perform a hash of the data with one random string. Hence, this hashing task is the only additional time compared to the baseline i.e. without doing integrity verification. Figure 6.6 shows a comparison between the baseline (without integrity check) and checking integrity performed using the setup described above. The baseline operation (the solid blue line in Figure 6.6) measures only the time needed to download the given data while the integrity check (the dotted orange line in Figure 6.6) measures time to download the data plus the time for integrity checking operations as described in Section 5. Doing integrity checks introduces an addition of around of 6% of total time. The additional time is a result of hashing, and it is directly related to the data size which is also related to the baseline (upload or download) time. In the next section we present optimization measures for hashing large size data.

The other resources needed to perform the proposed SLA verification is a node to participate in the HLF blockchain network. A tenant or a provider can join the HLF network as an organization with minimum requirements for operation. These include a node which participates in the ordering service (i.e. consensus process), a node acting as a peer to maintain the state and store a copy of the ledger, and a client which acts on behalf of the tenant and submit transactions. These requirements can be achieved using light containers. HLF provides a set of containers for different services, and they can be hosted in a machine having as low as 2GB memory. Regarding disk requirements to store the blockchain data, every entry in the ledger is a key-value pair, and this should be easily manageable by using regular personal computer storage devices.

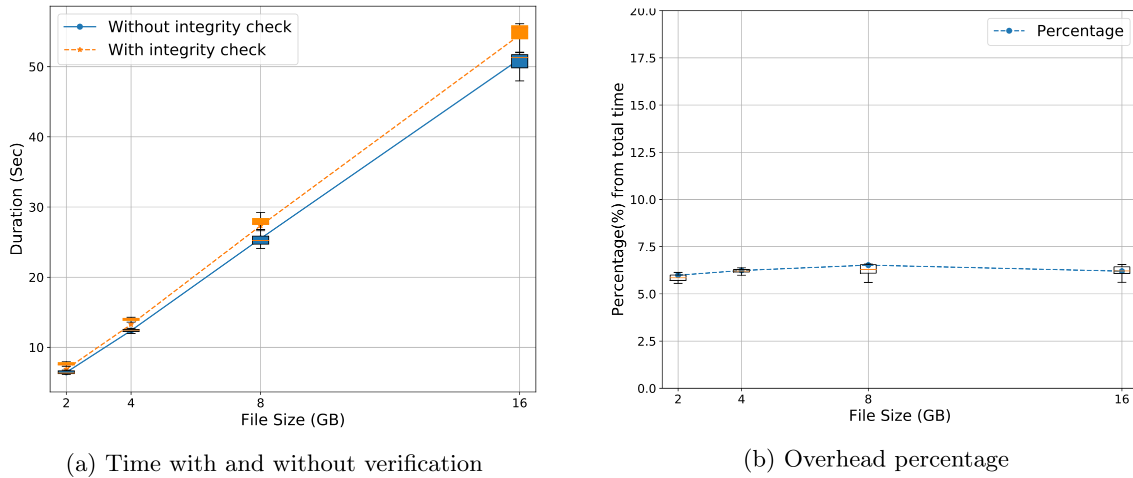
In the next section, we present a discussion and an analysis of the proposed SLA verification mechanism. We discuss advantages, drawbacks and future optimizations that can be done to have better performance.

6.8.3 Analysis and Discussion

The method proposed in this chapter addresses the issue of openness (non-secretive) in the SLA verification process. The method relies on a distributed ledger which runs a consensus algorithm to keep an untampered state of the stored data. The verification process integrates security and trust by design. Tenants and providers can perform verification independently, and no one is dependent on another single entity. However other entities are required to keep the blockchain network running.

We used HLF for our prototype implementation, and we deployed HLF with the default configuration without doing any optimization. If there is a need for high throughput, e.g. a large number of clients with frequent uploads, HLF can be optimized to handle more than 3500 transactions per second as presented in [18]. Moreover, in our experiment, the blockchain network is entirely running on a single physical node using Docker containers and virtual networks. In practice, participants in a blockchain network are geographically distributed. Such distribution introduces additional latency in the process, and our method should be further evaluated on this regard.

In our verification process, the given data is hashed with n random strings, i.e the process executes n hash operations. This verification process takes a longer time when compared to other integrity checking



(a) Time with and without verification

(b) Overhead percentage

Figure 6.6: Time overhead as a result of integrity verification

protocols. The hashing process can be optimized by using parallel processes. Since the tasks of hashing the data with n different random strings are independent of one another.

One of the assumptions in our threat model is a secure communication system between the cloud provider, tenant, and ledger. In the absence of such secure communication system, a man-in-the-middle attack can affect our verification process and create a conflict between tenants and providers. For example, a malicious attacker can alter the data sent from the provider to the ledger. Such an attack can cause a conflict when a tenant claims an SLO violation.

The provider neither lies nor guesses the result in advance when asked to compute the hash of data with a given random value because the tenant uses a different nonce value for each verification. Service providers can also check the validity of SLO violation claims by computing the hash of the data over its current state and comparing the result with the one stored in the ledger.

We assumed that a tenant would keep some part of the generated proof private. If a tenant loses these values, it is impossible to proceed with the verification tasks. Hence, the tenant may require a highly available, secure and private data storage mechanism.

On top of the proposed method, additional features can be added to the process. For instance, encryption can be used to add confidentiality. Blockchains can be further elevated to automate different tasks in the SLA life-cycle including payments for service and automatic compensation for SLA violation.

In this chapter, we showed the advantage of having secure elements in the SLA verification process for data integrity. The secure element used in our case is a distributed ledger (blockchain). This secure component is highly programmable to perform different tasks; as a result, any remote data integrity checking protocol can be implemented following our method.

In addition, the same logic can be applied to perform verification of SLAs described in the previous chapters. Secure components are designed for attesting and reducing the number of trusted components in different layers of an information system. For example, having a secure hardware component in the IaaS cloud could help to attest the outputs of security monitoring devices to protect them from intentional or unintentional changes by the provider. Of course, the implementation requires a more detailed study.

6.9 Summary

In this chapter, we started by discussing the problem of trust between providers and tenants while performing SLA verification. The previous chapters recommended cooperation between tenants and providers in order to achieve the desired goal. Different incentives were presented for both parties to cooperate in the verification process. However, in practice, it is difficult to have verified cooperation. This chapter addresses such an issue by introducing a third component, a secure trusted and distributed ledger (blockchain) in the process.

The method allows users to directly participate in the process of SLA verification and in keeping a proof used to support an SLA violation claim. As a user-centric service, our goal is to make users part of the SLA life-cycle process and distributed ledgers formed by tenants and providers help to achieve the desired user-centric design goal. In this chapter we assumed an SLA guaranteeing data integrity in the

cloud. We have presented the seriousness of keeping data integrity in many applications of IT systems.

We briefly discussed related works and presented an introduction to blockchains. We described types and implementations of the blockchain technology in addition to consensus algorithms used in existing blockchains. We dived into the details of a specific implementation of the blockchain, Hyperledger Fabric (HLF). We use HLF in our prototype implementation. HLF is a permissioned blockchain and offers few advantages over other types of implementations like allowing pluggable consensus and writing chaincode (smart contracts for HLF) using any general purpose language.

Most existing works for remote data integrity checking rely on a third-party entity to perform the audit (verification) but such solutions require just as much trust as providers for the task. Solutions without a trusted third-party fail to fulfill our desire to have a two-sided verification, i.e. they do not allow verification by both parties.

In our work, we proposed to anchor a piece of evidence for data integrity in a ledger and guarantee that the tenant and the provider will have the same block ID from the ledger, indicating the same value. It should be noted that having such a value cannot directly resolve a conflict between providers and tenants. However, it could help in the process of conflict resolution. One way can be in the process of legal action.

The verification protocol requires tenants to generate a fixed number of random nonce values and hash the data with each value before sending it to the cloud. The verification is performed by asking the provider to perform hash over the data and compare the return result with a locally stored value. Previously generated nonce values are used to force providers in performing a fresh hash computation in every verification request. Our process guarantees that the tenant and the provider will hold the same hash value which can be used to resolve the disagreement between a tenant and a provider.

We implemented a prototype for the proposed SLA verification process and run experiments to do a performance evaluation. We have measured the time required for three basic operations namely hashing, uploading data and publishing the hash value into the ledger. The time overhead to perform a verification is the result of the hashing operation which takes around 6% of the total time. In the context of SLA verification, we believe this is an acceptable overhead compared to the consequences of not having integrity check. The impact of the observed overhead should be analyzed given use cases; for example, if the use case is in time-sensitive applications, a 6% overhead may not be acceptable. Since the output of the hash function is always the same size, the time to publish it is constant.

We finalized the chapter by providing a discussion and analysis of a few points that are related to the proposed method. The chapter showed the advantages of having secure components in the SLA life-cycle management. Using a secure, trusted and distributed storage to keep a piece of evidence for data integrity, it is possible to reduce the need for trust between providers and tenants in SLA verification. With the same approach, other secure components can be used to reduce and remove unnecessary trust between SLA participants.

Chapter 7

Conclusion and Future Work

In this chapter we present a summary of the thesis by correlating our contributions with the objectives described in Chapter 1. We also present a discussion on how to further extend the works presented in this thesis.

7.1 Conclusion

In this thesis, we presented a user-centric security monitoring service for clouds. We designed and implemented mechanisms to define a security monitoring objective which is relevant for the tenant, achievable for the provider, and verifiable by both tenants and providers. The proposed method follows the user-centric design principles i.e users have control, choices, and flexibility throughout the process. Such an approach is achieved through Service Level Agreements (SLAs). The SLA passes through different phases of life-cycle. By allowing the users to be part of the SLA life-cycle and by including them in the decision making process it is possible to have a user-centric security monitoring setup in the cloud.

As presented in Section 1.2, we add security guarantees for cloud services through an SLA. At the same time, the method should take the users requirements into account, and it should be implementable by the providers. In the cloud, every tenant has a set of distinct requirements. Hence, taking the requirements of users indicate providing a tenant-specific service. From the providers perspective, the method should help to implement the security monitoring SLA for multiple tenants, even when the requirements from those tenants are different or conflicting with one another.

We proposed guaranteeing the performance of security monitoring devices, which provides an indirect solution to the issue of security SLAs. Indeed rather than directly guaranteeing security properties in SLA, we guarantee probes that are used to monitor those properties. This is addressed by including terms into SLA describing the performance of a security monitoring device while configured to monitor vulnerabilities that are relevant for the tenant. This is a complementary approach to other proposed security SLAs where, as described in Section 3.1, security properties are used in SLA terms.

We proposed guaranteeing the performance of security monitoring devices, which provides an indirect solution to the issue of security SLAs. Indeed rather than directly guaranteeing the security properties in SLAs, we guarantee the probes that are used to monitor those properties. This is addressed by including terms into SLAs describing the performance of a security monitoring device configured to monitor vulnerabilities that are relevant for the tenant. This is a complementary approach from other proposed security SLAs where, as described in Section 3.1, security properties are used in SLA terms.

In the following sections, we present conclusions on our contributions.

7.1.1 Security Monitoring SLA Language

In practice, adding terms into SLAs means performing the required tasks in the different phases of the SLA life-cycle. The life-cycle of an SLA starts by defining the terms to be included in the SLA. We focused on a specific security monitoring device, NIDS. To define terms that guarantee the performance of NIDSs, we proposed *ECSLA*. ECSLA is an extension to the CSLA language, and it is used to describe the performance of NIDSs in clouds formally. ECSLA inherits the *fuzziness* and *confidence ratio* properties

which give flexibility for SLOs without violating the terms. In addition, ECSLA adds features to define users requirement in terms of vulnerability and relationship between vulnerabilities and other services.

To describe the performance of NIDSs we used the C_{ID} metric which is single-valued metric which takes the NIDS operational environment into account, specifically it takes the rate of attack occurrence (base rate) into account. Since the value of the base rate cannot be determined before the occurrence of an attack, our SLA definition provides an estimation method. The method uses known performance values of an NIDS with known base rate values and provides an estimation model for unknown base rate values.

We performed experiments to show the applicability of the proposed estimation modeling process.For our experiment, we used vulnerable applications and real attacks to exploit the vulnerabilities. We wrote and collected rules for Snort [13] which is used as NIDS for our experiment. From the experiment, we measured the performance of an NIDS using known values of the base rates. Using the result, we prepared a model which is a function of the base rate (B), i.e given a new value of B the function estimates the performance of the NIDS in terms of TPR and FPR.

Our SLA definition language achieves most of the objectives listed in Section 1.2. The language allows describing security monitoring services in a user-centric manner. Additionally, the SLA definition process uses a metric which describes the effectiveness of a security monitoring device while taking relevant parameters into account. However, it lacks a high level language to describe the tenants' requirements. Currently, a tenant is expected to list the vulnerabilities that she/he wanted to be monitored. This is not an optimal solution, as most cloud tenants may have no knowledge about the vulnerabilities in their software.

7.1.2 Preparing SLA Templates

To address the specific needs of a tenant, providers should estimate the performance of their NIDS on different configurations, specifically on different types of rules to monitor vulnerabilities. In addition, configuring an NIDS with multiple rules to monitor multiple vulnerabilities has an effect on its performance. We introduced *interference* to describe the effect of a rule(s) for a given vulnerability on another rule(s) monitoring other vulnerability. The set of interferences between a given rule and other rules form an *interference vector (IV)* and a group of *IV* forms an *interference matrix (IM)*. IV and IM are used to estimate the performance of an NIDS configured with specific rules.

Since the number of vulnerabilities is large, the number of required operations to build the interference matrix is large. As a solution, we proposed an efficient knowledge base building mechanism. The mechanism helps to reduce the number of required operations by clustering vulnerabilities using heuristics. The performance of an NIDS on a given set of rules provides a lower bound for the NIDS performance on any subset of those rules. The result of the clustering process is used to build the interference matrix which in turn is used to estimate the NIDS performance.

Experiments are performed to show the feasibility of rule clustering in order to reduce the number of required performance tests. For the experiment, we used the *registered rule set version 2.9.8.0* from Snort and performed clustering using different heuristics, namely the *application type, attack class type, severity of the vulnerability and applications working together*. For example, using the rule set for Snort which contains 10628 rules, the number of required operations to build an interference matrix is more than 54 million. Using the proposed clustering method with the "application group" heuristic, we can reduce the required number of operations to 1378.

The theoretical work presented above achieves the objective stated in Section 1.2. We set an objective to develop an efficient and practical method to prepare a tenant specific SLA before advertising for potential tenants. However, our work lacks experimental evaluation to support the theoretical results. Section 7.2.1 describes the remaining tasks as future work.

7.1.3 SLA Verification

Once the SLA is defined it should be implemented and there should be a verification mechanism for participants in the agreement. Such mechanisms provide assistance to check fulfillment of SLOs and if there is any violation, to hold the violating party accountable. This phase is required in order to claim a penalty in cases of SLA violation. We presented an NIDS performance evaluation method which is used for SLO verification. The verification is applied for SLAs guaranteeing the performance of NIDSs.

The method uses a dynamic attack injection campaign on a given configuration. The injected attacks are redirected to a specific target machine, which is carefully prepared to mimic the network behavior of

production VMs. The attack injection algorithm is unique in a way that it dynamically respects a given base rate. To successfully use this method cooperation between tenants and providers is required, i.e. tenants need to disclose the applications and few details running in their environment and providers need to configure the infrastructure in order to prepare it for the attack injection and provide an untampered output of monitoring devices.

The campaign sends both legitimate and attack packets. It generates two outputs, (i) the set of packets injected in the campaign (ii) the output of the NIDS on injected packets. We have designed a metric evaluator which takes these two outputs and compute the performance of the NIDS over the injected packets. The evaluator takes sets of packets, reconstruct sessions, and categorize those sessions into attack and legitimate groups using prior knowledge about injected packets. From the output of the NIDS, packets are extracted and matched with sessions. Using this process, basic metrics are computed and the input base rate value is used to compute the C_{ID} metric.

A prototype of our verification method is implemented using existing tools and new modules are written in Python and Java programming languages. Injecting packets into a production environment adds overhead on the network infrastructure. We performed experiments to measure this overhead and we showed the method can introduce an overhead of less than 10% on the response time of production services. The other affecting factor to perform verification of NIDS performance is time. Since the evaluation is multi-processed, the number of processes to perform the task determines the time required for the overall operation. The higher the number of processes the less time needed for the evaluation. But the overhead in the network also increases with an increasing number of processes. We showed the tradeoff between the number of processes and the required time to do verification.

As described above, the proposed verification requires cooperation between tenants and providers. In the context of SLAs, achieving unbiased cooperation between tenants and providers is difficult. This is mainly because violating an SLA has consequences, hence participants are incentivized not to violate SLAs. For example, if a monitoring device is not performing as expected a provider may alter the output of that monitoring device before providing the result to tenants. In order to avoid the need for such cooperation we proposed to use a logical third party which can resolve disagreements between tenants and providers.

We proposed an SLA verification method with a logical third party as solution to remove dependency between tenants and providers in the verification process. For this work we consider an SLA guaranteeing data integrity in clouds. Tenants use storage-as-a-service cloud service provider to store their data and they require to be monitored against any change on the data. The SLA provides guarantee for tenants on data integrity.

The monitoring mechanism uses a *secure and trusted distributed ledger*, also known as *blockchain*, to store evidence and use that evidence to resolve conflicts between tenants and providers. The method uses a consortium blockchain network where the participants are tenants and providers. We proposed a protocol which contains two phases. In the first phase tenants generate cryptographic tags to be used later for verification, upload the data to the cloud and commit evidence to the ledger. In the second phase tenants request the provider to confirm integrity of the data. Providers perform operation over the current state of the data and reply to tenants. Then, tenants can check the correctness of the result which confirms integrity of the data. Using our protocol tenants can easily check data integrity without relying on the provider. In cases of any discrepancy the ledger can be used to resolve disagreements.

We implemented the proposed protocol and performed experiments to measure the overhead introduced by the verification method. We used Hyperledger Fabric [18] to build the back-end blockchain. The cryptographic tasks in our verification process add overhead on the communication between the service provider and tenants. In addition, a constant time overhead is introduced as a result of publishing evidence into the ledger. In total an average of 6% time overhead is introduced in the communication between tenants and providers as a result of our verification method.

In the next section, we present possible extensions to the work presented in this thesis.

7.2 Future Work

In this section we present future works. It is categorized into three sections. Section 7.2.1 describes future works that can be achieved in a short time period and focuses on performance, design and implementation improvements. Section 7.2.2 describes goals that can be achieved in a medium time range and focuses mainly on the remaining tasks of the SLA life-cycle like SLA enforcement. Finally, Section 7.2.3 presents our perspective on the future of user-centric and fully automated security monitoring setup in clouds.

7.2.1 Short Term

In this section we describe performance, design and implementation improvements that can be done on the proposed methods.

In our SLA definition tenants are expected to describe their needs in terms of software vulnerabilities. As described in Section 3.6, this is not an optimal solution because usually, tenants have no detailed information about the vulnerabilities of their software. In order to address this issue, an extension can be done for our SLA definition mechanism. The extension connects our SLA definition mechanism with the vulnerability scanning and fuzzing methods (e.g OpenVAS [164]). Ideally, users specify the software they need to deploy, the vulnerability scanner and the fuzzer check if there are vulnerabilities or security loopholes in the software and list the existing vulnerabilities with detailed information. Then tenants can select their desired vulnerabilities, and the result can be passed to the SLA definition method.

Practically there are challenges to achieve the connection between vulnerability scanners and SLA definition methods. Specially, considering the diversity of existing applications standardizing the scanning method and communicating with SLA components are few examples. In addition, generalized automation of a vulnerability scanning method for custom developed applications also requires more study.

In our work we considered a specific security monitoring probe, a signature-based NIDS and Snort [13] is used in our experiments. The output of an NIDS is processed by the *metrics evaluator* component which is not specific to Snort. Any type of NIDS can be used by implementing the appropriate parser to the output of the NIDS. Anomaly-based IDSs could be addressed by extending the SLA definition method. Specifically, the parameters used to describe performance should be adapted for the type of IDS used.

Extending for anomaly-based IDS opens the opportunity to consider monitoring for unknown vulnerabilities. Including unknown vulnerabilities requires extending the service definition section of ECSLA. Currently, a list of known vulnerabilities is defined and the performance of the NIDS is guaranteed on those vulnerabilities.

ECSLA can also be extended to describe the performance of security monitoring probes other than NIDSs (e.g firewalls). Depending on the device described in the SLA, adaptation may be required for different sections in the SLA; like an adaptation for the description of users requirements, for the guarantees, metrics, and verification method to that specific device. For example, the security monitoring service definition for the firewalls may be expressed in different forms than for the NIDSs and outputs of the firewalls are not the same as the output of NIDS. As a result, a parser implementation can be required to process the output of firewalls.

In our work we introduced the theoretical concept of rules interference (*interference vector and matrix*). A practical evaluation of rule interference is not presented. Collecting a large and enough number of attacks with their corresponding rules to perform experiments would validate our theoretical work. The experiment can be done following the attack injection method presented in Chapter 5 and the result can be used to build the interference vector and matrix. In addition, while introducing interference the assumption was NIDS giving output for every matching rules (as described in Section 4.6.2). Other cases of interference (e.g NIDS giving outputs for few of the matchings based on some heuristics) are not considered. Having a method with all possible cases makes the formula more consistent and this can be achieved with more studies.

Regarding our experiments few scenarios are not considered. Notable examples of such scenario include multiple tenants sharing a monitoring device and dynamic events that can affect the verification process. In our experiment we considered a single tenant using a dedicated NIDS but a service provider may use a single NIDS to monitor multiple tenants' environments. In addition, we did not include dynamic events, like the creation of new a VM, in our experiments. These scenarios can further strengthen our results and can be achieved by just doing more experiments. In addition, the attacks used in our experiments are somehow limited to application level attacks. As described in Section 5.7.3, there are few types of attacks that can affect the verification process itself. Other than that, by collecting and preparing attacks with their corresponding rules more experiments can be performed, in order to include scenarios like the ones described above.

For data integrity SLAs, we presented the monitoring method without describing the SLA definition method. This was a result of the context where the work is proposed (internship work) and the goal we wanted to show, i.e using secure components in the SLA verification process. ECSLA can be extended and SLA definition for data integrity can be done following a similar method to NIDS SLA definition. It requires studying the performance metrics and users requirement description mechanism for the data integrity property.

7.2.2 Medium Term

In this section we present future works that may require a longer time, i.e require detailed study, than the tasks listed in the previous section. The discussion includes the remaining phase in our SLA life-cycle, i.e SLA enforcement, considering variable length of vulnerabilities to be monitored and testing the proposed method in real production environments.

As described in Section 5.1.1 we used simple SLA enforcement mechanisms for NIDS. For every vulnerability proposed in the SLA template there are rules corresponding to that vulnerability. For vulnerabilities selected by tenants, our SLA enforcement simply adds the corresponding rules to the NIDS. In practice, there are more complicated scenarios and a more in depth study is required to have an optimal security monitoring SLA enforcement mechanism. For example, let us assume a provider uses one IDS per physical machine and tenants' VMs can be spread across multiple physical machines. In such case careful analysis is required to add the correct rule in the correct place. Another example could be when tenants share a single NIDS, the NIDS may monitor the same vulnerability for different tenants. In that case it may be better to update the IP list of existing rules rather than adding the same rule with different IP addresses. Of course while doing such kind of changes, the effect of the new rules on the performance of the NIDS should be considered. By doing more studies various cases of enforcement with different requirements can be addressed.

In our SLA definition and verification process we assumed a fixed number of vulnerabilities, i.e the number of vulnerabilities to be monitored is fixed at the time of SLA definition. But in practice a vulnerability can be obsolete, i.e when a patch is available for a vulnerability (assuming the tenant applies the patch) it becomes useless to monitor that vulnerability. In addition, a new vulnerability which is relevant for a tenant can be discovered after the SLA definition phase. In our work to address such issues it requires restarting the SLA life-cycle. In the future, an automatic way for increasing and decreasing the number of vulnerabilities can be applied. There is an ongoing research activity to address issues related to dynamic events in the security monitoring process [73].

Our SLA verification process encounters difficulty while evaluating for very small base rate values, i.e it requires large resources to measure performance with a very low base rate. To get statistically convincing results the attack injection needs to run for multiple rounds. When the base rate is very low, running the experiment for multiple rounds takes a long time. In addition, storing the packet transmitted between an attacker and a target machine takes a large amount of disk space. To reduce the time it is possible to increase the number of processes which are running in parallel, but this helps until the maximum network bandwidth. If we pass the available network resource, it may create congestion and possible packet drops from the NIDS. To address this it requires modifying our verification method. For example, performing the metrics computation process directly, while the attack injection is ongoing, could help to address the disk space issue.

In our work almost all experiments are done on the Grid5000 [22] testbed. Grid5000 is designed for reproducible experiment and the user is in control of physical machines. As a future work , testing and deploying the proposed methods in production environments will greatly support the results found in experimentations.

7.2.3 Long Term

In this section we present our vision on the idea of user-centric security monitoring in clouds.

In the long term user-centric security monitoring should be provided in an automated fashion. Specifically, SLA-based security monitoring should support an easy and intuitive way to describe users requirement. This requires addressing the semantic gap between a user intuitive language and machine understandable languages. Addressing the challenge of this semantic gap paves the way for fully automated SLA-based security monitoring.

Tenants will describe their needs in the most natural way possible and the requirement will not only apply to a specific monitoring probe but to the entire security monitoring setup. To accomplish such kind of enforcement an adaptive method where a requirement can be interpreted for different types of devices and appropriate configuration can be generated for each components is required. The semantic gap problem is a big challenge in order to address such a vision.

The enforcement will be done on every component of the monitoring system with check and balance between components. A component can send feedback if a neighboring component is behaving in an unexpected manner. The verification will also be automated with live reporting of results. The challenge to make this vision a reality is lack of an efficient method to monitor events (not specific to security) and

get information inside systems. As the functionality of systems get more complex, extracting information for SLA-based security monitoring becomes more challenging.

Automated security monitoring will be combined with prevention mechanisms in order to automatically respond to intrusions and protect tenants' assets. The monitoring mechanism will cope up with new vulnerabilities and dynamically adjusts SLO. Any violation will be addressed automatically i.e if possible restore a correct service level and apply penalty automatically or report and halt the monitoring process. In summary easy requirement description, an efficient method to extract information for systems, device independent and automated enforcement and verification method will facilitate an effective user-centric security monitoring process.

The other big challenge in such an automated system is the human component, since, it is a well-known fact that as long as there is a human component in the information system security process, that is the weakest link of the system. By nature humans make mistakes and it is almost impossible to prevent such kind of errors. To address this a challenging way would consist in totally removing the human component from the security monitoring SLA life-cycle.

Thus, doing automation for security purposes, ideally without the human component in the system, is challenging.

Appendix A

Appendix

A.1 XML schema of ECSLA

```
1 <?xml version ="1.0" encoding ="utf-8"?>
2 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
3 numberstargetNamespace ="http://www.inria.fr/cslamodel "
4 numberselementFormDefault ="qualified ">
5
6 <xs:annotation>
7   <xs:appinfo>ECSLA, a DSL for security monitoring SLA description in cloud</xs:appinfo>
8   <xs:documentation xml:lang="en">
9   Schema for the ECSLA, an extention for CSLA Language.
10  Version: 1.0
11  Author: Amir Teshome Wonjiga
12  </xs:documentation>
13 </xs:annotation>
14
15 <xs:complexType numbersname=" CloudServiceType ">
16   <xs:choice >
17     <xs:element numbersname=" software " numberstype="csla: SoftwareType "
18     numbersminOccurs ="1" numbersmaxOccurs ="1"/>
19     <xs:element numbersname=" platform " numberstype="csla: PlatformType "
20     numbersminOccurs ="1" numbersmaxOccurs ="1"/>
21     <xs:element numbersname=" infrastructure " numberstype="csla: InfrastructureType "
22     numbersminOccurs ="1" numbersmaxOccurs ="1"/>
23     <xs:element numbersname="macro" numberstype="csla:any" numbersminOccurs ="0"
24     numbersmaxOccurs ="1"/>
25   </xs:choice >
26 </xs:complexType >
27
28 <xs:complexType numbersname=" SecurityMonitoringType ">
29   <xs:sequence >
30     <xs:element numbersname=" product " numberstype="csla: ProductType " numbersminOccurs ="1"
31     numbersmaxOccurs =" unbounded "/>
32     <xs:element numbersname=" vulnerabilities " numberstype="csla: VulnerabilitiesType "
33     numbersminOccurs ="0" numbersmaxOccurs =" unbounded "/>
34     <xs:element numbersname=" infrastructure " numberstype="csla: InfrastructureType "
35     numbersminOccurs ="1" numbersmaxOccurs =" unbounded "/>
36   </xs:sequence >
37 </xs:complexType >
38
39 <xs:complexType numbersname=" ProductType ">
40   <xs:sequence >
41     <xs:element numbersname="mode" numberstype="csla: ModeType " numbersminOccurs ="1"
42     numbersmaxOccurs =" unbounded "/>
43   </xs:sequence >
44   <xs:attribute numbersname="id" numberstype="xs:string "/>
45   <xs:attribute numbersname="name" numberstype="xs:string "/>
46   <xs:attribute numbersname="mode" numberstype="xs:integer "/>
47   <xs:attribute numbersname=" version " numberstype="xs:string "/>
48   <xs:attribute numbersname=" distribution " numberstype="xs:string "/>
49   <xs:attribute numbersname="price" numberstype="xs:string "/>
50   <xs:attribute numbersname=" license " numberstype="xs:string "/>
51 </xs:complexType >
```

```
52
53 <xs:complexType numbersname=" VulnerabilitiesType ">
54   <xs:sequence >
55     <xs:element numbersname=" vulnerability " numberstype="csla: VulnerabilityType "
56     numbersminOccurs ="1" numbersmaxOccurs =" unbounded "/>
57     </xs:sequence >
58 </xs:complexType >
59
60 <xs:complexType numbersname=" VulnerabilityType ">
61   <xs:attribute numbersname="id" numberstype="xs:string "/>
62   <xs:attribute numbersname="cve" numberstype="xs:string "/>
63   <xs:attribute numbersname=" description " numberstype="xs:string "/>
64 </xs:complexType >
65
66
67 <xs:complexType name="MetricType">
68   <xs:sequence>
69     <xs:element name="description" type="xs:string" minOccurs="1" maxOccurs="1"/>
70     <xs:element name="formulation" type="xs:string" minOccurs="0" maxOccurs="1"/>
71   </xs:sequence>
72   <xs:attribute name="id" type="xs:string"/>
73   <xs:attribute name="name" type="xs:string"/>
74   <!-- Type can be simple of complex -->
75   <xs:attribute name="type" type="xs:string"/>
76   <xs:attribute name="unit" type="xs:string"/>
77 </xs:complexType>
78 </xs:schema >
```

Listing A.1: ECSLA XML schema

Bibliography

[1] *A tool for dumping the contents of Unified2 log files to stdout*. accessed July 2018. URL: http://manpages.ubuntu.com/manpages/xenial/man8/u2spewfoo.8.html.

[2] Ali Al-Shabibi et al. "OpenVirteX: A Network Hypervisor." In: *ONS*. 2014.

[3] *AlienVault OSSI, Open Source Security Information and Event Management (SIEM)*. accessed July 2018. URL: https://www.alienvault.com/products/ossim.

[4] *Amazon CloudWatch*. accessed July 2018. URL: https://aws.amazon.com/cloudwatch/.

[5] *Amazon Compute Service Level Agreement*. accessed August 2018. URL: https://aws.amazon.com/compute/sla/.

[6] *Amazon EC2 Spot Instances*. accessed Jun 2018. URL: https://aws.amazon.com/ec2/spot/.

[7] *Amazon Elastic Compute*. accessed July 2018. URL: https://aws.amazon.com/ec2/.

[8] *Amazon S3*. accessed July 2018. URL: https://aws.amazon.com/s3/.

[9] *Amazon S3 Service Disruption in the Northern Virginia (US-EAST-1) Region*. accessed July 2018. URL: https://aws.amazon.com/message/41926/.

[10] *AMD Platform Security Processor (PSP)*. accessed July 2018. URL: https://www.amd.com/en/technologies/security.

[11] *AMD-V Technology for Client Virtualization*. accessed July 2018. URL: https://www.amd.com/en/technologies/virtualization.

[12] *An Open Global Forum for Advanced Distributed Computing*. accessed July 2018. URL: https://www.ogf.org/ogf/doku.php.

[13] *An open source intrusion detection and prevention system*. accessed August 2018. URL: https://www.snort.org/.

[14] *An open standard for creating a timestamp proof of any data, file, or process*. accessed July 2018. URL: https://chainpoint.org/.

[15] David Andersen et al. *Resilient overlay networks*. Vol. 35. 5. ACM, 2001.

[16] Thomas Anderson et al. "Overcoming the Internet impasse through virtualization". In: *Computer* 38.4 (2005), pp. 34–41.

[17] Alain Andrieux et al. "Web services agreement specification (WS-Agreement)". In: *Open grid forum*. Vol. 128. 1. 2007, p. 216.

[18] Elli Androulaki et al. "Hyperledger fabric: a distributed operating system for permissioned blockchains". In: *Proceedings of the Thirteenth EuroSys Conference*. ACM. 2018, p. 30.

[19] ARM ARM. "Security technology building a secure system using trustzone technology (white paper)". In: *ARM Limited* (2009).

[20] *Avast Antivirus*. accessed July 2018. URL: https://www.avast.com/index#pc.

[21] Stefan Axelsson. "The base-rate fallacy and its implications for the difficulty of intrusion detection". In: *Proceedings of the 6th ACM Conference on Computer and Communications Security*. ACM. 1999, pp. 1–7.

[22] Daniel Balouek et al. "Adding Virtualization Capabilities to the Grid'5000 Testbed". In: *Cloud Computing and Services Science*. Vol. 367. Communications in Computer and Information Science. Springer International Publishing, 2013, pp. 3–20. ISBN: 978-3-319-04518-4. DOI: 10.1007/978-3-319-04519-1_1.

[23] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS operating systems review*. Vol. 37. 5. ACM. 2003, pp. 164–177.

[24] Salman A Baset. “Cloud SLAs: present and future”. In: *ACM SIGOPS Operating Systems Review* 46.2 (2012), pp. 57–66.

[25] Richard Bejtlich. *The Tao of network security monitoring: beyond intrusion detection*. Pearson Education, 2004.

[26] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[27] Orna Agmon Ben-Yehuda Muli Ben and Yehuda Assaf Schuster Dan Tsafrir. *The Resource-as-a-Service (RaaS) Cloud*.

[28] Karin Bernsmed, Martin Gilje Jaatun, and Astrid Undheim. “Security in service level agreements for cloud computing”. In: *CLOSER 2011-Proceedings of the 1st International Conference on Cloud Computing and Services Science*. 2011.

[29] Karin Bernsmed et al. “Security SLAs for federated cloud services”. In: *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*. IEEE. 2011, pp. 202–209.

[30] *Bochs*. accessed July 2018. URL: `http://bochs.sourceforge.net/`.

[31] Zdravko Bozakov and Panagiotis Papadimitriou. “Autoslice: automated and scalable slicing for software-defined networks”. In: *Proceedings of the 2012 ACM conference on CoNEXT student workshop*. ACM. 2012, pp. 3–4.

[32] *Brief History of Virtualization*. `https://docs.oracle.com/cd/E26996_01/E18549/html/VMUSG1010.html`. Accessed: June 2018.

[33] Vitalik Buterin et al. “A next-generation smart contract and decentralized application platform”. In: *white paper* (2014).

[34] Vitalik Buterin et al. “Ethereum white paper, 2014”. In: *URL https://github. com/ethereum/wiki/wiki/White-Paper* (2013).

[35] Alvaro A Cárdenas and John S Baras. “B-ROC curves for the assessment of classifiers over imbalanced data sets”. In: *Proceedings of the National Conference on Artificial Intelligence*. Vol. 21. 2. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999. 2006, p. 1581.

[36] Alvaro A Cárdenas, John S Baras, and Karl Seamon. “A framework for the evaluation of intrusion detection systems”. In: *Security and Privacy, 2006 IEEE Symposium on*. IEEE. 2006, 15–pp.

[37] Alvaro A Cardenas, Pratyusa K Manadhata, and Sreeranga P Rajan. “Big data analytics for security”. In: *IEEE Security & Privacy* 11.6 (2013), pp. 74–76.

[38] Martin Casado et al. “Ethane: Taking control of the enterprise”. In: *ACM SIGCOMM Computer Communication Review*. Vol. 37. 4. ACM. 2007, pp. 1–12.

[39] Valentina Casola, Alessandra De Benedictis, and Massimiliano Rak. “Security monitoring in the cloud: an SLA-based approach”. In: *Availability, Reliability and Security (ARES), 2015 10th International Conference on*. IEEE. 2015, pp. 749–755.

[40] Valentina Casola et al. “Automatically enforcing security slas in the cloud”. In: *IEEE Transactions on Services Computing* 10.5 (2017), pp. 741–755.

[41] Miguel Castro, Barbara Liskov, et al. “Practical Byzantine fault tolerance”. In: *OSDI*. Vol. 99. 1999, pp. 173–186.

[42] Antonio Celesti et al. “How to enhance cloud architectures to enable cross-federation”. In: *2010 IEEE 3rd international conference on cloud computing*. IEEE. 2010, pp. 337–345.

[43] CERT Coordination Center. “TCP SYN flooding and IP spoofing attacks”. In: *CERT Advisory CA-1996-21* (1996), pp. 1996–2021.

[44] Shirlei Aparecida de Chaves, Carlos Becker Westphall, and Flavio Rodrigo Lamin. “SLA perspective in security management for cloud computing”. In: *Networking and Services (ICNS), 2010 Sixth International Conference on*. IEEE. 2010, pp. 212–217.

[45] Lin Chen et al. “On security analysis of proof-of-elapsed-time (poet)”. In: *International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Springer. 2017, pp. 282–297.

[46] Thomas M Chen and Alden W Jackson. "Active And Programmable Networks [Guest Editorial]". In: *IEEE Network* 12.3 (1998), pp. 10–11.

[47] Tsung-Huan Cheng et al. "Evasion techniques: Sneaking through your intrusion detection/prevention systems". In: *IEEE Communications Surveys & Tutorials* 14.4 (2012), pp. 1011–1020.

[48] William R Cheswick, Steven M Bellovin, and Aviel D Rubin. *Firewalls and Internet security: repelling the wily hacker*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[49] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. "A survey of network virtualization". In: *Computer Networks* 54.5 (2010), pp. 862–876.

[50] NM Mosharaf Kabir Chowdhury and Raouf Boutaba. "Network virtualization: state of the art and research challenges". In: *IEEE Communications magazine* 47.7 (2009).

[51] Cristina Cifuentes and Vishv M Malhotra. "Binary translation: Static, dynamic, retargetable?" In: *icsm*. 1996, pp. 340–349.

[52] *Cloud File Sharing and Storage for your Business*. accessed July 2018. URL: `https://www.dropbox.com/`.

[53] *Cloudstatus: Monitoring Cloud Infrastructure*. accessed July 2018. URL: `https://cloudstatus.eu/`.

[54] Fred Cohen. "Computer viruses". In: *Computers & security* 6.1 (1987), pp. 22–35.

[55] *Common Criteria for Information Technology Security Evaluation, Part 1: Introduction and general model*. Version 3.1, Revision 5. URL: `https://www.commoncriteriaportal.org/cc/`.

[56] *Common Vulnerabilities and Exposures*. accessed July 2018. URL: `https://cve.mitre.org/`.

[57] *Corrupt iCloud data causes iOS home screen crash*. accessed July 2018. URL: `https://www.macobserver.com/tmo/article/corrupt-icloud-data-can-cause-ios-springboard-home-screen-crash`.

[58] Victor Costan and Srinivas Devadas. "Intel SGX Explained." In: *IACR Cryptology ePrint Archive* 2016 (2016), p. 86.

[59] Thomas M Cover and Joy A Thomas. *Elements of information theory*. John Wiley & Sons, 2012.

[60] Carlos Alberto Da Silva, Anderson Soares Ferreira, and Paulo Licio de Geus. "A methodology for management of cloud computing using security criteria". In: *Cloud Computing and Communications (LATINCLOUD), 2012 IEEE Latin America Conference on*. IEEE. 2012, pp. 49–54.

[61] Carlos Alberto Da Silva and Paulo Licio de Geus. "An approach to security-SLA in cloud computing environment". In: *Communications (LATINCOM), 2014 IEEE Latin-America Conference on*. IEEE. 2014, pp. 1–6.

[62] Mehiar Dabbagh et al. "Software-defined networking security: pros and cons". In: *IEEE Communications Magazine* 53.6 (2015), pp. 73–79.

[63] Eli Dart et al. "The science dmz: A network design pattern for data-intensive science". In: *Scientific Programming* 22.2 (2014), pp. 173–185.

[64] Amir Vahid Dastjerdi, Kamalrulnizam Abu Bakar, and Sayed Gholam Hassan Tabatabaei. "Distributed intrusion detection in clouds using mobile agents". In: *Advanced Engineering Computing and Applications in Sciences, 2009. ADVCOMP'09. Third International Conference on*. IEEE. 2009, pp. 175–180.

[65] Alessandra De Benedictis et al. "Rest-based SLA management for cloud applications". In: *2015 IEEE 24th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises*. IEEE. 2015, pp. 93–98.

[66] *Decentralized cloud object storage that is affordable and easy to use*. accessed July 2018. URL: `https://storj.io/`.

[67] *Digital Ocean*. accessed July 2018. URL: `https://www.digitalocean.com/`.

[68] *Docker*. accessed July 2018. URL: `https://www.docker.com/`.

[69] Avri Doria et al. *Forwarding and control element separation (ForCES) protocol specification*. Tech. rep. 2010.

[70] *DriveHQ Service Level Agreement*. accessed July 2018. URL: `https://www.drivehq.com/premium/DriveHQSLA.aspx`.

[71] Dmitry Drutskoy, Eric Keller, and Jennifer Rexford. "Scalable network virtualization in software-defined networks". In: *IEEE Internet Computing* 17.2 (2013), pp. 20–27.

[72] James P Egan. *Signal Detection Theory and ROC Analysis Academic Press Series in Cognition and Perception.* London, UK: Academic Press, 1975.

[73] Clément Elbaz, Louis Rilling, and Christine Morin. "Reactive and Adaptive Security Monitoring in Cloud Computing". In: *2018 IEEE 3rd International Workshops on Foundations and Applications of Self* Systems (FAS* W).* IEEE. 2018, pp. 5–7.

[74] Vincent C Emeakaroha et al. "Low level metrics to high level SLAs-LoM2HiS framework: Bridging the gap between monitored metrics and SLA parameters in cloud environments". In: *High Performance Computing and Simulation (HPCS), 2010 International Conference on.* IEEE. 2010, pp. 48–54.

[75] *Facebook temporarily loses more than 10% of photos in hard drive failure.* accessed July 2018. URL: https://www.computerworld.com/article/2531672/disaster-recovery/facebook-temporarily-loses-more-than-10--of-photos-in-hard-drive-failure.html.

[76] Zubair M Fadlullah et al. "DTRAB: Combating against attacks on encrypted protocols through traffic-feature analysis". In: *IEEE/ACM Transactions on Networking (TON)* 18.4 (2010), pp. 1234–1247.

[77] Dino Farinacci et al. *Generic routing encapsulation (GRE).* Tech. rep. 2000.

[78] Paul Ferguson and Geoff Huston. *What is a VPN?* 1998.

[79] *Floodlight: an Open SDN Controller.* accessed July 2018. URL: http://www.projectfloodlight.org/floodlight/.

[80] Jose Fonseca, Marco Vieira, and Henrique Madeira. "Evaluation of web security mechanisms using vulnerability and attack injection". In: *IEEE Transactions on Dependable and Secure Computing* 1 (2014), p. 1.

[81] Joint Task Force and Transformation Initiative. "Security and privacy controls for federal information systems and organizations". In: *NIST Special Publication* 800.53 (2013), pp. 8–13.

[82] US Air Force. "Analysis of the Intel Pentium's ability to support a secure virtual machine monitor". In: *Proceedings of the... USENIX Security Symposium. USENIX Association.* 2000, p. 129.

[83] Jason Franklin et al. "Attacking, repairing, and verifying SecVisor: A retrospective on the security of a hypervisor". In: *Tech. Rep. CMU-CyLab-08-008, Carnegie Mellon University* (2008).

[84] *FreeBSD Jails.* accessed July 2018. URL: https://www.freebsd.org/cgi/man.cgi?query=jail&sektion=8.

[85] Edoardo Gaetani et al. "Blockchain-based database to ensure data integrity in cloud computing environments". In: (2017).

[86] John E Gaffney and Jacob W Ulvila. "Evaluation of intrusion detectors: A decision theory approach". In: *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on.* IEEE. 2001, pp. 50–61.

[87] Simson L Garfinkel and Michael Shick. *Passive tcp reconstruction and forensic analysis with tcpflow.* Tech. rep. Monterey, California. Naval Postgraduate School, 2013.

[88] Tal Garfinkel and Mendel Rosenblum. "When Virtual Is Harder than Real: Security Challenges in Virtual Machine Based Computing Environments." In: *HotOS*. 2005.

[89] Tal Garfinkel, Mendel Rosenblum, et al. "A Virtual Machine Introspection Based Architecture for Intrusion Detection." In: *Ndss.* Vol. 3. 2003. 2003, pp. 191–206.

[90] Tal Garfinkel et al. "Compatibility Is Not Transparency: VMM Detection Myths and Realities." In: *HotOS*. 2007.

[91] *GFD.224 – Open Cloud Computing Interface – Infrastructure.* accessed July 2018. URL: http://occi-wg.org/about/specification/.

[92] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering.* Prentice Hall PTR, 2002.

[93] Anna Giannakou. "Self-adaptable Security Monitoring for IaaS Cloud Environments". Theses. INSA de Rennes, July 2017. URL: https://hal.inria.fr/tel-01653831.

[94] Robert P Goldberg. “Survey of virtual machine research”. In: *Computer* 7.6 (1974), pp. 34–45.

[95] *Google APP Engine*. accessed July 2018. URL: https://cloud.google.com/compute/.

[96] *Google APPs*. accessed July 2018. URL: https://gsuite.google.com/.

[97] *Google Compute Engine*. accessed July 2018. URL: https://cloud.google.com/compute/.

[98] Higher Layer LAN Protocols Working Group et al. *802.1 Q-2014-IEEE Standard for local and metropolitan area networks-bridges and bridged networks. IEEE Std 802.1 Q. 2014*. 2014.

[99] Guofei Gu et al. “Measuring intrusion detection capability: an information-theoretic approach”. In: *Proceedings of the 2006 ACM Symposium on Information, computer and communications security*. ACM. 2006, pp. 90–101.

[100] Natasha Gude et al. “NOX: towards an operating system for networks”. In: *ACM SIGCOMM Computer Communication Review* 38.3 (2008), pp. 105–110.

[101] Zhuo Hao and Nenghai Yu. “A multiple-replica remote data possession checking protocol with public verifiability”. In: *Data, Privacy and E-Commerce (ISDPE), 2010 Second International Symposium on*. IEEE. 2010, pp. 84–89.

[102] Zhuo Hao, Sheng Zhong, and Nenghai Yu. “A privacy-preserving remote data integrity checking protocol with data dynamics and public verifiability”. In: *IEEE transactions on Knowledge and Data Engineering* 23.9 (2011), pp. 1432–1437.

[103] Ronda R Henning. “Security service level agreements: quantifiable security for the enterprise?” In: *Proceedings of the 1999 workshop on New security paradigms*. ACM. 1999, pp. 54–60.

[104] *Heroku Runtime*. accessed July 2018. URL: https://www.heroku.com/.

[105] Steve Hoffman. *Apache Flume: distributed log collection for Hadoop*. Packt Publishing Ltd, 2013.

[106] Markus C Huebscher and Julie A McCann. “A survey of autonomic computing—degrees, models, and applications”. In: *ACM Computing Surveys (CSUR)* 40.3 (2008), p. 7.

[107] *IBM QRadar*. accessed July 2018. URL: https://www.ibm.com/security/security-intelligence/qradar.

[108] Amani S Ibrahim et al. “Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model”. In: *Network and System Security (NSS), 2011 5th International Conference on*. IEEE. 2011, pp. 113–120.

[109] Mehmet Sinan Inci et al. “Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud.” In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 898.

[110] David E Irwin, Laura E Grit, and Jeffrey S Chase. “Balancing risk and reward in a market-based task service”. In: *null*. IEEE. 2004, pp. 160–169.

[111] Raj Jain and Subharthi Paul. “Network virtualization and software defined networking for cloud computing: a survey”. In: *IEEE Communications Magazine* 51.11 (2013), pp. 24–31.

[112] Qin Jia et al. “Supercloud: Opportunities and challenges”. In: *ACM SIGOPS Operating Systems Review* 49.1 (2015), pp. 137–141.

[113] Anderson J.P. *Computer Security Threat Monitoring and Surveillance*. Tech. rep. 1980.

[114] Nesrine Kaaniche et al. “Security SLA based monitoring in clouds”. In: *2017 IEEE International Conference on Edge Computing (EDGE)*. IEEE. 2017, pp. 90–97.

[115] Steffen Kachele et al. “Beyond IaaS and PaaS: An extended cloud taxonomy for computation, storage and networking”. In: *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*. IEEE. 2013, pp. 75–82.

[116] Paul Karaenke and Stefan Kirn. “Service level agreements: An evaluation from a business application perspective”. In: *Proceedings of eChallenges*. 2007.

[117] Richard A Kemmerer and Giovanni Vigna. “Intrusion detection: a brief history and overview”. In: *Computer* 35.4 (2002), supl27–supl30.

[118] *Kernel Virtual Machine*. accessed July 2018. URL: https://www.linux-kvm.org/page/Main_Page.

[119] *Keystone, Open-source Secure Hardware Enclave*. accessed Jun 2018. URL: https://keystone-enclave.org/.

[120] Leonard Kleinrock. "A vision for the Internet". In: *ST Journal of Research* 2.1 (2005), pp. 4–5.

[121] Paul Kocher et al. "Spectre attacks: Exploiting speculative execution". In: *arXiv preprint arXiv:1801.01203* (2018).

[122] Yousri Kouki. "SLA-driven cloud elasticity anagement approach". Theses. Ecole des Mines de Nantes, Dec. 2013. URL: https://tel.archives-ouvertes.fr/tel-00919900.

[123] Yousri Kouki and Thomas Ledoux. "CSLA: a Language for improving Cloud SLA Management". In: *International Conference on Cloud Computing and Services Science, CLOSER 2012*. 2012, pp. 586–591.

[124] D Davide Lamanna, James Skene, and Wolfgang Emmerich. "Slang: A language for defining service level agreements". In: *NINTH IEEE WORKSHOP ON FUTURE TRENDS OF DISTRIBUTED COMPUTING SYSTEMS, PROCEEDINGS*. IEEE COMPUTER SOC. 2003, pp. 100–106.

[125] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 4.3 (1982), pp. 382–401.

[126] K.C. Laudon and J.P. Laudon. *Management Information Systems: Managing the Digital Firm*. Prentice Hall, 2012. ISBN: 9780132142854. URL: https://books.google.fr/books?id=uET_kQAACAAJ.

[127] C-S Li et al. "Software defined environments: An introduction". In: *IBM Journal of Research and Development* 58.2/3 (2014), pp. 1–1.

[128] Thomas A Limoncelli. "Openflow: a radical new idea in networking". In: *Communications of the ACM* 55.8 (2012), pp. 42–47.

[129] Moritz Lipp et al. "Meltdown: Reading Kernel Memory from User Space". In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018.

[130] Richard P Lippmann et al. "Evaluating intrusion detection systems: The 1998 DARPA off-line intrusion detection evaluation". In: *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*. Vol. 2. IEEE. 2000, pp. 12–26.

[131] Bin Liu et al. "Blockchain based data integrity service framework for IoT data". In: *Web Services (ICWS), 2017 IEEE International Conference on*. IEEE. 2017, pp. 468–475.

[132] Heiko Ludwig et al. "rSLA: Monitoring SLAs in dynamic service environments". In: *International Conference on Service-Oriented Computing*. Springer. 2015, pp. 139–153.

[133] Heiko Ludwig et al. "Web service level agreement (WSLA) language specification". In: ().

[134] Wenjun Luo and Guojing Bai. "Ensuring the data integrity in cloud data storage". In: *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*. IEEE. 2011, pp. 240–243.

[135] *LXC containers and security*. accessed July 2018. URL: https://linuxcontainers.org/lxc/security/.

[136] *LXC containers and security*. accessed July 2018. URL: https://linuxcontainers.org/.

[137] Huan Ma et al. "Bayes-based ARP attack detection algorithm for cloud centers". In: *Tsinghua Science and Technology* 21.1 (2016), pp. 17–28.

[138] David Marshall. "Understanding full virtualization, paravirtualization, and hardware assist". In: *VMWare White Paper* (2007), p. 17.

[139] Frederic Massicotte et al. "Automatic evaluation of intrusion detection systems". In: *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*. IEEE. 2006, pp. 361–370.

[140] Claudio Mazzariello, Roberto Bifulco, and Roberto Canonico. "Integrating a network ids into an open source cloud computing environment". In: *Information Assurance and Security (IAS), 2010 Sixth International Conference on*. IEEE. 2010, pp. 265–270.

[141] John McHugh. "Testing intrusion detection systems: a critique of the 1998 and 1999 darpa intrusion detection system evaluations as performed by lincoln laboratory". In: *ACM Transactions on Information and System Security (TISSEC)* 3.4 (2000), pp. 262–294.

[142] Nick McKeown et al. "OpenFlow: enabling innovation in campus networks". In: *ACM SIGCOMM Computer Communication Review* 38.2 (2008), pp. 69–74.

[143] Peter Mell, Tim Grance, et al. "The NIST definition of cloud computing". In: (2011).

[144] Albert Mestres et al. "Knowledge-defined networking". In: *ACM SIGCOMM Computer Communication Review* 47.3 (2017), pp. 2–10.

[145] *Metasploit: The world's most used penetration testing framework*. accessed July 2018. URL: https://www.metasploit.com/.

[146] *Microsoft Security Essentials (MSE)*. accessed July 2018. URL: https://support.microsoft.com/en-us/help/14210/security-essentials-download.

[147] *Microsoft Security Essentials (MSE)*. accessed July 2018. URL: https://www.eset.com/us/.

[148] Aleksandar Milenkoski et al. "Evaluation of intrusion detection systems in virtualized environments using attack injection". In: *International Workshop on Recent Advances in Intrusion Detection*. Springer. 2015, pp. 471–492.

[149] Aleksandar Milenkoski et al. "Quantifying the Attack Detection Accuracy of Intrusion Detection Systems in Virtualized Environments". In: *Software Reliability Engineering (ISSRE), 2016 IEEE 27th International Symposium on*. IEEE. 2016, pp. 276–286.

[150] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. "Compiling pcre to fpga for accelerating snort ids". In: *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*. ACM. 2007, pp. 127–136.

[151] *Model-based cross cloud development and deployment platform*. accessed July 2018. URL: https://paasage.ercim.eu/.

[152] *Moloch*. accessed July 2018. URL: http://molo.ch/.

[153] Christine Morin and Roberto Cascella. *Contrail final publishable summary report*. Contract. Inria Rennes Bretagne Atlantique, Apr. 2014, p. 36. URL: https://hal.inria.fr/hal-01346091.

[154] Satoshi Nakamoto. "Bitcoin: A peer-to-peer electronic cash system". In: (2008).

[155] Akihiro Nakao. "Network virtualization as foundation for enabling new network architectures and applications". In: *IEICE transactions on communications* 93.3 (2010), pp. 454–457.

[156] *Nessus*. accessed July 2018. URL: https://www.tenable.com/products/nessus/nessus-professional.

[157] *Next-Generation Intrusion Prevention System (NGIPS)*. accessed August 2018. URL: https://www.cisco.com/c/en/us/products/security/ngips/index.html.

[158] Jason Nikolai and Yong Wang. "Hypervisor-based cloud intrusion detection system". In: *Computing, Networking and Communications (ICNC), 2014 International Conference on*. IEEE. 2014, pp. 989–993.

[159] *Nmap: Network mapper utility for network discovery and security auditing*. accessed July 2018. URL: https://nmap.org/.

[160] Nicole Oldham et al. "Semantic WS-agreement partner selection". In: *Proceedings of the 15th international conference on World Wide Web*. ACM. 2006, pp. 697–706.

[161] *On Public and Private Blockchains*. accessed July 2018. URL: https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/.

[162] *Open Networking Foundation (ONF)*. accessed July 2018. URL: https://www.opennetworking.org/mission/.

[163] *Open source software for creating private and public clouds*. accessed July 2018. URL: https://www.openstack.org/.

[164] *Open Source vulnerability scanner and manager (OpenVAS)*. accessed July 2018. URL: http://www.openvas.org/.

[165] *Open vSwitch is a production quality, multilayer virtual switch*. accessed July 2018. URL: https://www.openvswitch.org/.

[166] *OpenDaylight SDN Controller*. accessed July 2018. URL: https://www.opendaylight.org/.

[167] Nouha Oualha, Jean Leneutre, and Yves Roudier. "Verifying remote data integrity in peer-to-peer data storage: A comprehensive survey of protocols". In: *Peer-to-Peer Networking and Applications* 5.3 (2012), pp. 231–243.

[168] *OWASP Zed Attack Proxy (ZAP)*. accessed July 2018. URL: https://www.owasp.org/index.php/ZAP.

[169] *Packets of Death: Vulnerability in ICMPv6 could allow Denial of Service.* accessed August 2018. URL: https://docs.microsoft.com/en-us/security-updates/SecurityBulletins/2013/ms13-065.

[170] *Papertrail Log Management.* accessed July 2018. URL: https://papertrailapp.com/.

[171] Michael Pearce, Sherali Zeadally, and Ray Hunt. "Virtualization: Issues, security threats, and solutions". In: *ACM Computing Surveys (CSUR)* 45.2 (2013), p. 17.

[172] Ronald Perez, Leendert van Doorn, and Reiner Sailer. "Virtualization and hardware-based security". In: *IEEE Security & Privacy* 5 (2008), pp. 24–31.

[173] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. "vTPM: virtualizing the trusted platform module". In: *Proc. 15th Conf. on USENIX Security Symposium.* 2006, pp. 305–320.

[174] *Performance Monitoring Solution for DevOps and IT Operations.* accessed July 2018. URL: https://www.site24x7.com/.

[175] Dana Petcu and Ciprian Craciun. "Towards a Security SLA-based Cloud Monitoring Service." In: *CLOSER.* 2014, pp. 598–603.

[176] *Physical security and the Cloud.* accessed July 2018. URL: https://morphean.com/whitepaper/.

[177] Raluca Ada Popa et al. "Enabling Security in Cloud Storage SLAs with CloudProof." In: *USENIX Annual Technical Conference.* Vol. 242. 2011, pp. 355–368.

[178] Gerald J Popek and Robert P Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[179] *Practical guide to cloud service level agreements version 2.0.* Whitepaper. Cloud Standards Customer Council (CSCC), 2015.

[180] Thibaut Probst et al. "Automated evaluation of network intrusion detection systems in iaas clouds". In: *Dependable Computing Conference (EDCC), 2015 Eleventh European.* IEEE. 2015, pp. 49–60.

[181] Thomas H Ptacek and Timothy N Newsham. *Insertion, evasion, and denial of service: Eluding network intrusion detection.* Tech. rep. SECURE NETWORKS INC CALGARY ALBERTA, 1998.

[182] Nicholas J. Puketza et al. "A methodology for testing intrusion detection systems". In: *IEEE Transactions on Software Engineering* 22.10 (1996), pp. 719–729.

[183] Elias Raftopoulos and Xenofontas Dimitropoulos. "A quality metric for IDS signatures: in the wild the size matters". In: *EURASIP Journal on Information Security* 2013.1 (2013), p. 7.

[184] Massimiliano Rak et al. "Security as a service using an SLA-based approach via SPECS". In: *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on.* Vol. 2. IEEE. 2013, pp. 1–6.

[185] Wei Ren, Randal W Beard, and Ella M Atkins. "A survey of consensus problems in multi-agent coordination". In: *Proceedings of the 2005, American Control Conference, 2005.* IEEE. 2005, pp. 1859–1864.

[186] RR Righi, F Pelissari, and C Westphall. "SEC-SLA: Specification and validation of metrics to security service level agreements". In: *IV Workshop on Computer System Security.* 2004, pp. 199–210.

[187] Damien Riquet, Gilles Grimaud, and Michaël Hauspie. "DISCUS: A massively distributed IDS architecture using a DSL-based configuration". In: *Proc. ISEEE.* 2014.

[188] Thomas Ristenpart et al. "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.* ACM. 2009, pp. 199–212.

[189] Sebastian Roschke, Feng Cheng, and Christoph Meinel. "Intrusion detection in the cloud". In: *Dependable, Autonomic and Secure Computing, 2009. DASC'09. Eighth IEEE International Conference on.* IEEE. 2009, pp. 729–734.

[190] Eric Rosen, Arun Viswanathan, and Ross Callon. *Multiprotocol label switching architecture.* Tech. rep. 2000.

[191] Christian Esteve Rothenberg et al. "When open source meets network control planes". In: *Computer* 47.11 (2014), pp. 46–54.

[192] *RRDtool is OpenSource industry standard, high performance data logging and graphing system for time series data.* accessed August 2018. URL: https://oss.oetiker.ch/rrdtool/.

[193] *rsync: an open source utility that provides fast incremental file transfer*. accessed July 2018. URL: https://rsync.samba.org/.

[194] *S3 data corruption ?* accessed July 2018. URL: https://forums.aws.amazon.com/thread.jspa?start=0&threadID=22709&tstart=0.

[195] Tomoaki Sato and Masa-aki Fukase. "Reconfigurable hardware implementation of host-based IDS". In: *Communications, 2003. APCC 2003. The 9th Asia-Pacific Conference on.* Vol. 2. IEEE. 2003, pp. 849–853.

[196] *Scalable, multi-platform, open source Host-based Intrusion Detection System (HIDS).* accessed August 2018. URL: http://www.ossec.net/index.html.

[197] Karen Scarfone and Peter Mell. "Guide to intrusion detection and prevention systems (idps)". In: *NIST special publication* 800.2007 (2007), p. 94.

[198] Fred B Schneider. "Replication management using the state-machine approach, Distributed systems". In: (1993).

[199] *Secure Provisioning of Cloud Services Based on SLA Management.* accessed July 2018. URL: http://www.specs-project.eu/.

[200] Rob Sherwood et al. "Flowvisor: A network virtualization layer". In: *OpenFlow Switch Consortium, Tech. Rep* 1 (2009), p. 132.

[201] Richard L Sites et al. "Binary translation". In: *Communications of the ACM* 36.2 (1993), pp. 69–81.

[202] James Skene. "Language support for service-level agreements for application-service provision". PhD thesis. University of London, 2007.

[203] *SLA-Ready.* accessed Jun 2018. URL: http://www.sla-ready.eu/.

[204] Mea Smith et al. "OpFlex control protocol". In: *IETF, Apr* (2014).

[205] Haoyu 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.* ACM. 2013, pp. 127–132.

[206] IDS Suricata. "open-source IDS". In: *IPS/NSM engine (http://suricata-ids. org/)* (2014).

[207] *syslog-ng: The Swiss army knife of log management.* accessed July 2018. URL: https://www.syslog-ng.com/products/open-source-log-management/.

[208] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. "Security and privacy challenges in cloud computing environments". In: *IEEE Security & Privacy* 8.6 (2010), pp. 24–31.

[209] *tcpdump: powerful command-line packet analyzer.* accessed July 2018. URL: http://www.tcpdump.org/.

[210] *The Bro Network Security Monitor.* accessed August 2018. URL: https://www.bro.org/.

[211] *Torshammer: Slow POST Denial Of Service Testing Tool.* accessed July 2018. URL: https://sourceforge.net/projects/torshammer/.

[212] Jonathan S Turner and David E Taylor. "Diversifying the internet". In: *Global Telecommunications Conference, 2005. GLOBECOM'05. IEEE.* Vol. 2. IEEE. 2005, 6–pp.

[213] Rich Uhlig et al. "Intel virtualization technology". In: *Computer* 38.5 (2005), pp. 48–56.

[214] Rafael Brundo Uriarte, Francesco Tiezzi, and Rocco De Nicola. "Slac: A formal service-level-agreement language for cloud computing". In: *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing.* IEEE Computer Society. 2014, pp. 419–426.

[215] *U.S. Code, Title 44, Chapter 35, Subchapter III, no 3542.* accessed July 2018. URL: https://www.law.cornell.edu/uscode/text/44/3542.

[216] Kleber Vieira et al. "Intrusion detection techniques in grid and cloud computing environment". In: *IT Professional, IEEE Computer Society* 12.4 (2010), pp. 38–43.

[217] *VirtualBox.* accessed July 2018. URL: https://www.virtualbox.org/.

[218] NSX VMware. *The platform for network virtualization.*

[219] *VMware Workstation*. accessed July 2018. URL: `https://www.vmware.com/products/workstation-pro.html`.

[220] Peipei Wang, Daniel J Dean, and Xiaohui Gu. "Understanding real world data corruptions in cloud systems". In: *2015 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2015, pp. 116–125.

[221] *web application attack and audit framework (w3af)*. accessed July 2018. URL: `http://w3af.org/`.

[222] Philipp Wieder et al. *Service level agreements for cloud computing*. Springer Science & Business Media, 2011.

[223] *Wireshark*. accessed July 2018. URL: `https://www.wireshark.org/`.

[224] Rafal Wojtczuk and Joanna Rutkowska. "Attacking intel trusted execution technology". In: *Black Hat DC* 2009 (2009).

[225] Xiwei Xu et al. "A taxonomy of blockchain-based systems for architecture design". In: *Software Architecture (ICSA), 2017 IEEE International Conference on*. IEEE. 2017, pp. 243–252.

[226] Yunjing Xu et al. "An exploration of L2 cache covert channels in virtualized environments". In: *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM. 2011, pp. 29–40.

[227] William A Yasnoff. "Privacy, Confidentiality, and Security of Public Health Information". In: *Public Health Informatics and Information Systems*. Springer, 2014, pp. 155–172.

[228] Yan Zhu et al. "Cooperative provable data possession". In: *Beijing: Peking University and Arizona University* (2010).

SUPERVISION DE LA SÉCURITÉ CENTRÉE SUR L'UTILISATEUR DANS UN ENVIRONNEMENT DE NUAGE INFORMATIQUE

1. Introduction

L'informatique en nuage a introduit un modèle de service différent pour les services des systèmes d'information. Par le passé, l'obtention de ressources informatiques nécessitait avant tout des investissements en capital. L'informatique en nuage réduit ou supprime ces exigences, ce qui permet aux entreprises d'utiliser plus de ressources informatiques au moment où elles en ont besoin. L'un des facteurs majeurs qui conduisent au succès de l'informatique en nuage est son modèle économique gagnant-gagnant pour les fournisseurs de services et les locataires. Pour les locataires, l'informatique en nuage offre un modèle de paiement à l'utilisation selon lequel les locataires ne doivent payer que pour les ressources qu'ils utilisent. Pour les fournisseurs, il permet de maximiser les profits en configurant et allouant dynamiquement les ressources demandées par les locataires. Les nuages informatiques offrent de nombreux autres avantages, notamment un accès omniprésent avec un service informatique modulaire.

Les services que les fournisseurs offrent dans les nuages informatiques le sont en fonction des besoins des locataires. En particulier, l'aspect fonctionnel d'un service dans un nuage informatique devient de plus en plus centré sur l'utilisateur dans la mesure où une fonction, une action ou un élément de logique métier spécifique peut être exécuté en une fraction de seconde. Toutefois, l'aspect sécurité dans les nuages informatiques est resté essentiellement générique. Fournir des propriétés de sécurité spécifiques en fonction des besoins et des exigences d'un locataire demeure un défi.

L'inconvénient des nuages informatiques découle de leurs propres caractéristiques. Lors de la migration vers un nuage informatique, le locataire perd le contrôle total de l'infrastructure physique et le fournisseur est responsable de la gestion de l'infrastructure, y compris sa sécurité. Comme cela force les locataires à compter sur les fournisseurs de services pour assurer la sécurité de leurs propres services, cela crée un problème de confiance. Les fournisseurs de services reconnaissent le problème de confiance et fournissent une garantie par le biais d'un contrat appelé *Service Level Agreement (SLA)*. Le contrat, en plus du service fourni, décrit les pénalités en cas de violation. Presque tous les SLAs existants traitent des fonctionnalités des nuages informatiques, c'est-à-dire qu'ils ne garantissent pas l'aspect sécurité des services hébergés des locataires.

La supervision de la sécurité consiste à recueillir et à analyser des indicateurs sur les menaces potentielles pour la sécurité, puis à traiter ces menaces par des actions appropriées. Comme le nuage informatique continue d'être de plus en plus intégré, il est nécessaire d'avoir des services de supervision de la sécurité adaptés à l'utilisateur qui sont fondés sur les exigences des locataires. Et comme pour les autres services, les fournisseurs doivent offrir des garanties pour l'aspect supervision de la sécurité du système.

Dans cette thèse, nous présentons notre contribution sur les questions liées à l'inclusion de termes de supervision de la sécurité centrés sur l'utilisateur dans les SLAs de l'informatique en nuage. Cela nécessite de compléter chaque phase du cycle de vie du service exécuté dans le nuage informatique, en commençant avant le déploiement effectif du service jusqu'à la fin du service. Des objectifs plus détaillés sont décrits dans le paragraphe suivant.

2. Objectifs

Le paragraphe précédent a présenté comment la conception centrée sur l'utilisateur a aidé et continue d'aider à l'adoption des nuages informatiques. Il souligne la nécessité que les tâches de supervision de la sécurité suivent la même philosophie de conception afin d'améliorer la confiance dans les services des nuages informatiques et d'accroître le succès des nuages informatiques. L'objectif global de notre travail est d'inclure des termes de supervision de la sécurité dans les SLAs des nuages informatiques.

Nous y parvenons en incluant dans les SLAs des termes (ou *Service Level Objectives*, SLO) qui garantissent la performance des dispositifs de supervision de la sécurité. Le SLO définit la performance d'un dispositif de supervision de sécurité, où le dispositif est configuré en fonction des besoins des locataires. Ensuite, l'infrastructure demandée est configurée pour atteindre les SLOs définis dans le contrat. Lors de l'utilisation du service, les participants peuvent vérifier l'infrastructure pour s'assurer que le niveau de performance attendu est atteint. Dans les cas où l'infrastructure ne fonctionne pas comme prévu, des pénalités peuvent être appliquées comme décrit dans le SLA. Dans la suite, nous définissons les objectifs des SLAs de supervision de la sécurité centrés sur l'utilisateur.

— Comme tout autre type de SLA, pour définir les SLAs de supervision de la sécurité, nous avons besoin d'un langage de SLA. Le but d'un tel langage est de faciliter et de normaliser la communication entre

les locataires et les fournisseurs. Le langage doit permettre de décrire les différentes parties composant le SLA :

(1) Il devrait permettre de décrire les services de supervision de la sécurité dans les nuages informatiques. Dans cette thèse, un service de supervision de la sécurité est assuré par un dispositif appelé système de détection d'intrusion réseau (NIDS).

(2) Comme nous visons à avoir un SLA de supervision de la sécurité centré sur l'utilisateur, le langage devrait permettre aux locataires de décrire leurs besoins spécifiques en matière de supervision de la sécurité.

En outre, le langage doit être sémantiquement équilibré entre les locataires et les fournisseurs, c'est-à-dire qu'il ne doit pas être de trop bas niveau pour être compréhensible par les locataires ordinaires des nuages informatiques, et qu'il ne doit pas être de trop haut niveau pour être capable d'automatiser la mise en œuvre.

— Afin de garantir la performance d'un dispositif de supervision de sécurité, nous avons besoin de métriques. Fournir des métriques décrivant les performances d'un dispositif de supervision de sécurité nécessite la prise en compte de différents paramètres. Il s'agit de facteurs internes et externes : par exemple, des facteurs internes comme la configuration du dispositif de supervision, qui est liée aux besoins du locataire, et des facteurs externes comme l'environnement opérationnel (le débit du réseau, le taux d'attaques, etc.) du dispositif de supervision. Deux dispositifs ayant la même configuration interne ne se comportent pas de la même façon lorsqu'ils fonctionnent dans des environnements externes différents. Par exemple, il est prouvé que le taux d'occurrence d'attaques affecte grandement la performance des NIDS.
— Les paramètres utilisés dans les SLOs doivent être pertinents pour les locataires. Dans notre définition de SLA, les SLOs décrivent la performance des dispositifs de supervision de la sécurité. Plus précisément, une mesure qui décrit la performance d'un dispositif de supervision de la sécurité doit informer le locataire sur un aspect de la qualité du service qui est pertinent pour le locataire. Le SLO doit définir comment mesurer et calculer la métrique attendue et la métrique doit être calculable par toutes les parties prenantes du SLA.
— Lorsqu'ils fournissent un service de supervision de la sécurité, les fournisseurs de services en nuage doivent connaître sa performance avant d'en faire la publicité auprès des locataires potentiels. Cela signifie que les fournisseurs doivent tester leur capacité de supervision au préalable. Cependant, puisqu'il peut y avoir des dizaines de milliers de failles de sécurité et que chaque locataire s'intéresse à des sous-ensembles différents (peut-être très petits) de failles, les performances doivent être estimées pour un très grand nombre de configurations possibles. Cela nécessite donc une méthode ne demandant qu'une quantité acceptable de tâches de préparation.
— Les SLOs définis dans le contrat de service doivent être vérifiables par tous les participants. La vérification de la satisfaction des objectifs est effectuée après qu'un locataire et un fournisseur ont négocié le SLA et que l'infrastructure a été configurée en conséquence. Pour ce faire, nous devons disposer d'un mécanisme de vérification transparent, c'est-à-dire d'une méthode de vérification qui permette à la fois aux locataires et aux fournisseurs de services d'effectuer cette tâche. En outre, toute partie devrait être en mesure d'effectuer la vérification sans se fier à l'autre partie. Étant donné que la vérification permet de réduire le degré de confiance nécessaire à l'égard de l'autre partie, le processus de vérification doit exiger le moins de confiance possible vis à vis de l'autre partie.

3. Contributions

Dans cette partie, nous présentons nos contributions pour atteindre les objectifs décrits dans la partie précédente. Pour résoudre le problème de l'inclusion de termes de supervision de la sécurité dans les SLAs, nous avons proposé une solution fondée sur le cycle de vie des SLAs. Nous avons conçu et mis en œuvre les tâches requises pour les phases de définition des SLAs et de vérification des SLOs. Nous présentons également un mécanisme pour réduire la dépendance entre les locataires et les fournisseurs en utilisant un registre sécurisé, fiable et distribué. Nos principales contributions sont les suivantes :

— Nous avons conçu des extensions à un langage de SLA existant appelé Cloud SLA (CSLA). Notre extension, appelée Extended CSLA (ECSLA), permet aux locataires de décrire leurs besoins en matière de supervision de la sécurité en termes de vulnérabilités. Plus précisément, un service de supervision de la sécurité est décrit comme une relation entre les besoins des utilisateurs en tant que vulnérabilités, un produit logiciel présentant ces vulnérabilités et une infrastructure dans laquelle le logiciel est exécuté.
— Nous proposons une méthode efficace pour estimer les performances, qui réduit le nombre d'évaluations nécessaires par rapport au nombre de configurations possibles. Pour offrir des SLAs de supervision de la sécurité, les fournisseurs doivent mesurer la performance de leur dispositif de supervision de la sécurité avant de commencer à négocier avec les locataires. En présence de dizaines de milliers de vulnérabilités et d'une variété de locataires, les fournisseurs de services doivent effectuer un grand nombre d'évaluations

pour estimer la performance. Nous proposons une solution afin d'aider les fournisseurs de services dans la préparation des modèles de SLA. La solution proposée introduit deux nouvelles idées. Tout d'abord, nous concevons une méthode de construction d'une base de connaissances qui repose sur des regroupements de vulnérabilités à partir d'heuristiques. Deuxièmement, nous proposons un modèle pour quantifier l'interférence entre des règles de détection associées à des vulnérabilités différentes. En utilisant ces deux méthodes, nous pouvons estimer la performance d'un dispositif de supervision avec peu d'évaluations par rapport à une approche naïve.

— Les métriques utilisées dans nos SLOs tiennent compte de l'environnement opérationnel des dispositifs de supervision de la sécurité. Pour ce faire, il faut concilier deux faits. Premièrement, les paramètres représentant les environnements d'exploitation ne sont pas déterministes (c.-à-d. qu'ils ne peuvent être mesurés avant que l'événement prévu se produise). Par exemple, si une métrique prend le taux d'occurrence des attaques (taux de base) dans sa formule, nous ne pouvons pas estimer sa valeur avant l'occurrence d'une attaque réelle. D'autre part, au moment de la définition d'un SLA, il est nécessaire de spécifier la performance d'un dispositif de supervision de sécurité. Pour concilier ces deux questions, nous proposons un mécanisme d'estimation où la performance d'un dispositif de supervision est mesurée à l'aide de paramètres connus et le résultat est utilisé pour modéliser sa performance et l'estimer pour des valeurs inconnues de ce paramètre. Une définition de SLO contient le modèle qui peut ensuite être utilisé chaque fois que la mesure est effectuée.

— Nous proposons une méthode d'évaluation in situ de la configuration de la supervision de sécurité. Elle permet d'évaluer la performance d'une configuration de l'infrastructure de supervision de sécurité dans un environnement de production. La méthode utilise une technique d'injection d'attaques mais les attaques injectées n'affectent pas les machines virtuelles de production. La méthode peut être utilisée par l'une ou l'autre des parties. Elle permet également de calculer la métrique requise. Cependant, la méthode exige une coopération entre les locataires et les fournisseurs de service.

— Afin d'avoir un service de supervision de sécurité de bout en bout, les locataires ont besoin d'informations telles que la sortie des dispositifs de supervision et les fournisseurs ont besoin de connaissances sur les services qui fonctionnent dans l'environnement des locataires. Par conséquent, il est nécessaire que les deux parties coopèrent. Afin de résoudre la dépendance entre les locataires et les fournisseurs de service lors de la vérification, nous proposons d'utiliser un composant logique sécurisé. L'utilisation proposée d'un composant logique sécurisé pour la vérification est illustrée dans un SLA portant sur l'intégrité des données dans les nuages informatiques. La méthode utilise un registre sécurisé, fiable et distribué (chaîne de blocs ou *blockchain*) pour stocker les preuves de l'intégrité des données. La méthode permet de vérifier l'intégrité des données sans se fier à l'autre partie. S'il y a un conflit entre un locataire et le fournisseur, la preuve peut être utilisée pour résoudre ce conflit.

4. Perspectives

Dans cette partie, nous présentons les perspectives en trois paragraphes. Le paragraphe 4.1 décrit les travaux futurs qui peuvent être réalisés dans un court laps de temps et qui sont axés sur l'amélioration des performances, de la conception et de la mise en œuvre. Le paragraphe 4.2 décrit les objectifs qui peuvent être atteints à moyen terme et se concentre principalement sur les tâches restantes dans le cycle de vie des SLAs comme la configuration de l'infrastructure de supervision de la sécurité pour atteindre les SLOs décrits dans les SLAs. Enfin, le paragraphe 4.3 présente notre point de vue sur une future supervision de la sécurité centrée sur l'utilisateur et entièrement automatisée dans les nuages informatiques.

4.1. **Perspectives à court terme.** A court terme, nous nous concentrons sur l'amélioration des performances, de la conception et de la mise en œuvre. Dans notre définition des SLAs, les locataires sont tenus de décrire leurs besoins en termes de vulnérabilités logicielles. Ce n'est pas une solution optimale car les locataires n'ont généralement pas d'informations détaillées sur les vulnérabilités de leurs logiciels. Afin de résoudre ce problème, une méthode d'analyse de vulnérabilité peut être intégrée à notre méthode de définition des SLAs.

Dans notre travail, nous avons envisagé une sonde de supervision de sécurité spécifique, c.-à-d. une sonde de détection d'intrusion réseau fondée sur des signatures. Les NIDS fondés sur la détection d'anomalies pourraient être traités en étendant la méthode de définition des SLAs. Plus précisément, les paramètres utilisés pour décrire la performance devraient être adaptés au type d'IDS utilisé. Nous considérons également que seules les *vulnérabilités connues* sont garanties dans les SLAs. L'inclusion de vulnérabilités inconnues nécessite d'étendre la définition de SLAs aux IDS fondés sur les anomalies et d'étendre la section de définition des services de l'extension ECSLA.

Dans notre travail, nous avons introduit le concept théorique d'interférence entre règles (*vecteurs* et *matrice d'interférence*). Nous n'avons pas présenté d'évaluation pratique de mesure de l'interférence entre règles. En outre, lors de l'introduction de la notion d'interférence, nous avons supposé que le NIDS lève une alerte pour chaque règle correspondant au paquet réseau analysé. Nous n'avons pas considéré d'autres cas d'interférence

(par exemple, des NIDS ne levant des alertes que pour quelques unes des règles correspondantes, selon certaines heuristiques). Avoir une méthode considérant tous les cas possibles rendrait la formule plus cohérente.

Pour l'intégrité des données dans les SLAs, nous avons présenté une méthode de supervision sans décrire de méthode de définition de SLA. ECSLA peut être étendu et la définition de SLAs pour l'intégrité des données pourrait être effectuée selon une méthode similaire à celle utilisée pour la définition de SLAs pour les NIDS. Pour ce faire, il faut étudier les métriques de performance et les mécanismes de description des exigences des utilisateurs en matière d'intégrité des données.

4.2. **Perspectives à moyen terme.** Dans ce paragraphe nous considérons la phase restante de notre cycle de vie des SLAs, c'est-à-dire la configuration de l'infrastructure de supervision pour la garantie des SLOs, en tenant compte du nombre variable de vulnérabilités à surveiller et des limites de la méthode proposée pour l'évaluation de performance.

Nous avons utilisé un mécanisme simple pour la configuration de sondes de détection d'intrusion réseau selon les SLOs du SLA. D'autres cas pour d'autres exigences pourraient faire l'objet d'études plus poussées. Dans notre processus de définition et de vérification des SLAs, nous avons considéré un nombre fixe de vulnérabilités, c'est-à-dire que le nombre de vulnérabilités à surveiller est fixé au moment de la définition des SLAs. Toutefois, dans la pratique, une vulnérabilité peut être obsolète, par exemple lorsqu'un correctif est rendu disponible pour la corriger, ou bien une nouvelle vulnérabilité pertinente pour le locataire peut être découverte. Dans le futur, un mécanisme pour ajuster automatiquement l'ensemble des vulnérabilités à considérer pourrait être utilisé.

Notre processus de vérification de SLO est difficile à utiliser lorsqu'il s'agit de réaliser des évaluations pour de très petites valeurs du taux de base. En effet, il faut des ressources importantes pour mesurer les performances avec des taux de base très bas. Pour obtenir des résultats statistiquement convaincants, l'injection d'attaques doit se dérouler sur plusieurs cycles. Lorsque le taux de base est très bas, il faut beaucoup de temps pour mener l'expérience pendant plusieurs cycles. De plus, le stockage des paquets transmis entre un attaquant et une machine cible prend beaucoup d'espace disque. Pour réduire la durée, il est possible d'augmenter le nombre de processus qui fonctionnent en parallèle, mais seulement jusqu'à ce que la bande passante réseau maximale soit atteinte. Si nous saturons la ressource réseau disponible, cela peut créer de la congestion et d'éventuelles pertes de paquets à la fois dans le réseau et dans le NIDS. Pour y remédier, il faut modifier notre méthode de vérification. Par exemple, l'exécution à la volée du processus de calcul des métriques, pendant l'injection des attaques, pourrait aider à résoudre le problème de l'espace disque.

4.3. **Perspectives à long terme.** À long terme, la supervision de la sécurité centrée sur l'utilisateur devrait être assurée de manière automatisée. En particulier, la supervision de la sécurité fondée sur les SLAs devrait permettre de décrire les besoins des utilisateurs de manière simple et intuitive. Pour ce faire, il faut combler le fossé sémantique entre le langage intuitif des utilisateurs et les langages compréhensibles par les machines.

Les locataires devraient être en mesure de décrire leurs besoins de la manière la plus naturelle possible et, au lieu de ne s'appliquer qu'à une sonde de supervision spécifique, les exigences devraient s'appliquer à l'ensemble du système de supervision de sécurité. Pour réaliser ce type de configuration de l'infrastructure de supervision de la sécurité, cela nécessite une méthode adaptative selon laquelle une exigence peut être interprétée pour différents types de dispositifs et une configuration appropriée peut être générée pour chaque dispositif. La configuration et la vérification devraient se faire pour chaque dispositif du système de supervision, avec un équilibre entre les dispositifs. Un dispositif peut lever une alerte si un dispositif voisin se comporte de manière inattendue. La vérification devrait également être automatisée et ses résultats devraient être transmis en temps réel.

La supervision de la sécurité automatisée devrait être combinée à des mécanismes de protection afin de réagir automatiquement aux intrusions et de protéger les biens des locataires. Le mécanisme de supervision devrait tenir compte des nouvelles vulnérabilités et ajuster dynamiquement les SLOs. Toute violation doit être traitée automatiquement, c'est-à-dire, si possible, rétablir la configuration à un état satisfaisant les SLOs, sinon appliquer automatiquement des pénalités ou bien signaler l'incident et interrompre le processus de supervision. En résumé, une description simple des exigences, une méthode de configuration et de vérification automatisée et indépendante du dispositif faciliteraient un processus efficace de supervision de la sécurité centré sur l'utilisateur.