Attack tolerance for services-based applications in the Cloud

Georges Ouffoué Ouffoué

▶ To cite this version:

Georges Ouffoué Ouffoué. Attack tolerance for services-based applications in the Cloud. Web. Université Paris Saclay (COmUE), 2018. English. NNT : 2018SACLS562 . tel-02422395

HAL Id: tel-02422395
https://theses.hal.science/tel-02422395v1

Submitted on 22 Dec 2019

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.

université
PARIS-SACLAY

UNIVERSITÉ PARIS SUD
Comprendre le monde, construire l'avenir

NNT : 2018SACLS562

Attack Tolerance for Services-based Applications in the Cloud

Thèse de doctorat de l'Université Paris-Saclay
préparée à Université Paris-Sud

Ecole doctorale n°580 Sciences et Technologies de l'Information et de la Communication (STIC)

Spécialité de doctorat : Informatique

Thèse présentée et soutenue à Gif-sur-Yvette, le 21 Décembre 2018, par

GEORGES OUFFOUÉ

Composition du Jury :

Joaquin Garcia-Alfaro Professeur, Télécom SudParis (SAMOVAR)	Président du jury
Frédéric Cuppens Professeur des Universités, IMT Atlantique (SERES)	Rapporteur
Pascal Poizat Professeur des Universités, Université Paris-Nanterre (LIP6)	Rapporteur
Manuel Núñez Professeur, Universidad Complutense de Madrid	Examinateur
Edgardo Montes De Oca Directeur R&D, Montimage Paris	Examinateur
Fatiha Zaïdi Maître de Conférences HDR, Université Paris-Sud (LRI)	Directrice de thèse
Ana R. Cavalli Professeur Émérite, Télécom SudParis (SAMOVAR)	Co-encadrante de thèse

Thèse de doctorat

Abstract

Web services allow the communication of heterogeneous systems on the Web. These facilities make them particularly suitable for deploying in the cloud. Although research on formalization and verification has improved trust in Web services, issues such as high availability and security are not fully addressed since the solutions proposed are sometimes attack-specific. In addition, Web services deployed in cloud infrastructures inherit their vulnerabilities. For example when different tenants in a cloud platform consume the same instance of the service, attacks such as side-channel can be performed by malicious tenants. Because of this limitation, they may be unable to perform their tasks perfectly. In this thesis, we claim that a good tolerance requires attack detection and continuous monitoring on the one hand; and reliable reaction mechanisms on the other hand. We therefore proposed a new formal monitoring methodology that takes into account the risks that our services may face. To implement this methodology, we first developed an approach of attack tolerance that leverages model-level diversity. We define a model of the system and derive more robust functionally equivalent variants that can replace the first one in case of attack. To avoid manually deriving the variants and to increase the level of diversity, we proposed a second complementary approach. The latter still consists in having different variants of our services; but unlike the first, we have a single model and the implementations differ at the language, source code and binaries levels. Moreover, to ensure detection of insider attacks, we investigated a new detection and reaction mechanism based on software reflection. While the program is running, we analyze the methods to detect malicious executions. When these malicious activities are detected, using reflection again, new efficient implementations are generated as a countermeasure. Finally, we leveraged a formal Web service testing framework by incorporating these complementary mechanisms in order to take advantage of the benefits of each of them.

Résumé

Les services Web permettent la communication de systèmes hétérogènes sur le Web. Ces facilités font que ces services sont particulièrement adaptés au déploiement dans le cloud. Les efforts de formalisation et de vérification permettent d'améliorer la confiance dans les services Web, néanmoins des problèmes tels que la haute disponibilité et la sécurité ne sont pas entièrement pris en compte. Par ailleurs, les services Web déployés dans une infrastructure cloud héritent des vulnérabilités de cette dernière. En raison de cette limitation, ils peuvent être incapables d'exécuter parfaitement leurs tâches. Dans cette thèse, nous pensons qu'une bonne tolérance nécessite un monitoring constant et des mécanismes de réaction fiables. Nous avons donc proposé une nouvelle méthodologie de monitoring tenant compte des risques auxquels peuvent être confrontés nos services. Pour mettre en oeuvre cette méthodologie, nous avons d'abord développé une méthode de tolérance aux attaques qui s'appuie sur la diversification au niveau modèle. On définit un modèle du système puis on dérive des variantes fonctionnellement équivalents qui remplaceront ce dernier en cas d'attaque. Pour ne pas dériver manuellement les variants et pour augmenter le niveau de diversification nous avons proposé une deuxième méthode complémentaire. Cette dernière consiste toujours à avoir des variants de nos services; mais contrairement à la première méthode, nous proposons un modèle unique avec des implantations différentes tant au niveau des interfaces, du langage qu'au niveau des exécutables. Par ailleurs, pour détecter les attaques internes, nous avons proposé un mécanisme de détection et de réaction basé sur la réflexivité. Lorsque le programme tourne, nous l'analysons pour détecter les exécutions malveillantes. Comme contremesure, on génère de nouvelles implantations en utilisant toujours la réflexivité. Pour finir, nous avons étendu notre environnement formel et outillé de services Web en y incorporant de manière cohérente tous ces mécanismes. L'idée est de pouvoir combiner ces différentes méthodes afin de tirer profit des avantages de chacune d'elle.

To my father Koffi Ouffoué.

Acknowledgements

I would like first to thank YESHOUA HA MASHIAH without whom this thesis would not have been possible. I would also like to convey my sincere thanks to the members of the jury for their presence and in particular the reviewers Prof. Cuppens and Prof. Poizat who kindly evaluated my manuscript. My thanks also go to my thesis advisors Mrs. Fatiha Zaïdi and Mrs. Ana R. Cavalli for the opportunity they offered me to do my thesis under their leadership. It was both a very good scientific and human experience. Their expertise and directions have been invaluable to help me in this thesis. I would like to thank Prof. Bernard Cousin and Prof. Oumtanaga who set up the partnership that allowed me to come and pursuing my studies in France. I thank also my parents, my beloved Elodie, my friends for their unwavering supports. I thank the VALS team of the LRI and the employees of Montimage Paris and Spain for their warm welcome and integration. I would like to thank in particular Dr. Huu Nghia Nguyen from Montimage, Dr. Mounir Lallali from Brest, Dr. Antonio Ortiz and Dr. Cesar Sanchez from Spain for their valuable support and collaboration during this thesis. Last but not least, I would like to thank the members of the group "le coup de la pieuvre" for the conviviality offered during every launch and entertainment parties (baby-foot).

Contents

Abbreviations **i**

1 Introduction **11**
1.1 General Context 12
1.2 Contributions 13
1.2.1 Risk-based monitoring methodology 13
1.2.2 Diversity-based attack tolerance 14
1.2.3 Reflection based attack tolerance 14
1.2.4 An attack tolerance framework for cloud applications 15
1.3 Publications 15
1.3.1 Workshops 16
1.3.2 International Conferences 16
1.3.3 Talks 16
1.3.4 Posters 16
1.4 Outline of the Thesis 16

2 Attack tolerance: Challenges & directions **19**
2.1 Research on Web services 20
2.2 Security issues related to Web services 22
2.2.1 XML DoS 22
2.2.2 Metadata Spoofing 23
2.2.3 SQL Injections 23
2.2.4 Capture and Replay Attacks 23
2.2.5 Session Hijacking 23
2.2.6 WSDL scanning 24
2.2.7 Parameter tampering 24
2.2.8 External reference attack 24
2.3 Cloud computing security issues 24
2.3.1 Cloud computing in a nutshell 24
2.3.2 Cloud Market and challenges 27
2.3.3 Virtualization vulnerabilities 31
2.4 Intrusion and attack tolerance for Web services 32
2.4.1 Attack tolerance techniques 32
2.4.2 Diversity techniques 35
2.4.3 Attack tolerance techniques for Web services 38
2.5 Formal methods 39

2.5.1 Static analysis . . . 39
2.5.2 Dynamic analysis . . . 40
2.6 Discussion . . . 42

3 Risk-based passive monitoring **43**
3.1 Risk-based monitoring methodology . . . 44
3.1.1 Identifying Assets . . . 45
3.1.2 Risk and vulnerability analysis . . . 47
3.1.3 Threats Modelling . . . 47
3.1.4 Attack scenarios . . . 48
3.2 The Montimage Monitoring Tool (MMT) . . . 52
3.2.1 MMT-Security architecture . . . 52
3.2.2 MMT-Security properties . . . 53
3.3 Discussion . . . 54

4 Diversity-based attack tolerance **57**
4.1 Model-based diversity for attack tolerance . . . 58
4.1.1 Overview . . . 58
4.1.2 Authentication example . . . 59
4.1.3 Experimentations . . . 62
4.1.4 Discussion . . . 68
4.2 Implementation-based diversity for attack tolerance . . . 68
4.2.1 Definition of key concepts . . . 69
4.2.2 Overview of the approach . . . 69
4.2.3 Experiments and discussion . . . 75
4.3 Discussion . . . 79

5 Software reflection based attack tolerance **81**
5.1 Background . . . 82
5.2 Framework . . . 83
5.3 Case studies . . . 88
5.3.1 Overview . . . 88
5.3.2 Detection and mitigation . . . 94
5.4 Experiments and results . . . 97
5.5 Discussion . . . 99

6 An attack tolerance framework for Web-based applications in the cloud. **101**
6.1 Web services and cloud applications . . . 102
6.2 SChorA . . . 105
6.2.1 A symbolic model and an integrated environment for specifying and analyzing service choreographies. . . . 105
6.2.2 Passive testing . . . 106
6.3 Attack tolerance in the cloud . . . 109

6.3.1 Part 1: Verification and code generation 109
6.3.2 Part 2: Deployment, monitoring and reaction 113
6.4 Discussion 114

7 Conclusion 117
7.1 Synthesis of results 117
7.2 Perspectives 119

A Appendix 121
A.1 Example: Vote application 121
A.1.1 Verification and code generation 121
A.1.2 Testing 122
A.2 Résumé de la thèse en Français 125
A.2.1 Contexte 125
A.2.2 Contributions 126
A.2.3 Publications 129
A.2.4 Posters 130

Index 143

List of Figures

2.1 Panorama of Web services research contributions. We must understand that attack tolerance spans these domains. 22
2.2 Cloud services Models 25
2.3 A typical cloud setup 27
2.4 Cloud market shares from the Synergy Research Group 28
2.5 Overview of a side channel attack on a cloud platform. 30

3.1 Risk-based monitoring loop 45
3.2 CLARUS proxy architecture 46
3.3 Attack tree for unauthorized users attack 49
3.4 Montimage Monitoring Tool overview 52

4.1 Attack tolerance Framework 58
4.2 Authentication Model 1 59
4.3 Authentication Model 2 60
4.4 Authentication Model 3 61
4.5 Authentication Model 4 61
4.6 Experimentation Framework 62
4.7 IF specification of Model 1 64
4.8 Test objectives 65
4.9 Generated test cases 65
4.10 The authentication showing the adaptation GUI 67
4.11 Architecture of the attack tolerance Framework 69
4.12 Feature Model of the e-health service 70
4.13 WSDL Sample 71
4.14 XSD Sample 72
4.15 Server side skeleton 72
4.16 Attack tolerance through diversity 73
4.17 Latency with and without our framework 77
4.18 Latency with and without our framework 78

5.1 Attack tolerance framework 85
5.2 Detection and Reaction model 86
5.3 API of the `HealthOperation` Center 88
5.4 Correct implementation of *updatePatient*. 88
5.5 Unexpected implementation of *updatePatient* function. 89
5.6 Setup of the running system. 90

5.7 Implementation of the *checkSource* method. 90
5.8 Normal case 91
5.9 Inconsistent messages diagrams. 93
5.10 Inconsistent messages diagrams. 93
5.11 Security Rule representation in MMT. 95
5.12 Example of usage of metaclasses 96
5.13 RESTful API of the e-health Center. 97
5.14 Experiments 1 & 2 results 98

6.1 Chor choreography language [Qiu et al., 2007] 106
6.2 SChorAcloud architecture and components. 109
6.3 Example of definition of a choreography with 2 roles 111

A.1 Verification of the Vote choreography 122
A.2 Conformance Checking 123
A.3 Generated skeletons 124
A.4 Security rule in MMT: The hashes of the called vote method should be equal to the hash existing in the database before that call 125

List of Tables

2.1 Mapping between attack tolerance mechanisms and architectures 34
2.2 Diversity techniques 38

3.1 DoS/DDoS attack detection 51

4.1 Scenario 2 measurements 68
4.2 Threshold measurement 76
4.3 T_D measurements 77
4.4 Time elapsed to detect dos attacks 79
4.5 Time elapsed to complete a client request in the presence of a DDoS attack 79

5.1 Taxonomy of malware 84

List of acronyms

API Application Programming Interface

DoS(DDoS) Denial Of Service (Distributed)

FM Feature Model

FSM Finite State Machine

IT Infrastructure Technology

IUT Implementation Under Test

LTL Linear Temporal Logic

MMT Montimage Monitoring Tool

NOP No Operation

REST REpresentational State Transfer

SChorA Symbolic Choreography Analysis

SOA Service Oriented Architecture

SOAP Simple Object Access Protocol

STG Symbolic Transition Graph

SUO System Under Observation

SUT System Under Test

SVM Support Vector Machine

VM Virtual Machine

W3C World Wide Web Consortium

WSDL Web Services Description Language

XML Extensible Markup Language

XSD XML Schema Definition

1 Introduction

1.1 General Context

Computer systems are now at the heart of all business functions (accounting, customer relations, production etc) and more generally in everyday life. These systems consist of heterogeneous applications and data. They should be described through modular architectures that integrate and compose them in order to meet the needs of the organization. Service Oriented Architectures (SOA) have been proposed for this purpose. According to the Gartner Group [1],

> *Service-oriented architecture (SOA) is a design paradigm and discipline that helps IT meet business demands. Some organizations realize significant benefits using SOA including faster time to market, lower costs, better application consistency and increased agility.*

These architectures are distributed and facilitate communication between environments of heterogeneous nature. The main components of such architectures are Web services. A web service is a collection of open protocols and standards for exchanging data between systems. Thus, software applications written in different programming languages and running on different platforms can therefore use Web services to exchange data. These services can be internal and only concern one organization. On the other hand, with technological advances in communication networks especially the Internet and the expansion of online services via cloud computing or simply the interconnection of IT systems, the need to expose services to the outside world is growing. Cloud computing for example enables sharing of IT resources (computing, storage, networks, etc.) on demand over the Internet. These services are often deployed on the basis of smaller components (containers, virtual machines, etc.) deployed on a single site or on several geographically distributed sites. They can also be provided by several different cloud service providers (multi-cloud applications).

However, Web services since they are open and interoperable, are privileged entry points for attacks. Web services like many others technologies taking advantage of the Internet, are also facing attacks on availability, integrity and confidentiality of platforms and user data. Moreover, Web services deployed in the cloud inherit their vulnerabilities. Recently, new attacks exploiting cloud vulnerabilities (such as side-channel, VM escape, hacked interfaces and APIs, account hijacking, etc.) are considerably reducing the effectiveness of traditional detection and prevention systems (e.g., firewall, intrusion detection systems, etc.) available in the market. Most of the time, these attacks are orchestrated by competitors to sabotage opponents companies in order, first to inflict them heavy financial losses, or secondly to access to confidential information such as intellectual property or private data. Thus, the enterprise today is under constant attacks from criminal hackers and other malicious threats. Facing these increasingly destructive attacks, research on Web services has been focused on modeling, composition and testing. They may not be sufficient to

[1] https://www.gartner.com/it-glossary/service-oriented-architecture-soa

fully ensuring the confidentiality, integrity and availability requirements of Web services. Moreover, they can only detect existing attacks with relatively high rates of false positives and negatives. Because of this limitation, Web services may fail to achieve their goals when attacks are successful. It is not enough just to detect intrusions; Web services need to effectively respond to the attack. Existing solutions limited to one application domain can not maintain an acceptable level of service for users. Continuous availability is then a critical need. Traditional intrusion detection techniques should be improved, or even new approaches that are more suited to these environments should be developed. The goal of our thesis is to answer the question: How to ensure attack or intrusion tolerance of Web-services based applications?

1.2 Contributions

We define attack tolerance or intrusion tolerance as the capability of a system to continue to function properly with minimal degradation of performance, despite intrusions. The aim is to detect the known and unknown attacks and if not possible reduce their impact on the system. We therefore believe that to ensure an effective attack tolerance, we must first detect attacks upstream in order to react effectively downstream. Furthermore, it is also appropriate to take into account functional requirements as well as non-functional requirements, at all levels: design, specification, development, deployment and execution. Functional requirements are requirements that define a function of the system to be developed. In other words, what the system needs to do. Non-functional requirements are requirements that characterize a desired quality or property of the system such as its performance, robustness, adaptability, responsiveness, availability, disaster recovery, and so on. In this thesis, we adopt a new end-to-end security approach based on formal monitoring and diversity. We therefore proposed a new formal monitoring methodology that takes into account the risks that our services may face. To implement this methodology, we first developed approaches of attack tolerance that leverages diversity. The basic idea that we propose is to have variants of the components of the software. Such variants react and replace themselves when one of these components is compromised due to the effects of an attack. More precisely, our contributions are the following:

1.2.1 Risk-based monitoring methodology

We leveraged the traditional risk management loop to build a risk-based monitoring that integrates risks into monitoring (Chapter 3). We claimed that the detection and prevention of attacks require a good knowledge of the risks that these systems face. As such, it is mandatory to include risk management in the monitoring strategy in order to reduce the probability of failure or uncertainty. This methodology involves the following aspects: i) assets identification to define what is necessary to protect. ii) Threats and vulnerability analysis, to evaluate the potential flaws the system may suffer. iii) Risk analysis to categorize the threats that can exploit the system vulnerabilities. iv) System monitoring to detect potential occurrences of attacks, and. v) Remediation strategies to repel or mitigate

the impact of the attacks. This methodology has been applied throughout this thesis. We also briefly presented the Montimage Monitoring Tool (MMT) that helps us to detect the attacks and implement the risk-based approach. The main components, functionality and the mechanisms used to define security properties of the tool have been described.

1.2.2 Diversity-based attack tolerance

Model-based diversity for attack tolerance

We investigated attack-tolerance at the design phase (Section 4.1). We designed a model of the software, system. From that model, we derived equivalent models that have the same purposes as the first one. According to these models, we obtained the corresponding implementations. In case of a detected attack by our monitoring tool, we dynamically change the model, choosing a model and its implementation which is more robust in the presence of the attack. This means that the attacker is confronted to a "new" system for which his attack is not successful. We evaluated this approach by injecting a brute-force attack on an authentication Web application we developed for the experiments. This showed that our approach seemed suitable for brute-force attacks. The limits of this first approach laid on the derivation of the variants of the model. How to automatically synthesize them?

Implementation-based attack tolerance

This contribution aims at extending and solving the issues raised by the former approach (Section 4.2). The idea is still the same like in the previous contribution but here, there are only one model and several implementations. To this end, we based our work on the concept of diversification. It should be noted that diversification refers to having multiple copies for the same concept. We illustrated the approach with a Web service that simplifies the management task in a hospital. We built several variants using diversity at Web services pattern, language, source code and binary levels. Furthermore, we designed in a generic manner attack tolerance to both active and silent attacks. We proposed an active reconfiguration process that mitigates attacks that are not easy or impossible to detect by monitoring (i.e., low bit-rate attacks, silent attacks). The experiments showed that the method proposed seems highly reactive.

1.2.3 Reflection based attack tolerance

In this contribution, our aim was to address attack tolerance in a different manner (chapter 5). In fact in the methods mentioned above, we had the capability to detect attacks coming from the outside(DDoS, Brute-force,..). In addition, their tolerance features were designed before the deployment of the software system (e.g. diversification of Web services). That's why we thought about finding a solution that would tolerate internal attacks. We proposed a new attack tolerance methodology for insider attacks that are known to be difficult to detect because the users are authenticated on the domain. This new contribution integrates Software reflection techniques as well as monitoring based on log files, for an efficient

detection and mitigation of such attacks. Reflection is a technique that helps a program to monitor, analyze and ajust its behavior dynamically. We considered that the software of the client is located in a safe environment. Potential attacks that can take place are internal ones. The goal of the intruder was to usurp the actions, i.e. to modify the methods of the API of the platform. By reflection we obtained the hashes of the source code of any method of the API. Any deviation at runtime of these hash values means the presence of a misbehavior. Such misbehavior could be whether an insider attack or a virus attack. We stored Date, Hour, Operation , hash, host in the file. Any request has then two traces in the logs: outbound(request) and inbound(response). Moreover, using the Montimage Monitoring Tool (MMT) we analyzed the code against a Virus DB and detect any change of the software. We fully implemented a plugin in the MMT tool for the detection of insider attacks based on reflection. We evaluated the e-health Web service using a realistic testbed. We evaluated the detection capability of the proposed framework proposed in comparison with an antivirus software of the market. The experiments show that our attack tolerance is effective. This contribution explored and laid the groundwork for attack tolerance using software reflection.

1.2.4 An attack tolerance framework for cloud applications

To do this, we extended a formal framework for choreography testing and conformance by incorporating the contributions above (Chapter 6). Cloud services are modelled as a choreography of Web services, each service running a unique, specific business function. The global choreography is projected on the different roles *i.e.* different services of this composition. We verified that each implementation conforms to the model of choreography. The different models generated locally are checked for the conformance to the original choreography of the application. From these local models we derive some skeletons in at least three different programming languages. After having these skeletons generated, the developer implement the methods and finalize the source code. After the implementation of the services, one among them is chosen for each role. They are two complementary ways for the monitoring of the applications. The first way is to test the implementations before launching them. The purpose is to detect programming errors or deviation with respect to the requirements. The second way consists in leveraging software reflection. This is to detect other misbehaviors. At runtime when a misbehaviour is detected, we verify if it is a programming error or if an attack occurred. We react by applying related countermeasures using also reflection. Adding mechanisms of detection and reaction on the fly to these applications verified correct by design, ensures optimal attack tolerance. As a result, we will ensure a total attack tolerance of cloud applications from both formal and practical points of view.

1.3 Publications

The main contributions of this thesis have already been published in proceedings of international conferences as well as presented in national research days.

1.3.1 Workshops

1. G. Ouffoué, A. M. Ortiz, A. R. Cavalli, W. Mallouli, J. Domingo-Ferrer, D. Sánchez,and F. Zaïdi. *Intrusion detection and attack tolerance for cloud environments:The clarus approach.* In 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW), pages 61–66. IEEE, 2016.

2. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *Model-based attack tolerance.* In 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pages 68–73, IEEE, 2017.

1.3.2 International Conferences

1. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *How Web Services Can Be Tolerant to Intruders through Diversification.* ICWS 2017 24th IEEE International Conference on Web Services, pages 436 - 443, IEEE 2017.

2. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *An Attack-Tolerant Framework for Web Services.* In 2017 IEEE International Conference on Services Computing (SCC), pages 503-506, IEEE, 2017.

3. Ana R. Cavalli, Antonio M. Ortiz, Georges Ouffoué, Cesar A. Sanchez, and Fatiha Zaïdi. *Design of a Secure Shield for Internet-based Services using Software Reflection.* ICWS 2018 International Conference on Web Services, Lecture Notes in Computer Science, vol 10966. Springer, Cham

1.3.3 Talks

1. *How web services can be tolerant to intruders through diversification?* in Journées du GDR MTV2/MFDL track. Dec 2017.

1.3.4 Posters

1. *Attack Tolerant Cloud* in C&ESAR National conference on Security.

1.4 Outline of the Thesis

The rest of the thesis is divided into six chapters. The state of the art is presented in the next chapter 2 while the main contributions are deeply presented in chapter 3 to 6. More precisely,

1. **Chapter 2** presents in a nutshell, Web services, the security issues and the solutions proposed in the literature, as well as formal methods,

2. **Chapter 3** presents the risk-based monitoring approach as well as MMT, the main monitoring tool we leveraged in this thesis,

3. **Chapter 4** presents model-based diversity and implementation-level diversity, two approaches leveraging diversity to enable attack tolerance,

4. **Chapter 5** presents reflection-based attack tolerance an approach leveraging software reflection to enable attack tolerance,

5. **Chapter 6** presents a complete attack tolerance for cloud applications based on Web services. This methodology leverages the other contributions and extends SChorA, a framework for choreography conformance and testing.

6. **Chapter 7** concludes this thesis. The limitations of the thesis are discussed and some future works are also pointed out.

2

Attack tolerance: Challenges & directions

Contents

1.1 General Context 12
1.2 Contributions 13
1.2.1 Risk-based monitoring methodology 13
1.2.2 Diversity-based attack tolerance 14
1.2.3 Reflection based attack tolerance 14
1.2.4 An attack tolerance framework for cloud applications 15
1.3 Publications 15
1.3.1 Workshops 16
1.3.2 International Conferences 16
1.3.3 Talks 16
1.3.4 Posters 16
1.4 Outline of the Thesis 16

Web services allow the interoperability and communication of heterogeneous systems in the Web through Internet protocols. These facilities make them particularly useful for implementing services oriented architectures (SOAs) of companies, cloud services (e.g., Amazon, Microsoft, Google) and even governments applications. Besides, the efforts and findings of the last decades of research on the formalization and the verification of Web services have given a certain level of assurance on Web services. However new challenges such as high availability and security issues are not fully addressed. Web services are also subject to attacks that are destructive even if they are well known. Moreover, Web services located in cloud platform inherit their vulnerabilities. Very few solutions exist to ensure the availability of Web services in the presence of these attacks. This chapter presents Web services and an overview of the literature and explored the scientific works and techniques that have brought more trust on Web services. The main cloud vulnerabilities are then

disclosed. We also present existing attack tolerance techniques highlighting the main issues that remain unsolved. We explore software formal methods as well, in order to disclose their benefits for attack tolerance.

2.1 Research on Web services

Service Oriented Architecture (SOA) is an approach that defines a pattern for the implementation of functionalities, grouped together in execution units called services, offered by suppliers or producers to customers or consumers. The relationship between the producer and the consumer of a service is described by means of a contract. The contract formalizes, according to standards, the functionalities offered by the service as well as the technical context in which these functionalities must be delivered. However, the contract does not specify how these features are implemented at the producer level. A service is independent of a given technology or programming language. Finally a service is publishable and discoverable: it can be registered by the provider within a directory or register. The consumer can consult the register and, from the information contained in the contract, choose to access the service. The SOA approach aims, on the one hand, to facilitate the reuse of software entities, on the other hand, to improve the interoperability between a customer and a supplier by reducing coupling. According to the World Wide Web Consortium (W3C), a Web service is a software system designed to support interoperable machine-to-machine interaction over a network. In other words, Web services enable the communication and exchange of data between heterogeneous applications in distributed systems. The idea is to solve interoperability and complexity issues in the Web.

Web services have increasingly gained a great popularity. Almost all software companies, governments or even individuals applications are deployed using Web services. This success story is due in large part to cloud computing. Indeed, in cloud computing with scalable previsions, abstracted IT infrastructures, platforms and applications with a pay-per-use model, everything is delivered as a service (XaaS). We will present in detail cloud computing in next sections. With the heterogeneity of standards, Web services must be safe and reliable, i.e., they must be able to satisfy the requirements and constraints of users. Consequently, over the last decades, several research have been carried out with the aim of ensuring the suitable interactions between Web services and ensuring safety of Web services.

[Hwang et al., 2009] [Morales et al., 2010] [Nguyen et al., 2013] [Cao et al., 2010] contributed to the verification of Web services. These verification techniques often use formal methods. The main advantage of formal modeling and analysis, is the establishment of a high level of confidence on the software. In addition, as Web services became more and more complex, architectures of Web services which were very often monolithic evolved to give more modular architectures. A composite Web service is defined as a Web service with several components that can be developed separately and deployed in distributed environments. Running tests on such systems requires more effort due to their size and distributed environments. The conventional verification techniques have been

proved insufficient and inadequate. The works [Nguyen et al., 2014] [Cavalli et al., 2010] have therefore provided a formal framework for testing such systems. Moreover, better management of these composite services is made possible through efficient techniques such as service choreography. Service choreography defines requirements from a global point of view, based on the interactions between a set of participants that are implemented as services.

[Nguyen et al., 2016] contributed to the development of reliable data service choreography using a dedicated projection algorithm. It should also be noted that the implementation of Web services can lead to significant design and development delays. Reuse and composition are therefore necessary to reduce the delays and time to market. However, reused components or entities may not have the same interfaces and may not meet the same business requirements. A lot of research has helped to investigate and promote software adaptation. [Mateescu et al., 2012] proposed new techniques based on process algebra. This allowed them to define innovative adapters obtained from the service description languages. Several other methods based on model-driven engineering techniques have emerged in order to address the problem of coping with dynamic service changes [Morin et al., 2009].

Moreover, Web services are mainly involved to ensure the quality of Web services in order to take into account the requirements of the customers. In addition, given the number of online services that are functionally equivalent, there was a need for relevant criteria for helping users to choose the best service that would meet their needs perfectly. The metrics of quality of service (QoS) and quality of experience (QoE) have therefore been proposed. The QoS is usually defined as a set of attributes or parameters, such as the response time, availability or reliability corresponding to a given Web service. [Kondratyeva et al., 2013] reviewed the works published on QoS evaluation for Web service and proposed a model based evaluation of QoS to evaluate the quality of Web services. They used weighted Finite State Machine (FSM). The weight associated with each transition represents the cost of the corresponding transaction execution (in terms of time for example). The quality parameter values were estimated via different execution paths of the corresponding FSM.

All these contributions, (cf. Figure 2.1) even though have significantly improve the state of art, are not enough to protect Web services against malicious and destructive attacks.

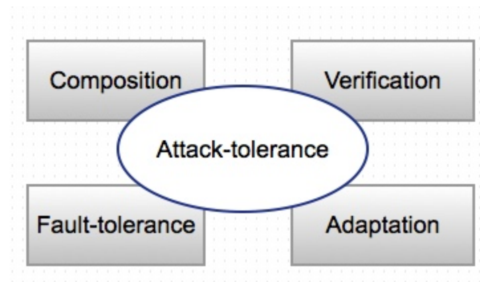

Figure 2.1 – Panorama of Web services research contributions. We must understand that attack tolerance spans these domains.

In fact, the popularity of Web services and the fact, that these services are accessible from the Internet, make them potential targets for malicious individuals and organizations. Internet applications and specifically these based on cloud-based Web services are under constant attack from criminal hackers and other malicious threats They generally perpetrate malicious attacks for stealing confidential data or altering availability of services. These attacks affect all actors of the society. For companies, this is a significant loss of turnover. Imagine an airport company whose reservation application is out of service for even one hour, the losses could be in millions of dollars. In the same way, the main threat for governments remains the destruction of critical and strategic military positions. In the next sections we will present the security issues of Web services as well as cloud computing-based vulnerabilities.

2.2 Security issues related to Web services

Web services face several attacks. The main attacks are the following [Kuyoro et al., 2012].

2.2.1 XML DoS

A XML DoS(X-DoS) attack can be defined as an explicit attempt by attackers to prevent legitimate users of a service from using that service. X-DoS can be manifested [Ficco and Rak, 2011]:

1. By flooding the network with XML messages thereby not allowing legitimate network traffic

2. By flooding the service with XML requests thereby affecting availability of the service. Oversize Payload: It is a resource exhaustion attack, which is performed by querying a service using a very large request message. It exploits the high memory consumption of XML processing implemented by most Web Service frameworks that use a tree-based XML processing model. A possible countermeasure against such an attack consists in the restriction of the total buffer size for incoming Simple Object Access Protocol (SOAP) messages.

3. By passing malicious XML content in a request thereby disrupting the service and making it unavailable for other legitimate users. We can cite Coercive Parsing also called Deeply-Nested XML. This attack exploits the SOAP message format by means of inserting of a large number of nested XML tags in the message body. The goal is to force the XML parser within the server application to exhaust the computational resources of the host system by processing numerous deeply-nested tags. A countermeasure could consist in limit the number of namespace declarations per XML elements. On the other hand, XML specification does neither limit the number of nested XML elements, nor the length of the namespace URIs. Any restriction could lead to unpredictable rejection of legitimate messages.

2.2.2 Metadata Spoofing

Metadata is a specific information (creation date, modification date, author name. . .) that characterizes an electronic component (file, data, picture. . .) and that is useful to its management. It can be found in emails, social media, websites, and even in phone calls. A client retrieves all the information regarding a web service invocation (i.e., message format, network location, security requirements, etc.) from the metadata documents provided by the web service server. This metadata is usually distributed using communication protocols like HTTP/S or mail. These circumstances open new attack possibilities aiming at spoofing these metadata. In this attack, the attackers try to modify the information in server's metadata so that user can be redirected to different place.

2.2.3 SQL Injections

When SQL statements are generated, security vulnerabilities can occur. SQL injections can be generated by inserting characters into SOAP requests, web forms, or URL parameters. The attacker can access privileged data, or connect to protected areas and compromise the Web service.

2.2.4 Capture and Replay Attacks

Such an attack can occur when a malicious party intercepts communications between two peers. An attacker may inserts, removes or modifies information within messages transmitted between peers. If he can capture messages and SOAP requests, he can replay them by adding his own information. This can cause malfunction for the actual users of the service.

2.2.5 Session Hijacking

It occurs when an attacker steals a user's valid session ID and uses it to gain that user's privileges in the Web service. For example, an attacker who by intercepting SOAP messages can hijack a user's session in the same way as with typical web application attacks.

2.2.6 WSDL scanning

This attack consists in trying to retrieve the WSDL of Web services in order to obtain precious information [Singhal et al., 2007].

2.2.7 Parameter tampering

This type of attack consists in a modification of the parameters expected by a service to bypass input validation and gain unauthorized access to some functionality [Singhal et al., 2007].

2.2.8 External reference attack

It is an attempt to bypass protections by including external references that will be downloaded after the XML has been validated but before its processed by the application [Singhal et al., 2007].

In conclusion, Web services are the target of Cyber attacks. Moreover Web services are increasingly used to develop Enterprise Service Oriented Architectures. These services are often deployed in the cloud. Indeed [Sharma et al., 2011], have shown the interest of deploying web services in the cloud. They pointed out that deploying Web services in the cloud increases the availability and reliability of these services and reduces the messaging overhead. In fact, the resources, provided per demand in the cloud with great elasticity, satisfy the requirements of the service consumers. As conclusion, Web services deployed in the cloud or used for building cloud applications inherit the vulnerabilities of the cloud platforms. This is why in the next section we will present cloud computing as well as the security issues related.

2.3 Cloud computing security issues

2.3.1 Cloud computing in a nutshell

The expansion of the Internet has allowed the emergence of cloud computing. The term cloud computing is not as recent as that. Indeed, according to some, in 1960, John McCarthy (1927-2011), one of the pioneers of artificial intelligence McCarthy had already planned that computing facilities would be provided to the general public as a public service.

Essential characteristics

According to the NIST [Mell and Grance, 2001], essential characteristics of cloud can be classified as follows:

- **On-demand self-service** : Different computing capabilities, storage services, software services etc. should be accessed as needed by consumers automatically without

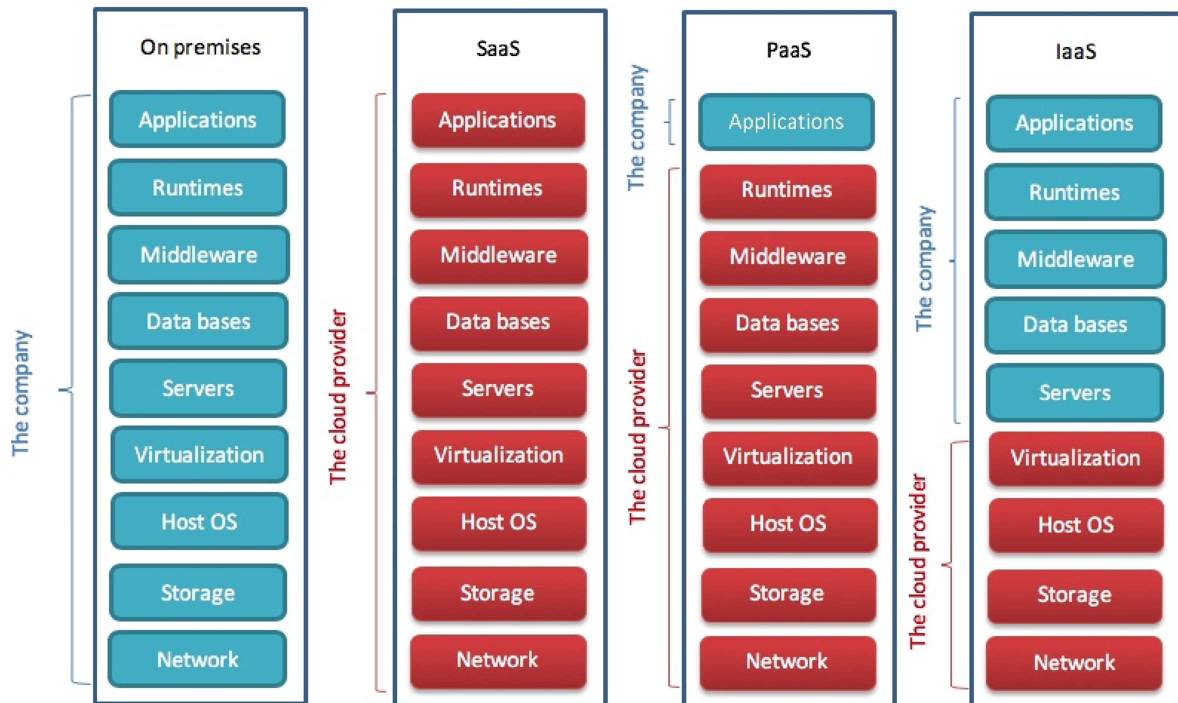

Figure 2.2 – Cloud services Models

service provider's intervention. The resources are pooled to serve the users at a single physical location and/or at different physical location according to the optimality conditions (e.g. security, performance, consumer demand).

- **Pricing** : No capital expenses are needed. Users may pay and use or pay for services and capacity as they need.
- **Broad network access** : All services are available over the network and are also accessible through standard protocols using web enabled devices computers, laptops, mobile phones etc.
- **User-Centric Interface**: Cloud interfaces are location independent and they can be accessed by well established interfaces such as Web services and Web browsers.
- **Autonomous System** : Cloud computing systems are autonomous systems managed transparently to users.
- **Rapid elasticity**: Elasticity is a key functionality of cloud computing. The resources appear to users as indefinite and are also accessible in any quantity at any time.

Software and data inside clouds can be automatically reconfigured and consolidated to a simple platform depending on user's needs

- **Measured service** : A metering capability is deployed in cloud system in order to charge users. The users can achieve the different quality of services at different charges in order to optimized resources at different level of abstraction suitable to the services (e.g. SaaS, PaaS and IaaS).
- **Quality of Service (QoS)** : Cloud computing can guarantee QoS for users in terms of hardware or CPU performance, bandwidth, and memory capacity.

Three service models

Three service models can be offered on the cloud(Figure 2.2):

- **Software as a service (SaaS):** This service model is characterized by the use of a shared application that runs on a cloud infrastructure. The user accesses the application via the network through various types of terminals (e.g a web browser). The application administrator does not manage and control the underlying infrastructure (networks, servers, applications, storage). He does control only the functions of the application.
- **Platform as a Service (PaaS) :** The user has the ability to create and deploy its own applications on a cloud PaaS infrastructure using the vendor's languages and tools. The user does not manage or control the underlying cloud infrastructure (networks, servers, storage) but controls the deployed application and its configuration.
- **Infrastructure as a Service (IaaS):** The user rents computing and storage resources, network capabilities and other necessary resources (load sharing, firewall, cache). The user has the opportunity to deploy any type of software including operating systems. The user does not manage or control the underlying cloud infrastructure (Host OS) but has control over operating systems, storage, and applications.

Three service deployments

Three deployments can be offered:

- **Private cloud:**

  The cloud infrastructure is used by a single organization. It can be managed by the organization or by a third party. The infrastructure can be placed on the premises of the organization or outside.
- **Community cloud:** The cloud infrastructure is shared by several organizations for the needs of a community that wants to pool resources (security, compliance, etc.). It can be managed by organizations or by a third party and can be placed on premises or outside. Examples of community clouds are Google Apps for Government and Microsoft Government Community Cloud.

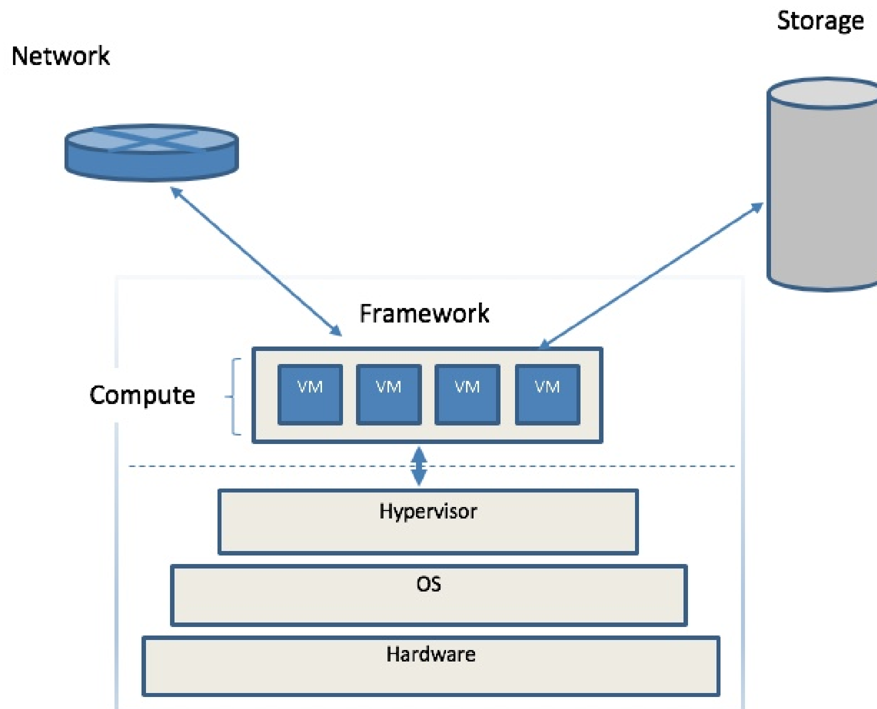

Figure 2.3 – A typical cloud setup

- **Public cloud :** The cloud infrastructure is open to the public or to large industrial groups. This infrastructure is owned by an organization that sells cloud services. Amazon with his platform Amazon Web Services (AWS) [1] is one of the leader in the public cloud market.

- **Hybrid cloud:** The cloud infrastructure consists of one or more of the above models that remain separate entities. These infrastructures are interconnected by the same technology that allows the portability of applications and data.

2.3.2 Cloud Market and challenges

The cloud with its promises and benefits has generated a lot of enthusiasm for companies of all types and sizes. Internet giants have seen a way to offer new services while secular companies have seen a way to reduce their operational costs. Cloud infrastructures are particularly suitable for small companies and start-ups as, in the beginning of their activities, do not necessarily have huge financial resources for the acquisition of qualified materials.

[1] https://aws.amazon.com/fr/

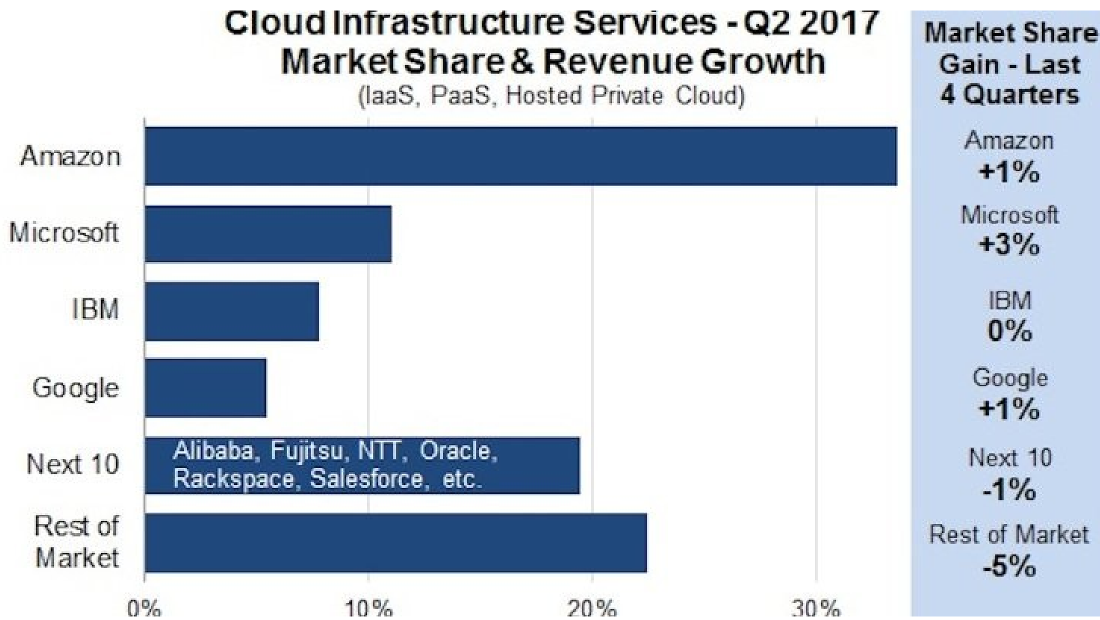

Figure 2.4 – Cloud market shares from the Synergy Research Group

As we can see on Figure 2.4 [2], the Internet giant share the market with some new, smaller competitors. The leading cloud providers continue grow their share of the worldwide cloud infrastructure services market. Amazon with Amazon Web Services (AWS) continues to lead the market with 34% followed by Microsoft 11%, IBM 8 %, and Google 5%. Then there are next 10 with less than 20 % combined market share. Despite the fact that cloud adoption is accelerating considerably year-on-year, there are still a number of practical or research challenges.

A number of challenges which are currently addressed by researchers are identified ([Prasad et al., 2013], [Dillon et al., 2010], [Ahamed et al., 2013]). Although all these issues are challenging, security and privacy remain a barrier to the complete adoption of the cloud. In this document we will then focus on the challenge of security. We've listed the top attacks targeting cloud applications and infrastructures.

Cloud based threats

Cloud computing infrastructures are complex and share many of the same vulnerabilities as legacy IT systems connected to the Internet (e.g., DoS attacks), while offering their own unique challenges. Most of cloud consumers use cloud services owned by Cloud Service Providers (CSP) for storage capabilities. In fact, they have no control and visibility of their own data. According to the IEEE, control is the ability to decide who and what is allowed to access consumer data. Visibility is the ability to monitor the status of a client's

[2]https://www.linkedin.com/pulse/cloud-battleground-wholl-win-skies-rahul-neel-mani/

data and program. In other words they don't know the location or the use of their data. Thus, they need to rely on the service provider to ensure that the platform is secure, and it implements necessary security properties to keep their data safe [Kazim and Zhu, 2015]. The new threats introduced by cloud computing can be classified as follows: data leakage vulnerabilities and virtualization based vulnerability. [Kazim and Zhu, 2015].

Data leakage vulnerabilities

Data loss

They mostly occurs due to malicious attackers, data deletion, data corruption, loss of encryption keys, faults in the storage system, or natural disasters [api, 2016a]. Data loss can have catastrophic consequences in the business, which may result in bankruptcy [api, 2016b]. Another source of data loss are insecure interfaces and APIs (Application Programming Interface) [Kazim and Zhu, 2015, M. Raju, 2014]. Cloud Services Providers (CSPs) expose APIs to third-parties to interact with their cloud services. In fact, whether launching applications services (SaaS), deploying their own applications (PaaS), or managing virtual machines (IaaS), cloud consumers use these APIs. Since the APIs are accessible from anywhere in the Internet, malicious attackers can use them to compromise the confidentiality and integrity of the enterprise customers [api, 2016a]. In [Georgiev et al., 2012], the authors show that some web applications such as payment services at Amazon and PayPal, have flaws in their implementation of the secure sockets layer (SSL) protocol that weaken their security when accessed through the APIs. Moreover, authentication and access control principles can also be violated through insecure APIs.

Cache-based side-channel

The prowess of modern cryptography led to the design of algorithms mathematically proven robust and secure [Rsa, 1977]. Although these algorithms are safe, their software and hardware implementations may be prone to attacks called covert-channel or side-channel attacks. A side-channel attack uses communication means that are not normally designed to leak the information [Kocher et al., 1999].

These attacks consist of two steps. First, a detailed analysis of the power consumption, the electromagnetic emanation or any other source or the execution timing of a cryptography system is made. Then, the exploitation of this analysis gives the attacker the ability to recover some bits of the encryption keys. In the cloud, the main micro-architectural leakage source is the cache. The side-channel attacks existed long before the emergence of cloud computing. The attacks in the cloud are possible thanks to the concept of multitenancy. The term multitenancy refers to a software architecture in which a single instance of software runs on a server and serves multiple tenants. A tenant is a group of users who share a common access with specific privileges to the software instance.

The first study that revealed the vulnerability of cloud computing platforms is [Ristenpart et al., 2009]. The attack was conducted on the public cloud platform Amazon

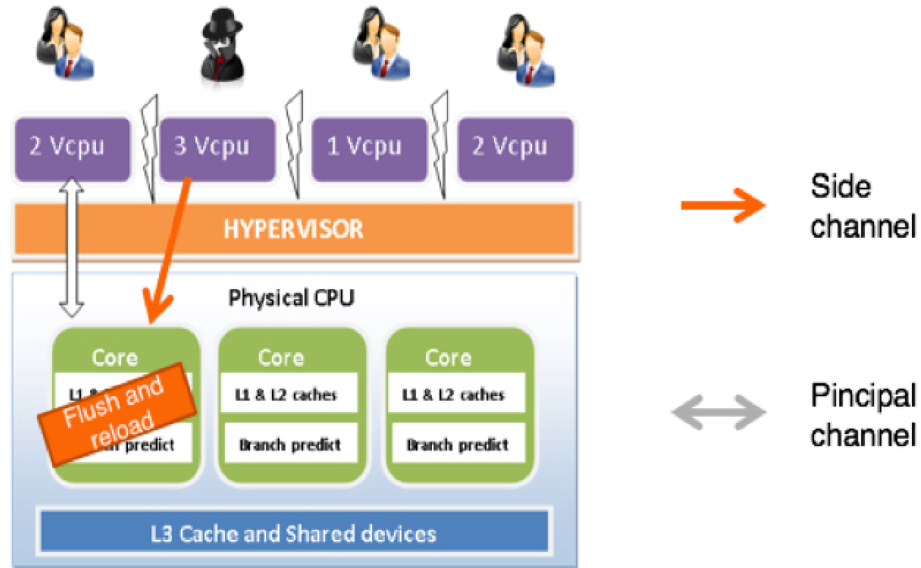

Figure 2.5 – Overview of a side channel attack on a cloud platform.

Web Services with the Xen hypervisor [3]. Note that a hypervisor is a software system that provides the launch, management and provisioning of virtual machines in a cloud platform. In the case of the Xen hypervisor, we distinguish a particular and privileged virtual machine called Dom0 that acts as orchestrator of other virtual machines and provides the management functions. The attack took place in two distinct steps. In the first step, the attacker make a mapping of IP addresses and launches its virtual machine (VM) on a the same platform (CPU, core) of the target VM. This is called co-residence. Two virtual machines are considered co-resident if one of the following three conditions is satisfied: i) They have relatively close internal addresses. ii) There are low network latencies between virtual machines.iii) The addresses correspond in part to the IP address of the Dom0 virtual machine.

In the second step, they used a technique called PRIME AND PROBE:

1. The spy fills one or more lines of cache with random data;
2. Then he lets out to the victim so that she can run;
3. Finally, the spy reload the same data in the same cache lines and measuring the time taken for charging.

A big loading time indicates that theses cache lines has been accessed by the victim and the spy data has been ejected from the cache. On the other hand, low duration shows that the relevant part of the L2 cache remained intact. With these measurements, the spy

[3] https://www.xenproject.org

inspected the traffic of a Web server and determine keystrokes of the victim. This attack even if it was a coarse-grained attack, gave researchers compelling reasons for addressing this new vulnerability [Zhang and Reiter, 2013, Zhang et al., 2012, Yarom and Falkner, 2014]. Nevertheless, this study has opened the field of research and served as a springboard for future side-channel attacks.

Zhang et al. showed in [Zhang et al., 2014] that the side-channel attacks were also possible on PaaS platforms. The attack was based on the method of FLUSH AND RELOAD. The FLUSH AND RELOAD is a variant of the method PRIME AND PROBE.

2.3.3 Virtualization vulnerabilities

Cloud computing leverages the concept of virtualization, that usually enables a new software layer (hypervisor) atop the operating system. However, virtualization introduces new security concerns [Studnia et al., 2012]:

- **Detecting a virtualized environment**: officially, a virtualized platform acts as a real platform, but an attacker has the possibility to detect if a machine is virtualized by measuring some instructions at execution time. Then, performing a comparison of the obtained values with previous measurements, the virtualization can be detected.

- **Identifying the hypervisor**: all of the hypervisors have their own vulnerabilities and flaws. Normally a cloud user cannot know on which hypervisor his/her VMs are running. However a spy can identify the hypervisor by using some specific instructions not well supported by each kind of hypervisor.

- **VM escape**: bad configuration within the hypervisor can break isolation between the hypervisor and the host. A virtual machine could directly interact with the host operating system. This would compromise all the data stored in the affected virtual machine and can affect other virtual machines deployed on the same host.

- **VM hopping**: this kind of attack is based on a virtual machine accessing other virtual machines. This vulnerability allows remote attacks and malware to compromise the overall virtual machine security, also granting access for the attackers to the host computer, the hypervisor and other virtual machines.

- **Malicious VM creation**: the creation of malicious virtual machines constitutes a high risk for the system since they can replace the legitimate virtual machines offering similar services, but making a malicious use of the exchanged data.

After describing the main attacks Web services and cloud platforms can face, we will present the current state of art of tolerance. We will first review general attack tolerance techniques. After, we will present the techniques specific to Web services.

2.4 Intrusion and attack tolerance for Web services

2.4.1 Attack tolerance techniques

Intrusion tolerance comes from fault tolerance, a term used in dependability [Deswarte and Powell, 2004]. Dependability is a generic notion that measures the trustworthiness of a system, so that the users have a justified trust in the service delivered by that system [Saha, 2007]. It mainly includes four components: reliability, maintainability, availability and security. Dependability has emerged as a necessity in particular with industrial developments. The goal was to build systems that are reliable and contain near-zero defaults. As IT systems are facing both diversified and sophisticated intrusions, intrusion tolerance, can been considered as one of the crucial attributes of dependability to maintain. According to [Deswarte and Powell, 2004], the common vulnerability sequence of fault tolerance is the fault-error-failure sequence. For them, a system failure occurred when the service its delivers deviates from the normal tasks it was supposed to execute. Moreover an error is any part of a system that can lead to a failure. They defined a fault as the probable cause of an error. Applying this sequence in the context of security, an intrusion can be defined as a malicious fault resulting from a successful attack on a vulnerability [Veríssimo et al., 2003]. According to the recent development trends and advancement of intrusion detection, detecting all kinds of intrusions effectively requires a holistic view of the monitored network. It is impossible in practice to detect all attacks targeting a computer system.

Intrusion or attack tolerance of a system is then the capability of that system to continue to function properly with minimal degradation of performance, despite intrusions or malicious attacks. Several approaches and techniques were proposed in the literature. The goal of the work in [Wang et al., 2003] is to identify common techniques for building highly available intrusion tolerant systems. They mention that a major assumption of intrusion tolerance is IT systems can be faulty and compromised and the main challenge consists in continuing to provide (possibly degraded) services when attacks are present. In addition, the main techniques used for attack tolerance ([La, 2016]) are:

- **Indirection**: It separates clients and servers by an additional layer that can be considered as another protection barrier. Proxies, wrappers, virtualization, and sandboxes are some examples of indirection techniques.

- **Voting**: The voting technique allows to detect misbehavior from the outputs observed by a set of servers. Indeed, the servers receive the same requests and any differences of their responses can lead to a consensus result based on the responses of perceived non-faulty components in the system.

- **Redundancy and diversity**: Redundancy refers to the extra reserved resources allocated to a system that are beyond its need in normal working conditions. Diversity is the idea of having many forms of the same object. It is argued that design diversity reduce the probability of intrusion [Veríssimo et al., 2003].

- **Threshold cryptography** : The general idea is to fragment a secret S into n pieces in such a way that it needs at least k shares to reconstruct original S. Anything less reveals no information.
- **Fragmentation redundancy scattering (FRS)**: It consists in splitting sensitive data, making them redundant and isolating fragments obtained.
- **Dynamic reconfiguration**: Reconfiguration takes place after the detection of an intrusion. In traditional systems reconfiguration is mostly reactive and generally performed manually by the administrator.
- **Detection and recovery**: It consists in detecting error after an intrusion has been activated. Error recovery means recovering from the error once it is detected, forwarding the systems to a state that ensures correct provision [Veríssimo et al., 2003].

Furthermore, there are several solutions that provide attack tolerance using one or a combination of such techniques ([Mishra et al., 2017], [Yan et al., 2016], [Meixner et al., 2016], [Raj and Varghese, 2011], [Saidane et al., 2009]). The work in [Constable et al., 2011] explores how to build distributed systems that are attack-tolerant by design. The idea is to implement systems with equivalent functionality that can respond to attacks in a more safe way. [Madan and Trivedi, 2004] propose a formalism based on graphs to model an intrusion tolerant system. In this model they introduce system's response to (some of) the attacks. They call this model that incorporates attacker's actions as well as the system's response an Attack Response Graph (ARG). Other approaches have been developed to cope with intrusion-tolerance.

[Deswarte et al., 2002, Nicomette et al., 2011] proposed an hybrid authorization service. The main contribution of that study is the introduction of an Intrusion tolerance authorization scheme. In this scheme the system is able to distribute proofs of authorization to the participants of the system. For the authors, an application is a set of objects interacting through method invocations. Thus, a proof of authorization allows objects to execute operations. Their architecture is based on two main components: the authorization server and the reference monitors. The authorization server grants or denies rights for operations involving several participants. The reference monitor controls the access to the resources on each participating host. Performance measures obtained from an implementation of this architecture, reveals that this framework needs specialized hardware for the best implementation of cryptographic operations.

Besides, [Nguyen and Sood, 2010], [Wang et al., 2003] and [La, 2016] classify ITS (Intrusion Tolerant Systems) architectures into four categories :

- **Detection-triggered** [Valdes et al., 2002, Valdes et al., 2004, Reynolds et al., 2003]: these architectures build multiple levels of defense to increase system survivability. Most of them rely on an intrusion detection that triggers reactions mechanisms.
- **Algorithm-driven** [O'Brien et al., 2003, Verissimo et al., 2006, Zhang et al., 2005, Sliti et al., 2009] : these systems employ algorithms such as the voting algorithm,

threshold cryptography, and fragmentation redundancy scattering (FRS) to harden their resilience.

- **Recovery-based** [Aung et al., 2005, Arsenault et al., 2007, Sousa et al., 2008, Reiser and Kapitza, 2007] : these systems assume that when a system goes online, it's compromised. Periodic restoration to a former good state is necessary.
- **Hybrid** [Sousa et al., 2007a]: these systems combine different architectures mentioned above.

Table 2.1 – Mapping between attack tolerance mechanisms and architectures

Mechanisms \ Architectures	Detection	Algorithm	Recovery	Hybrid
Indirection	[Wang and Upppalli, 2003, Chen et al., 2009, O'Brien et al., 2003]	[Stroud et al., 2004]	✗	✗
Diversity	[Wang and Upppalli, 2003, Chen et al., 2009, O'Brien et al., 2003]	✗	✗	[Lala, 2003]
FRS	[Stroud et al., 2004, Zhang et al., 2005]	✗	✗	✗
Threshold cryptography	✗	[Ganger et al., 2001, Weinstein and Lepanto, 2003]	✗	[Lala, 2003]
Recovery	[Knight et al., 2001, Wang and Upppalli, 2003, Chen et al., 2009]	✗	[Aung et al., 2005, Arsenault et al., 2007, Sousa et al., 2008, Reiser and Kapitza, 2007]	✗
Voting	[Wang and Upppalli, 2003, Valdes et al., 2002]	[Partha et al., 2006]	✗	✗

Table 2.1 depicts the mapping between the attacks and the architectures. This study shows firstly that these architectures ensure tolerance to attacks for conventional distributed

systems. One well known architecture is MAFTIA (Malicious and Accidental Tolerance Intrusion Architecture [Stroud et al., 2004]). The framework presented in [Karande et al., 2011] leverages the MAFTIA framework for cloud platforms. The detection relies on event analysis, while the intrusion detection is based on threshold cryptography. After the detection, the recovery module reallocates the VM running on these hosts and the hosts are turned off. But, the main bottleneck of this architecture is the performance overhead.

The survey also exhibited the strengths and the weaknesses of each architecture. The authors claimed that while the architectures proposed in the literature are efficient some enhancements need to be taken into account. In fact, in some approaches the detection algorithm depends on the application monitored and must be developed specifically for each application considered [Totel et al., 2006]. Moreover some rules proposed for the detection are sometimes naive (HTTP headers verification). Furthermore, despite good results are obtained, in some cases there are some false positives and the identification of the intruded server is not always possible. So the main conclusion of that study is that the complementary combination of these architectures can lead to the design of more efficient architectures. Our intrusion tolerance approach for Web services in this thesis will also combine the attack tolerance mechanisms in a coherent manner by incorporating new detection methods as we will see in the next chapters. As we will leverage diversity later, we will present its features in detail in the next section.

2.4.2 Diversity techniques

Definition 1. *Diversity*[4] *is the quality or state of having many different forms, types, ideas, etc. Diversity is used in theory and practice to a much greater extent in other disciplines. As an example, biodiversity is the fact that it can exist many forms of life in the nature.*

In computer science, many kinds of software diversity exist [Baudry and Monperrus, 2015]:

- Diversity for fault tolerance, or security, re-usability, software testing;
- Diversity for networks, operating systems, components, data structures;
- Diversity in market products.

As our work targets intrusion tolerance, we will concentrate on the use of diversity as a mean for achieving intrusion tolerance. We will list some existing approaches for diversity and show how they can be applied or enhanced to fit our requirements. There exists several types of diversification:

Design diversity

Design diversity refers to diversity at the early stage. N-version programming is one of the techniques in this area. N-version programming technique can be divided into

[4] http://www.merriam-Webster.com

two steps. The first consists in giving the same specification of a software product to N (N>=2) independent teams of developers. The implementation are coded using different programming, platforms, languages and run-time environment. This ensures independent bugs from all teams. After the development, these implementations are run in parallel and checked with a voting system. This reduces drastically the presence of bugs. N-version programming can be enhanced for example by forcing developers to use different data structures and different algorithms. Another design diversity mean is Recovery blocks ([Pen et al., 2014] [Xu et al., 2016]). Recovery blocks were proposed as a way of structuring the code, using diverse alternative software solutions, for fault tolerance. The idea is to have recovery blocks in the program. These spares are diverse variant implementations of the same function.

Compiler-based diversity

Multi-compiler [Franz, 2010] is an approach based on compiler. The functionality of this framework is the following. From the source code of any software, the multi-compiler tool generates several different binaries. The diversity engine leverages some transformations such address space randomization, NOP operations insertions. All the different versions of the same program behave in the same way from the point of view of the end-user. The idea is to increase the cost of the attacker activity, implementing programs with equivalent functionality that can respond to attacks in a more safe way. As a result a number of code variants are produced, which ensures the system will be more resistant to attacks. In conclusion compiler based diversification decreases drastically the risk of code injection and attacks like buffer overflow attacks.

Source code or runtime diversity

Some other works, rather than addressing diversification at the compiler layer, tried to produce functionally equivalent programs different from each other by the control flow. [Allier et al., 2015] provided 9 operations that transform the AST (Abstract Syntax Tree) of the target program. This reduces the predictability of the program's computation. Multi-variant code execution is another runtime technique that prevents malicious code execution by running a few slightly different instances (variants) of one program in lockstep and comparing their behavior against each other. It is a dynamic version of N-version programming. Any divergence among behavior of the variants at these synchronization points is an indication of an anomaly in the system and raises an alarm. Before each instruction is executed, the instructions are examined to ensure that the instructions and operands are the equivalent. Furthermore, in the literature some approaches, like metamorphic and polymorphic programming, have been investigated. These techniques are used by attackers to create very dangerous and not easy to detect malwares. Metamorphic malware is a malware that automatically re-codes itself each time it propagates or is distributed[5]. This attack can be develop through some simple operations such as:

[5] https://www.blackhat.com/presentations/bh-usa-08/Hosmer

- Adding varying lengths of NOP instructions
- Permuting use registers
- Adding useless instructions and loops within the code segments

Some other strong operations could be used :

- Reordering structures
- Program flow modification
- static data structures modification

On can use this method not to attack but to defend itself against other types of attack. But in practice building a polymorphic code requires very deep programming capabilities.

Multi-layer diversity

Because the above framework only address a particular class of vulnerabilities, recent works aim at combining many of these approaches in order to enhance the security and intrusion tolerance are introduced. [Collberg et al., 2012] composed multiple forms of diversity and code replacement in a distributed system in order to protect it from remote man-at-the-end attacks. This paper presents a new method to address Remote Man at the end attack. Remote man-at-the-end (R-MATE) attacks occur in distributed systems where untrusted clients are in frequent communication with trusted servers over a network, and malicious users can get an advantage by compromising an untrusted device.

[Obelheiro et al., 2006] show how to achieve intrusion tolerance in practice. They introduced two main concepts that are useful when one is trying to construct an intrusion tolerance system:

1. Axis of diversity: a component of a system that may be diversified,
2. Degree of diversity: the number of choices available for a specific axis of diversity.

They explained that there exist many axis of diversity among which we can notice: application, administrative, location, Commercial-off-the-shelf (COTS) Software DBMS (Data Base Management System) Middleware Virtual Machines, Compilers Libraries), Operating System, Security Methods, Hardware. They performed their experiments with six axes of diversity: implementation, execution environment (COTS), database (COTS), operating system, hardware, and location and a degree of diversity.

The following table 2.2 depicts the diversity mechanisms

Table 2.2 – Diversity techniques

Technique	Description	Target attack
N- version programming	N different versions of the program are launched and a voting mechanism checks source code errors	Bugs on software; requires a bug free specification
Recovery Blocks	Acceptance test with functionally equivalent spares modules	Hardware and software faults
Multi Variant Execution	dynamic N-version programming, error checking	Malicious code injection
Multi compiler	Diverse binaries for a source code	Buffer/heap overflow attacks
"Sosification"	Change the control flow of the source code	lack of input validation or business logic vulnerabilities
Homomorphic and polymorphic programming		

2.4.3 Attack tolerance techniques for Web services

Other works were conducted in order to transpose the aforementioned techniques and framework to Web services. [Sadegh and Azgomi, 2015] presented an attack tolerant Web services architecture based on Diversity techniques we mentioned above. They designed several composite Web services (intrusion detection Web services, intrusion containment Web services, etc. Ficco and al. [Ficco and Rak, 2011] on contrary, proposed an attack tolerant framework targeting Stealth DoS Attacks. In fact according to the authors background, Stealth-DoS attack seems difficult to detect because such attacks behaviors are variable and polymorphic. In order to protect against such attacks, they combine several anomaly based attack tolerance techniques. There is a dynamic threshold, an average number of nested XML tags that corresponds to a normal CPU usage, determined according to the resource overhead consumption.

While these approaches are interesting, some limits remain. First, the solutions are attack-specific. Moreover, there is no evidence that these frameworks protect against silent but very dangerous attacks. These attacks are actually developed by hackers who apply their acquired knowledge on learning and exploiting the vulnerabilities of the target application. The traffic then during these attacks may seem a priori legitimate. Therefore, the differentiation between malicious traffic and normal traffic is very difficult to achieve by conventional detection and intrusion tools. It requires in-depth expertise. In conclusion, attack tolerance for Web services is now quite insufficient. In conclusion we need a more efficient intrusion-tolerant mechanism. We will see in the next section other ways of thinking about attack tolerance namely using formal methods and monitoring.

2.5 Formal methods

One of the open problems in software engineering is the correct development of computer systems. We want to be able to design safe systems. A secure design of a software refer to techniques based on mathematics for the specification, development, and verification of software and hardware systems. The use of a secure design is especially important in reliable systems where, due to safety and security reasons, it is important to ensure that errors are not included during the development process. Secure designs are particularly effective when used early in the development process, at the requirements and specification levels, but can be used for a completely secure development of a system. One of the advantages of using a secure representation of systems is that it allows to rigorously analyze their properties. In particular, it helps to establish the *correctness* of the system with respect to the specification or the fulfilment of a specific set of requirements, to check the semantic *equivalence* of two systems, to analyze the *preference* of a system over another one with respect to a given criterion, to predict the possibility of *incorrect behaviors*, to establish the *performance* level of a system, etc. Formal methods are well suited to address the above mentioned issues as there are based on mathematical foundations that support reasoning. In this section, we briefly present the different techniques that are part of this set of methods [Attiogbe, 2007].

2.5.1 Static analysis

In these techniques, the code is not executed but some properties are proven. There are several methods among them:

- **Theorem proving:** The specification of a program is seen as a theorem to demonstrate. The reduction of a conjecture, by successive applications of deductive rules and axioms, constitutes a proof of this conjecture. The interpretation of the failure of the proof is delicate; indeed, either the property is not demonstrable (problem of non-decidability) or there are not enough elements or strategies to lead to the demonstration. Proof assistants are software or software platforms that allow help the user through the various steps of a proof. Most of the time they have a set of theories with their axioms and logical systems. Examples of proof assistant are: Coq, Isabelle etc.
- **Model Checking:** An abstract representation of the system is used to check the desired properties. It consists in exhaustively exploring the state space of that given representation (often a transition system or a graph) for the purpose of verifying properties that are true or false. The result of the evaluation is either a confirmation that the properties are true in all states of the input model, or a series of states leading to a state or property is not true: it is a counter example. Examples of model checkers are: Cubicle, SMV (Symbolic Model Checker for CTL), Spin etc.
- **Refinement methods :** Refinement is fundamentally a relationship between an abstract object and a less abstract one; one refines the other. Such methods start

from the specification of a problem and implements more and more accurately the program until an executable code is obtained. Each stage of the refinement is proven correct.

- **certified programming :** is based on the correspondence between mathematical proofs and programs. The specification of a program is associated with a logical formula (a theorem). From a proof of this formula, one extracts a program and a certificate from this program.

- **Abstract interpretation :** Abstract interpretation [Cousot and Cousot, 1977] formalizes the idea that a formal proof can be done at some level of abstraction where irrelevant details about the semantics and the specification are ignored. This amounts to proving that an abstract semantics satisfies an abstract specification.

2.5.2 Dynamic analysis

It is the most widespread. It consists of executing the code or simulating it in order to reveal any bugs. Software testing consists in comparing the result of a program with the expected result. For this method to be effective, it is necessary to test the various possible situations. There are two types of tests:

- **Functional tests:** that consider the program as a black box and are only established from the knowledge of program specification;

- **Structural tests:** that, from the knowledge of the program, seek to execute tests that cover all parts of the code.

A particular type of dynamic analysis is formal monitoring which remains more used for the detection of attack. This is why we will deeply present formal monitoring.

What is monitoring

Monitoring is the process of dynamically collecting, interpreting and presenting metrics and variables related to a system's behavior in order to perform management and control tasks [Stankovic and Strigini, 2009]. The idea behind monitoring is to measure and observe performance, connectivity, security issues, application usage, data modifications and any other variable that permits to determine the current status of the entity being monitored. By keeping a constant view of the different entities, we can obtain a real-time status of Key Performance Indicators (KPI) or Service Level Agreements (SLA) compliance as well as faults and security breaches. Monitoring can be performed in several domains that include user activity, network and Internet traffic, software applications, services and security. The monitoring processes should not disturb the normal operation of the protocol, application, or service under analysis.

Monitoring techniques

The general processes involved in monitoring are: definition of the detection method to track and label events and measurements of interest; the transmission of the collected information to a processing entity; the filtering and classification ; and finally, the generation decisions associated to the results obtained after the evaluation [Stankovic and Strigini, 2009]. Regarding how to collect events and measurements, monitoring techniques can be classified into three main categories: active, passive and hybrid approaches.

Active monitoring: the System Under Observation (SUO) is stimulated in order to obtain responses to determine its behavior under certain circumstances or events. This technique permits directing requests to the concerned entities under observation. However, it presents some drawbacks. The injection of requests towards the SUO might affect its performance. This will vary depending on the amount of data required to perform the desired tests or monitoring requests. For large amounts of data, the SUO processing load might increase and produce undesirable effects. Secondly, the injected information might also influence the measurements that are being taken, for example incurring in additional delay. Lastly, active monitoring injects data that could be considered invasive. In a network operator context, it could limit its use and applicability [Chen and Hu, 2002].

Passive monitoring: consists on capturing a copy of the information produced by the SUO without a direct interaction [Curtis, 2000]. This technique reduces the overhead required on active monitoring. Conversely, certain delay should be considered when analyzing large amounts of data. Additionally, in some cases it is not always possible to perform real-time monitoring because of required offline data post-processing [Anagnostakis et al., 2002]. This technique has the advantage over the active approach of not performing invasive requests.

Hybrid monitoring: by combining the aforementioned approaches we can perform both active and passive monitoring. The idea is to keep passive trace collection and to inject requests as necessary to maintain a continuous flow of observations and measurements [Zangrilli and Lowekamp, 2004]. In this case, pre-configured, on-demand, or event-based monitoring requests can be performed. Hybrid monitoring can be used in systems with limited resources, for example by periodically sending passive traces (with limited information) and requesting details if complementary information is needed. In this way, the resources consumed by the monitoring modules are reduced.

The selection of one of these monitoring approaches will depend on different premises like the level of intrusion allowed by the SUO, the type of data collected (which sometimes does not require a direct request to be obtained), the specified security policies, or post-processing and real-time constrains.

In conclusion, it is important to emphasize that test methods as important as they are not exhaustive. We can only test a number of values. In general, therefore, they do not constitute proof of correction of the program like in static methods. However [Moy and Wallenburg, 2010] list some reasons why static methods may still contain defects: i) not all parts of the product are formally verified, ii) not all properties can be formally verified, for

example covert channels can be hard to detect as well as dead code. So a better detection of bugs or attacks requires a joint use of both static and dynamic analysis techniques. So, enhancing trust for cloud services consumers by leveraging monitoring, cryptography and formal methods is the aim of the CLARUS project. We will present an overview of this project in the next section.

2.6 Discussion

Web services are increasingly used to develop Enterprise Service Oriented Architectures. Web services are also the target of cyber attacks. Moreover sometimes they leveraged cloud computing for their deployment. Cloud computing, as a paradigm of distributed systems, has emerged over the last decade. Cloud computing has revolutionized the industry. However, as discussed above, cloud infrastructures are the classic as well as specific attack targets. Web service then inherit the vulnerabilities of cloud computing. We have also presented techniques to detect and mitigate attacks. To date, there are very few that are able to manage the intrusions and the attacks and be able to insure the reactions and counter measures with the aim of protecting the system and guaranteeing its normal behavior in hostile environments.

Moreover, intrusion detection technologies based on signatures can efficiently catch known attacks. However, they will be no longer effective because of the rapid growth of new types of attacks and, indeed, attackers are constantly adapting their techniques to evade new means of protection or firewall policies. In addition, the construction of attack signature is a very vulnerable and delicate step. Indeed, the signatures must be sufficiently precise to match as closely as possible to attack and thus not generating too many false positives while being sufficiently generic to be able to detect sudden variations of the same attack. Otherwise, an attacker can modify the scenario of an attack so that it retains its power to harm but it no longer matches the pattern.

To cope with all these issues it is necessary to consider information security as a permanent issue that needs to be managed in order to obtain attack-tolerant Web services. In this thesis we propose to design an attack tolerant system that integrates intrusion detection methods, formal methods, diverse defence strategies. By means of constant monitoring, we will provide an attack-tolerant framework, so that potential security breaches within can be dynamically detected and appropriate mitigation measures can be activated on-line, so reducing the effects of the detected attacks. As a result, we ensure a total attack tolerance of Web services from both formal and practical points of view. We will demonstrate and evaluate practical implementations of the proposed framework, in the next chapters. The contributions of this chapter have been published in the proceedings of the International Conference on Distributed Computing Systems Workshops (ICDCSW 2016) [Ouffoué et al., 2016].

3

Risk-based passive monitoring

Contents

2.1 Research on Web services 20
2.2 Security issues related to Web services 22
2.2.1 XML DoS 22
2.2.2 Metadata Spoofing 23
2.2.3 SQL Injections 23
2.2.4 Capture and Replay Attacks 23
2.2.5 Session Hijacking 23
2.2.6 WSDL scanning 24
2.2.7 Parameter tampering 24
2.2.8 External reference attack 24
2.3 Cloud computing security issues 24
2.3.1 Cloud computing in a nutshell 24
2.3.2 Cloud Market and challenges 27
2.3.3 Virtualization vulnerabilities 31
2.4 Intrusion and attack tolerance for Web services 32
2.4.1 Attack tolerance techniques 32
2.4.2 Diversity techniques 35
2.4.3 Attack tolerance techniques for Web services 38
2.5 Formal methods 39
2.5.1 Static analysis 39
2.5.2 Dynamic analysis 40
2.6 Discussion 42

As it has been presented in the state of the art (Chapter 2), cyber attacks are multiplying and becoming more and more sophisticated. We seen that in order to better tolerate and limit the impact of these attacks, the monitoring of the information systems is of paramount importance for any organization. Traditional intrusion detection systems are deployed to identify and inhibit attacks as much as possible. Usually, the detection of anomalies is based on the comparison of observed behaviors with normal behaviors, previously established. An alert is raised when these two behaviors differ and in case of dysfunction of the information systems and they are able to act accordingly. Moreover, monitoring makes it possible to analyze in real time the state of the computer system and the state of the computer network for preventive purposes.

However, we believe that the monitoring and detection of attacks require an awareness of the risks that the system might be exposed to. This is why we propose in this chapter the notion of risk-based monitoring that consists in thinking of the risk when deploying a monitoring mechanism. We will first briefly present the foundations of this methodology and then we will describe the metrics we identified for detecting some known attacks. This methodology has been implemented in the context of the CLARUS H2020 project[1] and the metrics proposed are used for the detection part of the attack tolerance approaches described in the next chapter. CLARUS is a H2020 European research project [Sánchez and Domingo-Ferrer, 2015] that focuses on providing solutions to the usual security and privacy threats that affect cloud computing and hinder a franker migration by end users. Finally we will present Montimage Monitoring Tool (MMT), our main detection tool leveraged in this thesis.

3.1 Risk-based monitoring methodology

The supervision or monitoring of information systems is of paramount importance for any organization. It essentially consists in deploying probes in various parts of the system based on pre-established checkpoints. With automatic failure reporting, network agents can respond to key security risks. The disadvantage of such method is the following. If risks or failures are discovered during operation, the attacks may have already occurred and one or more parts of the system may be non-functional. So, the detection and prevention of attacks require a good knowledge of the risks that these systems face. As such, it is mandatory to include risk management in the monitoring strategy in order to reduce the probability of failure or uncertainty. Risk management attempts to reduce or eliminate potential vulnerabilities, or at least reduce the impact of potential threats by implementing controls and/or countermeasures. In the case, it is not possible to eliminate the risk, mitigation mechanisms should be applied to mitigate their effects. The traditional risk management loop at runtime, is composed of six elements: assets, threats, vulnerabilities, risk, security incidents detection, and remediation. We leveraged this loop to build our

[1] http://clarussecure.eu

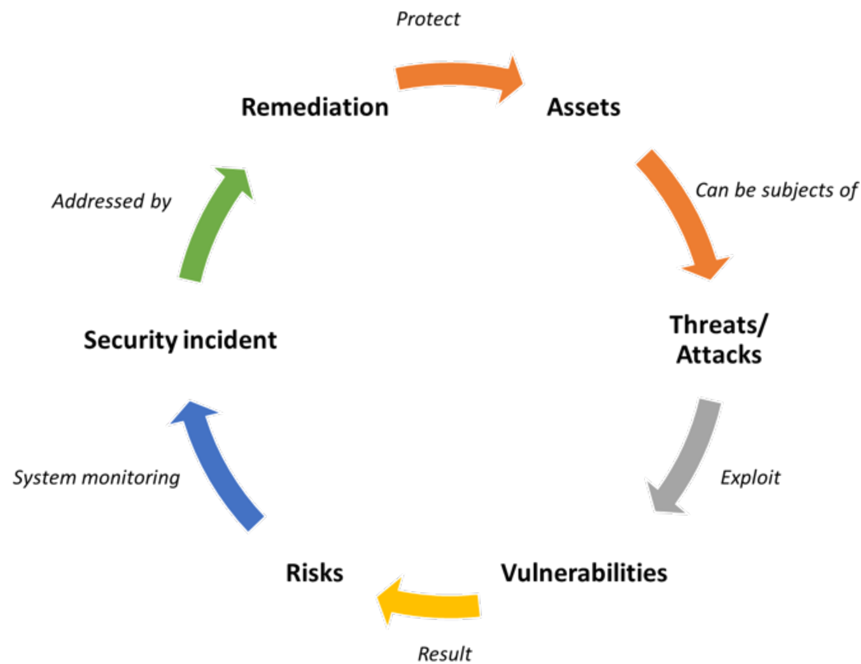

Figure 3.1 – Risk-based monitoring loop

risk-based monitoring loop depicted on the Figure 3.1 Indeed, this risk-based monitoring solution can be summarized in the following objectives:

1. Identification of system assets,
2. Risk analysis to categorize threats that can exploit system vulnerabilities and result in different levels of risks,
3. System monitoring to detect potential occurrences of attacks, and finally,
4. Remediation strategies to apply corrective actions for mitigating the impact of the attack on the target system.

The step 1 to 3 are described in detail below. As an example, the CLARUS proxy will be used for describing the approach. The step 4 is described in part here because the core remediation strategies will be described in the next following chapters.

3.1.1 Identifying Assets

Assets are defined as proprietary resources of value to the organization and necessary for its proper functioning We distinguish business-level assets from system assets. In terms of business assets, we mainly find information (for example credit card numbers) and

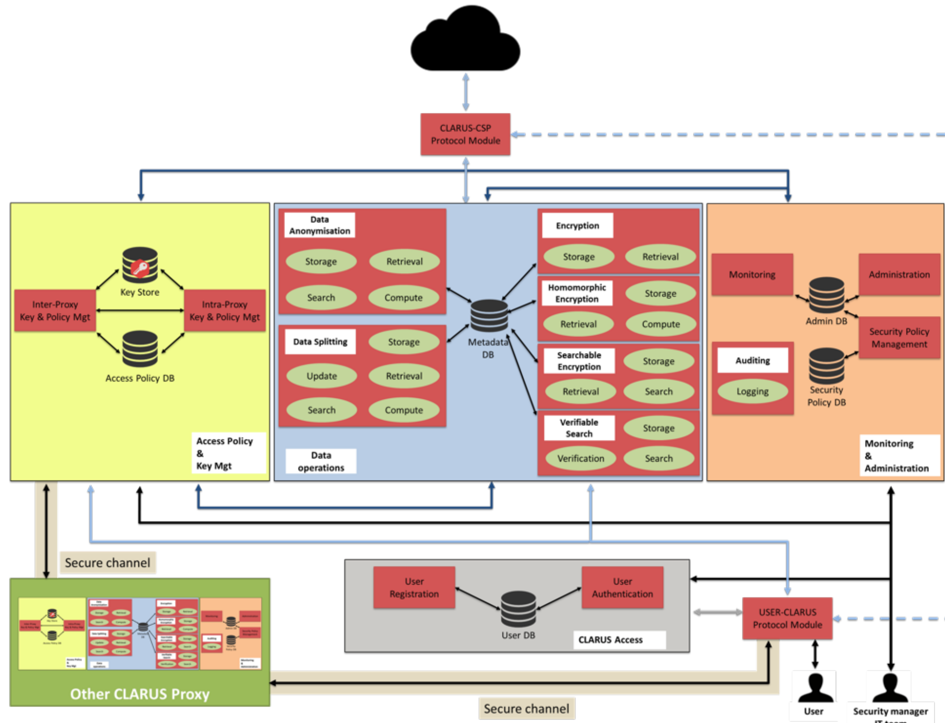

Figure 3.2 – CLARUS proxy architecture

processes (such as transaction management or account administration). The business assets of the organization are often entirely managed through the information system. System assets include technical elements, such as hardware, software and networks, as well as the computer system environment, such as users or buildings. System assets can also represent some attributes or properties of the system such as the data integrity and availability. This is particularly true for cloud services consumers.

For a user using the clarus proxy, the main assets can be summarized as:

- **Sensitive data integrity**: users data must be only accessed by authorized people, avoiding any third-party use or access.
- **Availability**: The CLARUS proxy (Figure 3.2) should be available even in the presence of threats, minimizing the impact on the overall system operation.
- **Data Control**: users must have more trust and knowledge of how their data is managed.
- **Data access**: users should have the possibility to securely access or store their data through CLARUS.

3.1.2 Risk and vulnerability analysis

Risk is the possibility or likelihood that a threat will exploit a vulnerability resulting in a loss, unauthorized access or deterioration of an asset. A threat is a potential occurrence that can be caused by anything or anyone and can result in an undesirable outcome. Natural occurrences, such as floods or earthquakes, accidental acts by an employee, or intentional attacks can all be threats to an organization. A vulnerability is any type of weakness that can be exploited. The weakness can be due to, for example, a flaw, a limitation, or the absence of a security control. So after identifying valuable assets, it is necessary to perform vulnerability analysis. This type of analysis attempts to discover weaknesses in the systems with respect to potential threats. For example, in the context of access control, vulnerability analysis attempts to identify the strengths and weaknesses of the different access control mechanisms and the potential of a threat to exploit these weaknesses. For access control in CLARUS, we identified the following potential vulnerabilities related to:

- Access control (console authentication and Single Sign-On (SSO) mechanisms): the usurpation of authentication credentials allows access to all of the CLARUS proxy functionality.
- The mapping of information between the "User-CLARUS" module, the data mapping module and the explicit authentication: an application uses a SQL database to store and query its data. The connection is established through a network and the application directly connects to the database server. If the application is used with CLARUS, the network server in the application would be changed to connect to the CLARUS proxy instead. The proxy then offers a module ("USER-CLARUS Protocol" module) that inspects the incoming data and picks out the data that should be secured while handing over the rest of the data directly to a corresponding storage system. To be able to match a user session with the incoming data stream, the data stream is inspected for user identification traits, for example SQL login phrases, HTTP tokens or similar unique identification elements. Vulnerability can arise if an attacker knows the protocol the "User-CLARUS" modules are using (PostgreSQL, OGC web services, S3...) and can intercept traffic and steal sensitive information or reuse the incoming data to further get access to CLARUS.
- CLARUS access metadata modification: considering the case of the policy server not being adequately protected, an attacker may change the policies, granting access to unauthorized users.

3.1.3 Threats Modelling

The first step to avoid or repel the different threats that can affect an asset is to model them by identifying: affected modules/components, actions/behaviour to trigger the threat, and potential objective of the threat. The formal model of a threat helps to understand the operation of the attacks and allows the creation of security mechanisms to protect, not only the assets, but also the software mechanisms that support them. Once the threats

are modelled, we can identify the vulnerabilities that can affect the system and define monitoring and remediation mechanisms to minimize the damages that might occur. Again, considering the access control example: An access control process has two main steps: authentication and authorization. The latter usually comes after the former in a normal workflow. The authentication step is the more critical part of the access control process. The following description illustrates this assertion: Let's assume that an attacker successfully impersonates the account of a legitimate user. If the user is the administrator, then the attacker can create a fake user and grant himself/herself all the necessary privileges for further attacks, or directly steal sensitive information with the administrator's capabilities.

3.1.4 Attack scenarios

We focused on two kinds of attacks: access by unauthorized users, DoS/DDoS. Attack tree formalism will be used for analyzing common attacks against CLARUS. Attack trees provide a formal, methodical way of describing the security of systems, based on varying attacks [Schneier, 1999]. Attack trees are represented in a graphical view by constructing all the possible attacks and then by differentiating the most effective attacks with those that are beyond the capability of attacker. The remaining attacks are the ones that may cause damage to the system. The construction of an attack tree starts by placing the goal of an attack at the top of the tree. Attacks against a system are represented by the tree structure, with the goal as the root node and the different ways of achieving the goal as leaf nodes.

Access by unauthorized users attack

Access control can be evaluated to identify threats that can bypass authentication or authorization mechanisms. Attacks that can occur against the CLARUS authentication or authorization mechanisms are:

- **Dictionary Attacks** A dictionary attack is an attempt to discover passwords by using every possible password in a predefined database or list of common or expected passwords [Hansche et al., 2003]. In other words, an attacker starts with a database of words commonly found in a dictionary. Dictionary attack databases also include character combinations that aren't normally found in a dictionary but are commonly used as passwords.

- **Brute-Force Attacks** A brute-force attack is an attempt to discover passwords for user accounts by systematically attempting all possible combinations of letters, numbers, and symbols [Science, 2013]. Attackers use programs that can systematically try all possible combinations. A hybrid attack combines several techniques and, for instance, attempts a dictionary attack and then performs a brute-force attack with one-upped-constructed passwords.

- **Rainbow Table Attacks** When attempting to find passwords the time it takes can be reduced by using a rainbow table. Rainbow tables are large databases of

pre-computed cryptographic hashes of guessed passwords [Tipton, 2009]. This allows deriving a password by looking at the hashed value stored in the table.

- **Sniffer Attacks** A sniffer attack occurs when an attacker uses a sniffer to capture information transmitted over a network [Hansche et al., 2003]. Any data sent over a network in clear text, including passwords, can be captured and read by the program.
- **Access aggregation** [Tittle et al., 2006] refers to collecting multiple pieces of non-sensitive information and combining or aggregating them to derive sensitive information.
- **Reconnaissance attacks** are access aggregation attacks that combine multiple tools to identify multiple elements of a system, such as IP addresses, open ports, running services, operating systems, and more.

As an example, the following attack scenario can arise: the attacker tries to intercept any information available through the communication between the user and the "User-CLARUS" module with the aim of collecting data to perform a brute force attack. For that, information such as the protocol/s used and other private information that is exchanged in the communications can be gathered via a network sniffer. The attacker can then aggregate all the collected information and start the brute force attack using also a Dictionary-guided search, so increasing the success possibilities. The typical attack tree for an intrusion in a system is depicted in Figure 3.3.

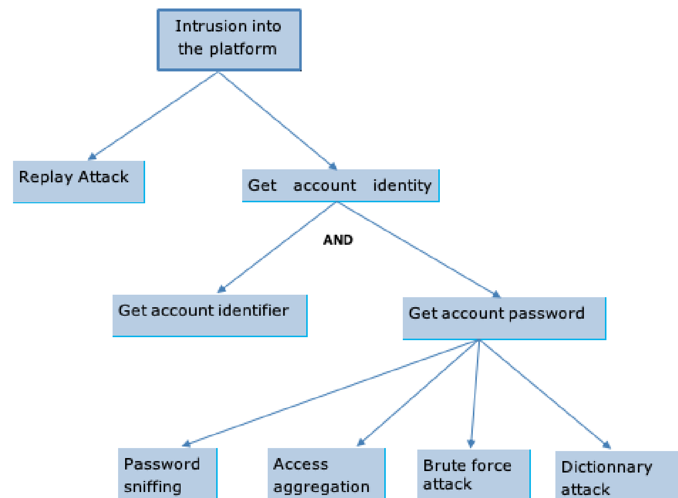

Figure 3.3 – Attack tree for unauthorized users attack

There is one branch containing the Boolean condition AND. The remaining branches conditions are implicitly OR branches. To intrude into the system, an attacker has 2 choices: either launch a replay attack or try to get account identity. This is why the

Boolean operator is OR. Before getting account identity, the intruder needs to find the login and the password of the victim. So there is an AND operator. Guessing login is quite straightforward as it is usually the name or the email of the victim. It results in a leaf node. Finally, getting the victim's password is done, for instance, by performing password sniffing, an access aggregation attack, a Dictionary attack, a brute force attack, or a combination of these techniques. The intrusion scenario can also be expressed by the following statement:

> *(Replay attack), (account-identifier, sniff-password), (account-identifier, password-access aggregation), (account-identifier, password-dictionary-attack), (account-identifier, password-Brute-force attack)*

Dos/DDos attacks

Denial of Service (DoS) attacks are based on attackers attempting to prevent legitimate users from accessing information or services. The most common DoS attack is based on flooding requests to the available services to overload the system. If an attack comes from a distributed set of sources, or the attacker uses third-party devices to perform the attack, it is known as a Distributed Denial of Service (DDoS) attack.

Moreover, the first detected symptom of a DoS/DDoS attack directed to a CLARUS proxy will be a reduction of the proxy's performance or the total denial of access to the proxy. To detect these situations, the monitoring module needs to constantly obtain information on the overall performance of the proxy, including response time, IP address, network information, CPU statistics, and storage statistics:

- **Number of requests per second**: refers to the number of requests that a CLARUS proxy receives in a second. Normally, this value will be very low. However, in the presence of a DoS/DDoS attack, it will tend to grow rapidly.
- **Response time:** this metric refers to the time spent by the CLARUS proxy to respond to a user request. Since the operation of the CLARUS proxy is defined to be transparent for the user, the response time for any request should be minimal.
- **IP address**: source addresses of the received requests. Although during the normal operation of the CLARUS proxy, the requests from a specific user will come from a unique IP or a specific IP range, the use of dynamic IP assignment from Internet Service Providers forces the CLARUS proxy to accept requests from any IP address of the network domain.
- **Network information** : apart from the aforementioned IP addresses, the Monitoring module will also collect network flow information, bandwidth usage, protocols and other traffic statistics that will be combined with other metrics to detect DoS and DDoS attacks.

- **CPU statistics:**: mainly refers to CPU and memory usage values that will be compared with historical records to allow the detection of overloads in order to allow the Monitoring module to identify the causes.

- **Storage statistics:**: in the course of a DoS/DDoS attack, the performance of the CLARUS proxy may be degraded. In this case, the storage operations will be reduced since no legitimate activity will be performed. Thus, the storage statistics are another indicator of a possible attack in the CLARUS system.

In order to detect DoS/DDoS attacks, it is necessary to be aware of all the metrics listed above. Table 3.1 details the metric values that trigger alarms. Each row represents a warning alert or an alarm that will be triggered if any of the metrics exceeds the indicated value.

Table 3.1 – DoS/DDoS attack detection

Metric / type of alarm	**nb of request/second**	**Response time**	**nb of distinct IPs**	**Bandwidth usage**	**CPU/memory usage**	**Storage operations**
Warning	≥ 2 x threshold*	≥ 2 x average response time	≥ 2 x last 24 hours range	≥ 0,5 x bandwidth capacity	≥ 0,7 capacity	1 active request
Alarm	≥ 10 x threshold*	≥ 4 x average response time	≥ 4 x last 24 hours range	≥ 0,8 x bandwidth capacity	≥ 0,9 capacity	≥ 1 active request & storage inactive

Once a DoS/DDoS attack has been detected, depending on the seriousness of the attack (measured through the patterns defined in Table 3.1), the remediation mechanisms to be applied are as follows:

i) Warning: it means that the attack is not completed yet (the system performance is not hardly affected), but it can evolve to a serious attack if the performance continues degrading. In this case, there is no automatic action to be performed, but the warning will be sent to the administrator. If a series of this kind of alarms are received in a short period of time, it may be necessary to manually supervise the specific problem, since it can represent a fault in the system resources planning. ii) Alarm: this situation represents a serious issue that needs to be solved immediately to allow the system continue working properly with the adequate quality of service. For that, any new access will be blocked, so those legitimate users that were using the CLARUS proxy can continue accessing to it. If the low-performance situation persists, it may be necessary to implement more secure access to the platform (e.g., by using white IP lists).

After describing the components of the risk-based monitoring, we will present in the next section Montimage Monitoring tool, one of the main tools we have extended.

3.2 The Montimage Monitoring Tool (MMT)

The MMT (Montimage Monitoring Tool) is a monitoring solution that combines: data capture; filtering and storage; events extraction and statistics collection; and, traffic analysis and reporting. It provides network, application, flow and user-level visibility. Through its real-time and historical views, MMT facilitates network security and performance monitoring and operation troubleshooting. MMT's rules engine can correlate network, system and application events in order to detect operational, security and performance incidents.

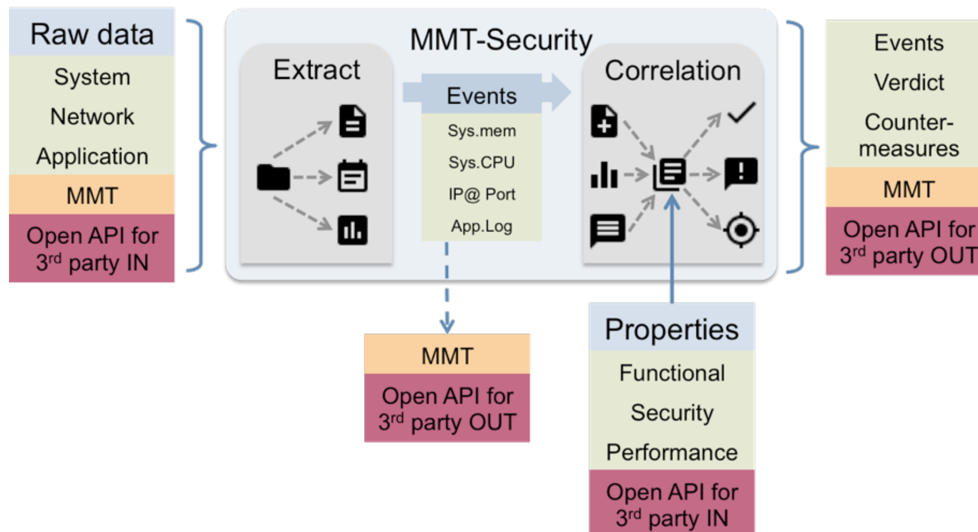

Figure 3.4 – Montimage Monitoring Tool overview

MMT-Security properties can be either “Security rules” or “Attacks” as described by the following: i) A Security rule describes the expected functional or security behaviour of the application or protocol under-test. The non-respect of the MMT-Security property indicates an abnormal behaviour. ii) An Attack describes a malicious behaviour whether it is an attack model, a vulnerability or a misbehaviour. Here, the respect of the MMT-Security property indicates the detection of an abnormal behaviour that might imply the occurrence of an attack.

3.2.1 MMT-Security architecture

MMT-Security is composed of three complementary, but independent, modules:

- **MMT-Extract** is the core packet-processing module. It is a C library that analyses network traffic using Deep Packet/Flow Inspection (DPI/DFI) techniques in order to extract hundreds of network and application based events, including: protocols field values, network and application QoS parameters and KPIs (like packet loss rate, Jitter etc.). MMT-Extract incorporates a plug-in architecture for the addition of new protocols and a public API for integrating third party probes.

- **MMT-Sec** is a security analysis engine based on MMT-Security properties. MMT-Sec analyses and correlates network and application events to detect operational and security incidents. For each occurrence of a security property, MMT-Sec allows to detect whether it was respected or violated.
- **MMT-Operator** is a visualisation application for MMT-Sec. It has not yet been implemented but will allow collecting and aggregating security incidents, and present them via a graphical user interface. MMT-Operator will be customizable: the user will be able to define new views or customize the large list of predefined ones. With its generic connector, MMT-Operator can be integrated with third party traffic probes.

3.2.2 MMT-Security properties

The MMT-Security properties are written in XML format. This has the advantage of simple and straightforward structure verification and processing by the tool. An MMT-Security properties XML file can contain as many properties as required. The file needs to begin with a <beginning> tag and end with </beginning>. Each property begins with a <property> tag and ends with </property>.

Formalism description

The MMT-Security properties are intended for formally specifying the occurrence of events that denotes a security rule to be respected or an attack or vulnerability to be avoided. They rely on LTL (Linear Temporal Logic) [Gabbay et al., 1994] and are written in XML format. This has the advantage of being a simple and straight forward structure for the verification and processing performed by the tool. In the context of this document, we use the terms of properties and rules interchangeably.

MMT refers to two types of properties “Security rules" and “Attacks" described as follows:

- A Security rule describes the expected behavior of the application or protocol under-test whether it is functional or security oriented. The non-respect of the MMT-Security property indicates an abnormal behavior, e.g. the access to a specific service must always be preceded by an authentication phase.
- An Attack describes a malicious behavior whether it is an attack model, a vulnerability or a misbehavior. Here, the respect of the MMT-Security property indicates the detection of an abnormal behavior that might indicate the occurrence of an attack, e.g. a big number of requests from the same user in a limited period of time can be considered as a behavioral attack.

The main definition of an MMT-Security property is defined as follows:

Definition 2. *Let $W \in \{$ BEFORE, AFTER $\}$, $n \in \mathbb{N}$, $t \in \mathbb{R}_{>0}$ and e_1 and e_2 two events. A MMT-Security property is an IF-THEN expression that describes constraints on network events captured in a trace $T = \{\mathsf{p}_1, ..., \mathsf{p}_m\}$. It has the following syntax:*

$$e_1 \xrightarrow{W,n,t} e_2$$

This property expresses that if the event e_1 *is satisfied (by one or several packets* p_i *, i* $\in \{1, ..., m\}$*, then event* e_2 *must be satisfied (by another set of packets* p_j *, j* $\in \{1, ..., m\}$*) before or after (depending on the W value) at most n packets and t units of time.* e_1 *is called triggering context and* e_2 *is called clause verdict.*

The nodes of the property tree are: the property node (required), operator nodes (optional) and event nodes (required). The property node is forcibly the root node and the event nodes are forcibly leaf nodes. The left branch represents the context and the right branch represents the trigger. This means that the property is found valid when the trigger is found valid; and the trigger is checked only if the context is valid. In other words:

- If the context is verified and the trigger is not, then a property non-respect instance is detected:
  - In the case of a "security rule", this means that the context of the rule has been found and, since the trigger was not, we conclude that the "security rule" has been violated.
  - In the case of an "attack", this means that the context of an attack has occurred but the trace was attack free.
- If the context and the trigger are verified, then a property respect instance is detected.
  - In the case of a "security rule", this means that the context of the rule has been found, as well as the trigger. We conclude that the "security rule" has been respected.
  - In the case of an "attack", this means that the context of an attack has occurred, as well as the trigger. We conclude that the behavioural attack has been detected, and it is necessary to trigger the appropriate remediation mechanisms depending on the attack nature.

3.3 Discussion

We presented in this chapter a methodology to perform risk-based monitoring that involves the following aspects: i) assets identification to define what is necessary to protect. ii) Threats and vulnerability analysis, to evaluate the potential flaws the system may suffer. iii) Risk analysis to categorize the threats that can exploit the system vulnerabilities. iv) System monitoring to detect potential occurrences of attacks, and. v) Remediation strategies to repel or mitigate the impact of the attacks. We also briefly presented the Montimage Monitoring Tool (MMT) that helps us to detect the attacks and implement the risk-based monitoring. This methodology has been applied in a work package of the CLARUS project and used as a basis of some of the contributions of this thesis. The risk-based monitoring methodology will support attack-tolerance by defining and executing

remediation mechanisms for each of the identified threats. This will enable the continuous operation of the system even in the presence of attacks. The next following chapters will present these remediation mechanisms. We contributed to several deliverables of the CLARUS project including D3.3, D3.4 and D3.6 so on and so forth. This project allowed us to collaborate with both academic and industrial actors. We had got the opportunity to share our respective knowledge of security in cloud computing.

4

Diversity-based attack tolerance

Contents

3.1 Risk-based monitoring methodology 44
3.1.1 Identifying Assets 45
3.1.2 Risk and vulnerability analysis 47
3.1.3 Threats Modelling 47
3.1.4 Attack scenarios 48
3.2 The Montimage Monitoring Tool (MMT) 52
3.2.1 MMT-Security architecture 52
3.2.2 MMT-Security properties 53
3.3 Discussion 54

In the previous chapters, we presented the foundations of our tolerance attack approach. We introduced the concept of risk-based monitoring. Once the risks of any system are established and the means of detection identified, it is essential to think about how to set up mechanisms that will allow to complete the risk-based monitoring loop *i.e.*, to remediate and tolerate the effects of the potential detected attacks. Moreover, in the chapter 2, we reviewed diversity, a technique of fault tolerance generally used in many systems to detect and tolerate design flaws. This method has a clear benefit since its reduces the number of missed threats. We think that diversity can be used for attack detection and tolerance. In fact, an efficient attack tolerance solution must combine and take advantage of monitoring, diversity and adaptation mechanisms in a coherent manner. Furthermore, such a solution should thwart as many attacks as possible.

In this chapter we propose the following idea that is built on diversification in a great part. At runtime, in case of an attack has been detected, we dynamically change the implementation of the running software, choosing an implementation which is more robust. This idea has been implemented through two complementary approaches. First, we will see

model-oriented diversity. This contribution is based on formal models. Then we will present the second approach, implementation-oriented diversity that reduces the shortcomings of the first approach and extends it. This second approach leverages Software Product lines (SPL) for devising a fine-grained attack tolerance system.

4.1 Model-based diversity for attack tolerance

4.1.1 Overview

Increasingly, software must dynamically adapt its behavior at run-time in response to changing conditions and also in response to attacks in the supporting computing and communication infrastructure. This first approach aims at handling these changes and ensure that the system continues to work after the adaptation process. We investigate attack-tolerance at the design and specification phase. This means that if we derive the program from a formal model, this model has been validated using, for instance model checking.

This approach is illustrated on Figure 4.16 and can be resumed as follows:

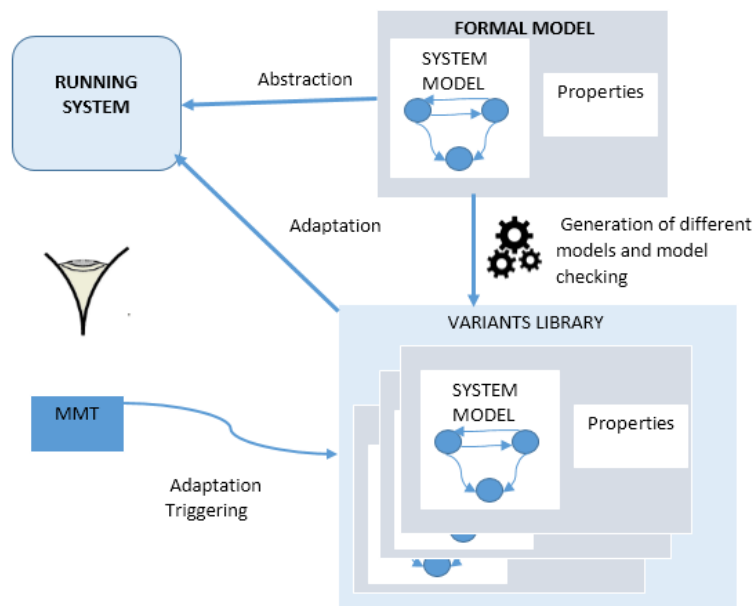

Figure 4.1 – Attack tolerance Framework

1. A running system is monitored to observe its run-time behavior with our Monitoring Tool (MMT). We design a formal model of the module that is susceptible to be suffering an attack. This model in our case is expressed as a finite state machine.

Our monitored values are abstracted and related to security properties we defined. These properties are written in linear temporal logic (LTL);

2. From this first model we will obtain others modified models that have the same functionality but can have more mechanisms to impeach attacks; these models are more secure and robust;

3. Associated to each model, we will produce an implementation. Given a model, other models are designed having the same functionality and the same behavior but different interfaces or required interactions;

4. Violations of the properties we describe above are thus detected by our monitoring tool. This detection triggers the adaptation process. We replace the model with another model that is more robust. We verify that the new model satisfies the properties. Model and implementations changes are then propagated to the running system. We finally verify that the new module is tolerant to the detected attack.

We illustrated this technique by the following example.

4.1.2 Authentication example

We start with a formal model of an authentication module. It is based on a password mechanism and we follow with other models that render the authentication more complex adding other requirements for the user. In our example, after an authentication mechanism based in a password checking, we propose authentication models with more complex mechanisms.

Model 1 (Figure 4.2). In this model the mechanism of authentication is based only on a login and password. This is generally called one factor authentication. This process of authentication, even though is generally considered the least secure, some Web sites such as e-commerce platforms continue to deploy it.

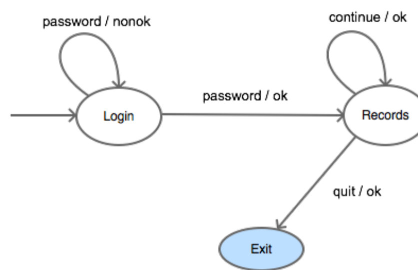

Figure 4.2 – Authentication Model 1

Model 2 (Figure 4.3). After, another authentication model is proposed in which we make the following assumption. A threshold of 3 attempts is defined as a security requirement. If

a user attempt is equal to the threshold, the user, by an adaptation mechanism, is required to fulfil other requirements (questions).

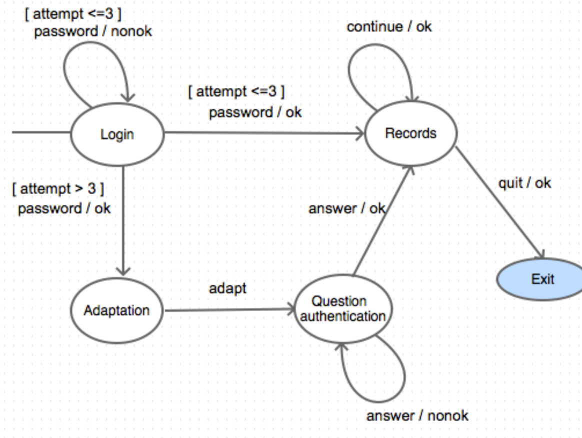

Figure 4.3 – Authentication Model 2

Model 3 (Figure 4.4). This model is based on the deployment of a multi-factor authentication mechanism. This enhances the security by adding another authentication process. This can be achieved by:

- SMS: A SMS that contains an authentication code is sent to the user through its smartphone. If the code entered corresponds to the sent code, then the user is authenticated.
- USB Smart Card: This is a new way of authentication. The authentication's keys of the user are stored on this USB device. After logging in with his login and password, the user plugs his authentication USB key. This confirms the access of the user to the system.
- Vocal message: As above, the user can get a vocal code for the second authentication step.

For this model we choose the first option for the sake of simplicity.

Model 4 (Figure 4.5). In this model we design an authentication model that requires multiple authentication steps. This will allow us to delay considerably the spy in its implementation of the attack. This model is thus based on the following mechanisms:

- Keep the system in an abnormal mode;
- Require then a 4 authentication processes: captcha, random challenge questions, calculus, and password again. This is a kind of moving forward approach;
- Return to the normal mode after the indirections;
- Adding delay: We add a delay to a normal authentication process.

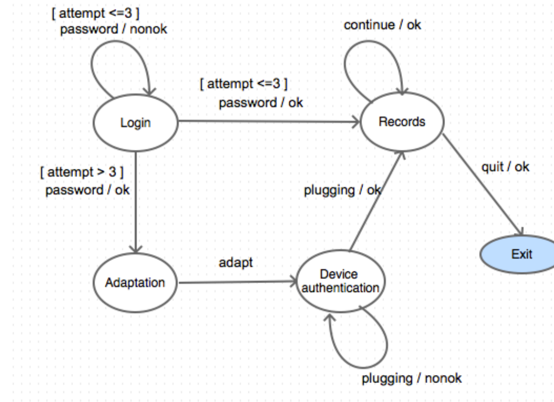

Figure 4.4 – Authentication Model 3

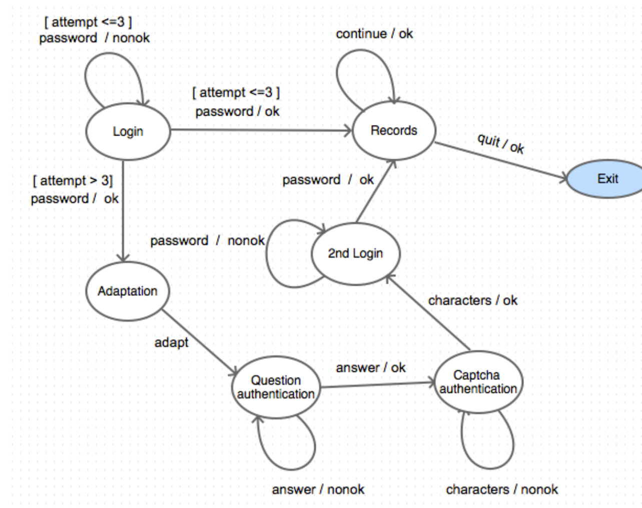

Figure 4.5 – Authentication Model 4

Note that for this model we use a more complicated model that integrates in addition to login/password, a private question authentication, a captcha authentication and a second login/ password step.

To summarize this approach, it can be said that it is a combination of monitoring methods and the generation of functionally equivalents models that are more robust to resist to an attack.

One can argue why we do not choose the more secure model first. We provide the following answer to respond to this assertion. One claim is that the first model can also detects attacks because we have a policy that says: after more than 3 unsuccessful attempts the account of the user is locked for security purposes. The first model is also secured. The main reason for not choosing the last model first, is the user quality of experience (QoE). Quality of Experience (QoE) is a measure of the overall level of customer satisfaction with

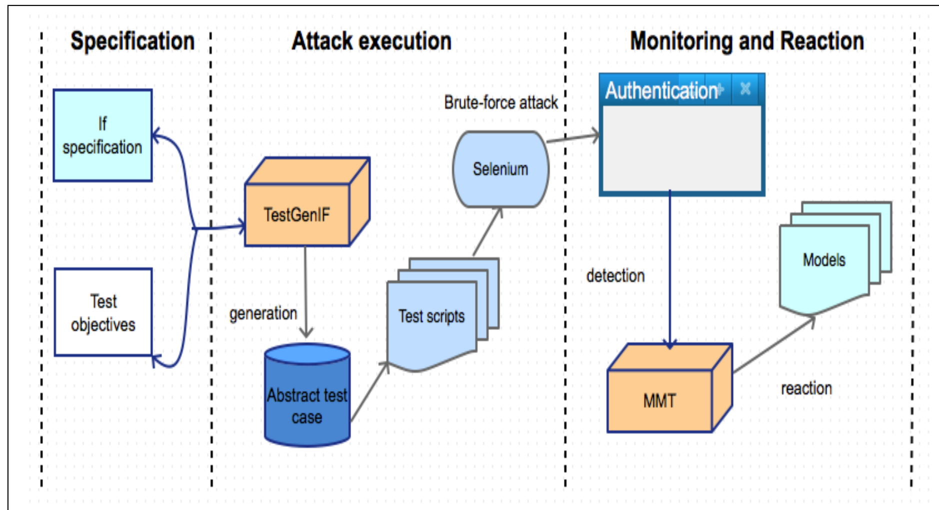

Figure 4.6 – Experimentation Framework

a solution[1]. QoE expresses user satisfaction both objectively and subjectively. To go back to our example, imagine a legitimate user that forgot his credentials and succeed to log on at the third attempt. On the first model he can access simply to its information and use the application without a waiting time. In the last model, for example, he will be bored answering to questions or doing the captcha image recognition and so one. The challenge would be to find a reasonable compromise between performance, Quality of Experience (QoE) and security. The choice can be leveraged depending on the type of application and consequently on the security level needed.

4.1.3 Experimentations

In this section, we provide a description of the tools or languages we use to implement our approach.

IF language

IF [Bozga et al., 2002] is a formal method based on communicating timed automata in order to model asynchronous communicating real-time systems. In IF, a system is expressed by a set of parallel processes communicating asynchronously through a set of buffers. An IF process is described as a timed automaton extended with discrete data variables. A process has a set of control states and a private buffer for input messages, and can have local data such as discrete variables and clocks. There are two types of control states: stable states and unstable states.

[1] http://searchcrm.techtarget.com/definition/Quality-of-Experience

TestGen-IF Tool

The TestGen-IF tool is based on the IF language simulator [Bozga et al., 2002]. It uses the IF simulator libraries which provides some functionalities for on-the-fly state-space traversal. TestGen-IF implements a timed test case generation algorithm [Hwang et al., 2009]. An IF specification document (.if file) and timed test purposes documents (i.e., inputs directory), and the jump depth value (i.e., maxdepth parameter), are given as inputs to TestGen-IF. During the test generation, when a timed test purpose is satisfied, a Hit message, the test purpose description and the number of timed test purposes to be satisfied are displayed.

Test Bench

The experiments were performed on the authentication Web application described in the previous sections. This application represents an authentication mechanism that can present different degrees of complexity in order to impeach the attacks. An authentication interface is available. A user can access its information after providing the correct username and correct password. We chose a Model Based Testing approach. Model-based testing (MBT) is an activity in which one can design and derive test cases from an abstract and high-level model of the system under test (SUT). On the basis of abstract models, test cases can be derived in the form of test suites. These test suites are not directly executable because they do not have the same level of abstraction as the executable code. This often requires manual intervention by a test engineer who has to design an adaptation layer to convert the abstract tests to the corresponding executable tests. Figure 4.6 depicts our framework we deployed for our tests. Our Framework can be divided into three parts: the specification part, the attack execution part and the monitoring and reaction part. This Framework works as follows. We begin with the specification part which is conducted through the process described in the sequel.

1. We translated the state machines of the authentication example we presented in the section 4.1.2 using the IF language [Bozga et al., 2002]. We chose the IF formalism for the following basic reasons. On the one hand, our models being built as a state machine, the IF model is therefore suitable to represent them. On the other hand, we wanted to be able to test our models and have at our disposal TestGen-IF a model-based testing framework with the IF language as input. This tool allows us to have more accurate and quasi exhaustive test cases of our models. We give an illustration of this formalization for the first model (cf. Figure 4.2) in the following Figure:

   This small piece of IF specification describes two states of the finite state machine. The first state, named *login*, corresponds to the entry point of the automaton. The *dologin(loginx, passx)* statement is a function that provides the inputs of our states; in our case, the login and the password of the user. In IF, this function is called a signal. After giving these two inputs, the system moves to the next state named *credent*. This state is an internal state. We choose to define this state as an internal state

```
state login #start ;
   input dologin(loginx, passx);
     nextstate credent;
endstate;

state credent #unstable ;
  provided((loginx<>login1)or(passx<>pass1));
    output nonok();
    nextstate login;
  provided((loginx=login1)and(passx=pass1));
    output ok();
    nextstate records;
endstate;
```

Figure 4.7 – IF specification of Model 1

because we want to express a conditional loop depending on the correctness of the credentials of the user. The attribute *#unstable* means that the state credent is an unstable state i.e. a temporary state where no interleaving between different processes is possible. The statement *provided((loginx <> login1) or (passx <> pass1))* and the statement *provided((loginx=login1) and (passx=pass1))* express the guards of our transition with the keyword *provided*. In the finite state model *provided* above it corresponds to the guards between brackets. If the login and password given as inputs correspond to the correct login and password of the user, the user can access to his information (the next state is *records*). Otherwise, he must type again his login and password (the next state is *login*).

2. As usual in a model-based testing approach and in order to use TestGen-IF tool, we need to specify clearly our test objectives. A test purpose or test objective describes a particular functionality of the implementation under test, by specifying the property to be checked in the system implementation. It is an observable action of the system that once described in IF language is used for guiding the space exploration of the system's states. One suite of these test objectives is depicted in Figure 4.8.

   This suite of test objectives is called *OBJ(1)* and contains two test objectives (*obj1* and *obj2*). Each test objective is a conjunction of conditions (*condi*) that must be satisfied in order to satisfy the test objective. *obj1* is made of five conditions. *cond1* verifies that the process is created. *cond2* verifies that the source state is "login". *cond3* verifies that the input is *dologin()* with its two parameters. *cond4* verifies that the output is *ok*. *cond5* verifies that the source state is *records*. On contrary, *obj2* expresses, that when the parameters of the input *dologin* are wrong, the *nonok* signal is sent and as a consequence the system is going back to the initial state.

After the specification phase, we describe the attack execution part of our Framework. The

```
OBJ(1)={obj1, obj2}
obj1= cond1  AND  cond2  AND  cond3  AND  cond4  AND  cond5
obj2= cond1  AND  cond2  AND  cond3  AND  cond6  AND  cond7
cond1= process : instance= log0
cond2= state : source : login
cond3= action : input dologin(loginx, passx)
cond4= action : output ok()
cond5= state : destination : records
cond6= action : output nonok()
cond7= state : destination : login
```

Figure 4.8 – Test objectives

tests objectives and the IF specification described at the specification phase are given as inputs to TestGen-IF. TestGen-IF launches its processing engine and thus generates test cases accordingly. The test case is a sequence of inputs and outputs. The question mark (*?*) denotes an input while the exclamation mark (*!*) denotes an output. Then, the line *?dologin{login1,pass1} !ok* means if the user discloses the correct credentials (login1 and pass1), the output must be ok. The line *?cont !ok* and *?quit !ok* represent the case where, if the user want to continue to access (respectively quit the system) to his data, the output should be ok (respectively nonok).

```
?dologin{login1,pass1} !ok{}
?dologin{login1,pass2} !nonok{}
?cont{} !ok{}
?quit{} !ok{}
?question{quest1,answ1} !ok{}
?question{quest1,answ2} !nonok{}
?characters{char1} !ok{}
?characters{char2} !nonok{}
```

Figure 4.9 – Generated test cases

In order to execute the test on our running system, we need a concretization step that transform these abstract tests into real test cases. We use the selenium[2] tool for this purpose. Selenium is a Framework that automates Web application testing. The abstract test cases generated have been translated into a set of http requests. These requests are based on the recording of the action made by the user with Selenium IDE, one component of our Selenium tool-suite, on the Web application. We then match the abstract test cases

[2] http://www.seleniumhq.org

to the corresponding concrete JUnit[3] test cases in a script we developed. The tests cases allowed us to test the implementations of the authentication example. This script will be used later to inject a brute-force attack on our implementations.

Finally we describe the monitoring and reaction part of the Framework. We examined all HTTP packets transiting between the client and the Web application. We collect the packets directly through the eth0 network interface rather than using a sample of packets obtained from network protocol analyzer tools like Wireshark[4]. In our Framework, a set of security properties representing this attack behavior has been specified as an input for MMT tool. We describe the following rule in order to detect the brute force attack. When analyzing the packets we record all HTTP request targeting a specific URI of our Web application (for example, the authentication page URI). From this record, we get all IP addresses source and count their occurrence in the set of packets. We analyse the traces using the MMT tool in order to check if attacks have been successful. The are two cases:

1. If we find out for example that any such IP address that has requested more than 3 times to the URI, we raise an alarm and block this IP address. We also leveraged the metrics defined in the previous chapter. We apply our methodology by adapting the current model and its implementation with one of our models we designed previously (see Figure 4.10)
2. Otherwise we continue our analysis of the network traffic.

Using this Framework, we conduct two different but complementary experiments for the validation of our model-based approach. We will present in detail these experiments in the next section.

Experiment methodologies

In order to prove the practicability and ability of our approach to tolerate attacks, we implemented the models we presented previously. We describe two different but complementary scenarios:

1. First we launched the brute-force attack to verify that the Framework ensures detection and remediation of the attack by changing the model and its implementation. This attack is an attempt to discover a password by systematically trying every possible combination of letters, numbers, and symbols in order to discover the correct combination that works. We use a dictionary of common most used passwords (over 8864 passwords here because the use of a very big dictionary takes a couple of days in order to obtain the correct password). This tests were performed on an Ubuntu System with Intel Core i7 CPU and 8 GB of memory.
2. After, we conducted another experimentation that consisted in launching the same brute-force attack against our three implementations. The idea is to check the

[3]http://junit.org/junit4/
[4]https://www.wireshark.org

robustness of our models and their implementations against this attack. For all models and implementations, we continuously look for the password of the user in order to access to the user's sensitive information page and measure the time needed to access to this information. We launched the attack and measured the time for recovering the password of a particular user. These experiments were executed on a MacBook Pro with Intel Core i5 CPU and 8 GB of memory.

We offer two choices. Let us first assume that we have found the password and try to access private user data. We then measure the time taken to get to this stage. This process is termed scenario 1.
In the second choice, we instead launch the attack until we found the password knowing that at each set of three unsuccessful attempts, the user is blocked and we need to restart the application to try another attempt. This process is called scenario 2.

Results

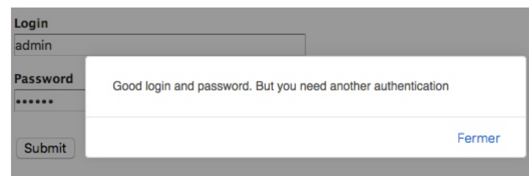

Figure 4.10 – The authentication showing the adaptation GUI

We analyse the results of both scenarios. In scenario 1, according to the detection policies we express, we observe that Montimage Monitoring Tool (MMT) successfully detects the attack in a relatively short time and with a high detection rate. As a result, an alert is delivered to the user (see Figure 4.10). Alongside with the detection, as a defense mechanism, our model based-adaptation is launched. When the alert for the detected attack is launched, the adaptation process consists in pointing directly to the implementation of the next robust model through its URL. The user must finally authenticate himself trough another authentication process. All these steps we described above are transparent to the user. We aimed at providing an attack-tolerance Framework that do not disturb the activity of the user.
The results of scenario 2 are presented in Table 5.14(a).

From this table, we observe that the average time needed to access to our sensitive data grows slowly from model 1 to model 2, but grows considerably from model 2 to model 3. This means that the third authentication is more robust than the second one which is more robust than the first implementation for this attack. This was predictable since the degree of complexity and security increases with the models. We must notice that for this

Table 4.1 – Scenario 2 measurements

Implementations	**Mean time (s)**
Implementation 1	3 384
Implementation 2	3 689
Implementation 3	21 736

example we only used roughly a 8000 words dictionary. Then, the time to execute these tests shall be more long if a full dictionary was used. Thus, an attacker could not succeed anyway to access to the user data with the third implementation.

4.1.4 Discussion

We proposed model-based diversity for attack tolerance. We begin by creating a formal model of our system or a part of this system. We derive a library of equivalent models that achieve the same functionality as the first one and are verified correct. We derive also the corresponding implementations of the models. When the MMT tool detects the attack, we dynamically change the model, choosing a model and its implementation that are more robust to the attack. However, the principal difficulty of this method lies in building the models. There are some questions that remain to solve in order to enhance model-based intrusion tolerance and make it more practical and easy to use by engineers. A difficulty is to know how many models do we need to generate. To answer to this question we can argue that it depends on the level of security that the engineer wants to ensure for its system. Another question is how can we synthesize these variants? We can overcome this problem by automatically deriving variants of the initial formal model by developing new operators similar to mutation operators. In the field of software testing, program mutation, consists in using mutants, that are syntactic change(for example && replaced with || in a condition) in the specification of the software in order to detect the misbehavior. If a test suite is able to detect the change, then the mutant is said to be killed. However, in our case here, these "mutants" can be used for diversifying the models i.e. diversifying the corresponding state machines. If we do not want or if we are not able to apply this method, another way of avoiding this question is to have a single model and implementations that differ at the language, source code and binaries levels. This is what we are going to see in the next section.

4.2 Implementation-based diversity for attack tolerance

This contribution aims at extending and solving the issues raised by the former approach. The idea is still the same but here we will only have one model and several implementations

4.2.1 Definition of key concepts

Throughout this section we have a number of terms that we need to define in order to precise how they will be used later: **Definition 1:** *Variability is a concept that makes it possible to express a characteristic or attribute that is not common for the members of a certain set. In our case, the variability will be described in such a way that the different variants are equivalent from a functional point of view (not necessarily the same operations).*

Definition 2: *A feature model (FM): is usually used in the area of Software Product Lines as a graphical representation of the differences and similarities between members of a given family. These points of differentiation are called variability points.*

Definition 3: *A variant or spare of a Web service is another Web service that performs the same operations, in other words functionally equivalent to the first one, but differs at the implementation level.*

Definition 4: *An epoch here denotes a given constant time interval. Time is thus divided into slices of period equal to an epoch.*

4.2.2 Overview of the approach

The architecture of the proposed framework is depicted on the Figure 4.11.

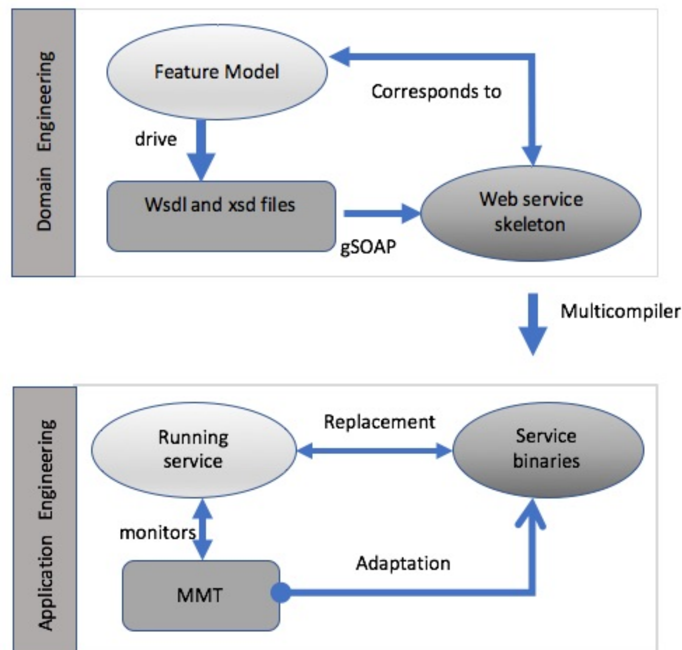

Figure 4.11 – Architecture of the attack tolerance Framework

We have developed our approach through an example. The example consists in a Web service for the management of physicians and patients in a hospital. All information are

stored in a database. Only the doctors have access to the platform and the information its contains. In particular, they may monitor the state of health of a patient under his supervision from the patient admission in the hospital until the end of treatment. For better clarity, we subdivided the process into three distinct part: a modeling part, a service generation part and finally a monitoring part.

Part 1: Modeling: Here we modeled the e-health service. We begin by defining the feature model (FM) of the e-health service (cf. Figure 4.12). The variability points are

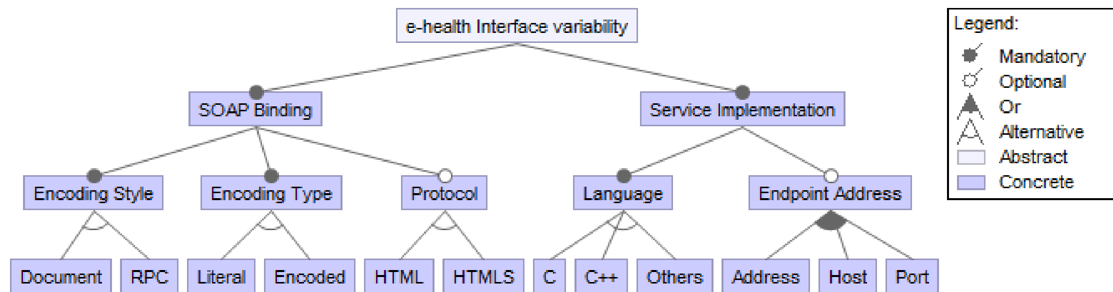

Figure 4.12 – Feature Model of the e-health service

described using three common patterns: style, encoding styles and types and language:

- *Encoding style*: There are two communication style that are used to translate a WSDL binding to a SOAP message body. They are: document and RPC. The structure of the SOAP request body with an RPC style contains both the operation name and the set of method parameters. This operation can return results. A document-style SOAP message body contains a specific XML structure that can be validated against a previously defined XML schema.

- *Encoding type*. There are two encoding models used to translate a WSDL binding to a SOAP message. They are: literal, and encoded. In the literal model, the data do not need to be encoded in a special way: they are directly encoded in XML according to a schema defined in the WSDL. With an encoded model, the message has to use XSD datatypes, but the structure of the message do not need to conform to any user-defined XML schema.

- *Language*: Our Web services spares will be developed with C and C++ languages.

The combination *document/encoded* as it does not enable interoperability is not used in our implementation. These variability patterns are set manually when specifying the WSDL files and the variation is done with respect to the FM. We define four different WSDL files for our Web services. A sample of the WSDL file is disclosed in the following Figure 4.13.

We see on this figure, the declarations of the types, and the operations that will be carried out in our Web service. The small example means that we will have the connection message of a doctor accessing the Web service whose connection parameters are *credentials*.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="med-webservice"
        ........

        <types>
         .......
        </types>

        <message name="connectDoctorRequest">
                <part name="credentials" element="ns0:Login"/>
        </message>
         ......
        <portType name="healthPortType">
                <operation name="connectDoctor">
                        <input message="tns:connectDoctorRequest"/>
                        <output message="tns:connectDoctorResponse"/>
                        <fault name="connectDoctorFault"
                            message="tns:connectDoctorFault"/>
                </operation>
                        ......
        </portType>
  </definitions>
```

Figure 4.13 – WSDL Sample

We see the definition of the SOAP port through which the response requests of the connection operation will pass. We also have the XSD file in which we defined the complex types *DoctorType* and *PatientType* (Figure 4.14).

To add even more randomness, the compilation of these different source codes will be done with the multicompiler tool we will present in the following sections.

Part 2: Service generation: We construct variants of the implementations of our Web services and used them in our attack tolerant framework. With the *.wsdl* and *.xsd* files of the previous phase, we generate skeletons of our variants thanks to *gSOAP*. First, an header file containing the declarations is generated. We generate then the corresponding implementations skeletons based on this header file. Figure 4.15 presents the server side skeleton.

We see on the listing the header files (*health.nsmap* and *soapH.h*) generated automatically by gSOAP as well as the file *mysql.h* header that helps us to connect to the database. We also have the main function that initializes the soap endpoint. Finally, we have the empty function *__ns1__creationPatient* which aims at adding a new patient in the hospital. We implement these skeletons and adapt them to fit our needs.

After the implementation of these skeletons, we have four *ready-to-use* implementations.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://xml.netbeans.org/schema/medical"
            xmlns:tns="http://xml.netbeans.org/schema/medical"
            elementFormDefault="qualified">

<xsd:complexType name="PatientType">
        <xsd:sequence>
            <xsd:element name="full-name" type="xsd:string" minOccurs="0"
                maxOccurs="1" />
            <xsd:element name="address" type="xsd:string" />
            <xsd:element name="age" type="xsd:int" />
            <xsd:element name="disease" type="xsd:string" />
            <xsd:element name="admission-date" type="xsd:date" />
            <xsd:element name="exit-date" type="xsd:date" />
            <xsd:element name="doctor" type="tns:DoctorType" />
        </xsd:sequence>
</xsd:complexType>
```

Figure 4.14 – XSD Sample

```c
#include "health.nsmap"
#include "soapH.h"

int main() {
    struct soap *soap = soap_new1(SOAP_XML_INDENT);
    if (soap_serve(soap) != SOAP_OK)
        soap_print_fault(soap, stderr);
    soap_destroy(soap); // delete deserialized objects
    soap_end(soap);     // delete allocated (deserialized) data
    soap_free(soap);    // free the soap struct context data
}

int __ns1__creationPatient(struct soap *soap, struct ns2__PatientType*
                                        ns2__Patient, char* *result) {
     *result = (char*) soap_malloc(soap, 1024);
      .........
      .......
}
```

Figure 4.15 – Server side skeleton

According to our case study, we have diversity based on 2 target languages (C and C ++), two communication styles (*rpc*, *document*), two types of encoding (*encoded* or *literal*), two syntactic modifications (i.e., order of instructions, unnecessary instructions). We will have at least 15 variants. (document/encoded combination not permitted).

Even though attacks against Web services are usually unrelated to the source code of the service, we must be very careful. In fact, the knowledge of the engine used to generate the skeletons can still leak information. For example, assume that an attacker succeeded to discover that the Web services are implemented in C through the gSOAP tool, it is easy for this attacker to infer source code-based attacks, such as buffer/heap overflow, memory load or even malicious code injection (shell code). This is why, we will pass the implementations to the multicompiler [Franz, 2010] tool for diversifying the binaries. This ensures that our implementations will not be vulnerable to the same attacks leveraging the computation flow (code reuse attack). The tools multicompiler, gSOAP and MMT will be fully described in the next section.

In addition, multicompiler has many options to generate different binaries. The relevant options are: i) nop-insertion that inserts NOP operations in the assembly; ii) nop-insertion-percentage similar to the previous one, but here inserts NOP operation before X% of instructions; and iii) frandom-seed that randomises the compilation with a 64-bit unsigned int. We can choose 3 seeds for example 25, 50, 75. Finally we could have for example at least 75 different variants. The main design of the solution are depicted on Figure 4.16.

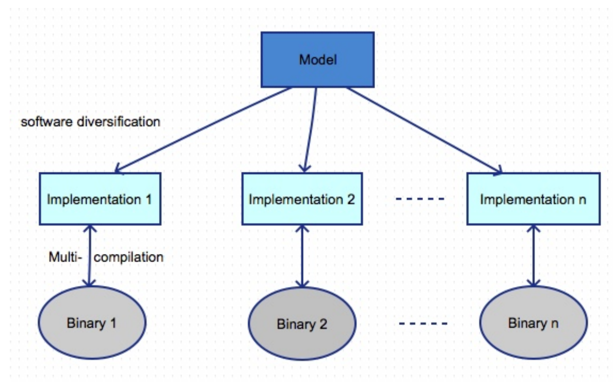

Figure 4.16 – Attack tolerance through diversity

This picture shows the first diversification which consists in having syntactical equivalent implementations and their binaries obtained with multicompiler. In conclusion we have three layers of diversity. That increases the confidence in the Web service.

Part 3: Monitoring and reaction: This is where the logic of attack tolerance lies. We design an attack tolerant library of the implementations. The idea is to formally verify these variants in order to limit the failures leveraging bugs. We launch functional tests on the implementations to ensure that they work normally. Before the system will be put online, a learning and testing phase is performed. Target attacks are also applied to these implementations. We then determine the implementations vulnerable or not to the target attacks. They will be listed in a vulnerability class. To ensure the continuous availability of our system we plan to configure our system in two ways (cf. Algorithm 1):

1. **Normal mode**. We divide time into *epochs*. In each *epoch* only a unique variant is chosen at all. When the *epoch* of time elapses, another implementation is deployed to ensure continuity of the service. This helps us to thwart to attacks that our system may not have detected. It is the case of a silent but dangerous attack: Slowloris[5]. Slowloris works by opening multiple connections to the targeted Web server and keeping them open as long as possible. It continuously sends partial HTTP requests, that are never completed. The attacked servers open more and more connections, waiting for each of the attack requests to be completed. Undetected or unmitigated, Slowloris attacks can also last for long periods of time. The recovery will then mitigate this attack as the server is restarted.

2. **Abnormal mode**. It is the case where our defense mechanism has successfully detected one attack. We will obviously react by switching to another more resistant implementation before even the *epoch* has elapsed.

Algorithm 1 Web service adaptation

```
1:  while True do
2:      epoch ← setEpoch()
3:      if date == null then
4:          date ← currentDate()
5:      end if
6:      if detected then
7:          chosenVariant ← random(1, n_variants)
8:      else
9:          current ← currentDate()
10:         if (current − date) ≥ epoch then
11:             chosenVariant ← random(1, n_variants)
12:             date ← current
13:         end if
14:     end if
15: end while
```

One can notice that the duration of the *epoch* is a parameter that must be taken into account in order to have an efficient tolerance to the targets. Indeed a small *epoch* induces too many commutations between the variants and a long *epoch* can miss a lot of attacks. We want to have *epochs* that minimize the latency of the users when they are accessing the Web service.

Moreover, even conceptually our framework is robust, on the case an attack succeeds, we will enable another learning phase. The aim of this phase is to adapt an attack database or training data in order to predict or detect this kind of attack if it appears later. We will

[5] https://github.com/llaera/slowloris.pl

leverage event logs and we will record copy of the Web traffic and traces. To reduce the size of these files, we will fix a maximum size. When new data are being inserted on the files, if the maximum size is reached, we will remove the oldest entries. Furthermore the content will be periodically updated.

4.2.3 Experiments and discussion

In this section, we provide a short description of the tools we use to implement our approach.

Multicompiler tool

The multicompiler tool [Franz, 2010] has been developed by Franz et al. in order to defend against code reuse attacks. Multicompiler is built on top of the standard LLVM project[6]. From the source code of any software, the multicompiler tool generates several different binaries. The diversity engine use some transformations such address space randomization, NOP operations insertions. As a result a number of code variants are produced, which ensures the system will be more resistant to attacks. The idea is to increase the cost of the attacker activity, implementing programs with equivalent functionality that can respond to attacks in a more safer way. The implementation of this approach could be facilitated by the use of cloud computing. This makes it possible to scale almost instantaneously to even very large peak demands without any up-front investment. Furthermore, by using cloud computing, the cost per unique version of a program is essentially constant, it doesn't matter if we are generating 1000 or 10 Millions versions per day, being also possible to change the demands almost instantaneously.

gSOAP Tool

The gSOAP toolkit [Engelen and Gallivan, 2002] is a C and C++ software development toolkit for SOAP and REST XML developed in order to make the development of Web services as platform-neutral as possible. gSOAP provides transparent SOAP API through the use of compiler technology that hides irrelevant SOAP-specific details from the user. There are two main components. *wsdl2h*, the first one, analyzes WSDLs and XML schemas (separately or as a combined set) and maps the XML schema types C and C++ code header file (*.h*). The generic string and float primitive types are encoded and decoded in SOAP as standardized XML schema types (i.e. xsi:type="xsd:string" and xsi:type="xsd:float"). *soapcpp2*, the second one, maps SOAP/REST XML messaging protocols to efficient C and C++ code source skeleton using the previous generated header file. It supports exposing (legacy) C and C++ applications as XML Web services by auto-generating XML serialization code and WSDL specifications. As a result, full SOAP interoperability is achieved with a simple user-loaded SOAP load of SOAP details and allowing it to focus on application-critical logic.

[6] http://llvm.org

Experiments

The evaluations were performed using the e-health Web service. We obtained 18 variants that are obtained with *rpc/document* styles, *literal/encoded encoding*, multicompiler (25, 50, 75 random seeds or NOP operation insertion), instructions obfuscation and C/C++ languages. The Web services variants has been compiled with the multicompiler's clang and clang++ compilers and deployed as CGI programs. Web services spares were compiled with multicompiler's clang and clang++ compilers and deployed as CGI programs on an apache Web server, to facilitate our recovery mechanism. We only need to adapt the running implementation without update the endpoint of the service.

In conclusion, we performed four different but complementary experiments. First, we performed a training experiment in which the detection *threshold* value i.e, the average number of packets by second during a normal activity; and also the best *epoch* value have been evaluated. We analysed as well, the overhead in term of latency of the framework. We tested the scalability and performance of e-health Web Service with and without our strategy. Tests were performed by invoking remote methods 100 times and during different times a day. After, we evaluated the capability and accuracy of our framework to detect a Denial of Service (Dos) attack.

The innovative aspect of our detection lies in the fact that we have widened the detection spectrum of the MMT tool by incorporating new detections of these attacks.

Training experiments

We generated traffic corresponding to a normal use of our Web services at different times of the day. Table 4.2 discloses the results of the execution. Accordingly to the experiments,

Table 4.2 – Threshold measurement

Min	71051 packets/sec
Max	87766 packets/sec
Average	81966 packets/sec

the threshold will be 81966 packets/sec. To obtain the *epoch* time of the system, we have followed the methodology proposed in [Sousa et al., 2007b]. In our case, we redefine T_D as the effective switchover time between our implementations and T_P is equivalent to the *epoch* we defined previously. Consequently, we have $n=18$, $f=5$, $k=1$, and *epoch* $>= 108 \times T_D$. As the average T_D is *2.53 seconds*, finally the value of *epoch* is *273.24* seconds (roughly *5 min*).

Table 5.14(a) discloses the results of the measurement.

Table 4.3 – T_D measurements

Min	2,30 seconds
Max	2.77 seconds
Average	2.53 seconds

Latency analysis

We also examine the overhead of the attack tolerance framework by carrying out a latency analysis of the system before and after deploying our methodology. The latency here is the round-robin duration of a client.

To examine the overhead of the attack tolerance framework, We carried out a latency analysis of the system before and after deploying our methodology. The latency here is the round-robin duration of a client. The results are depicted on Figure 4.17.

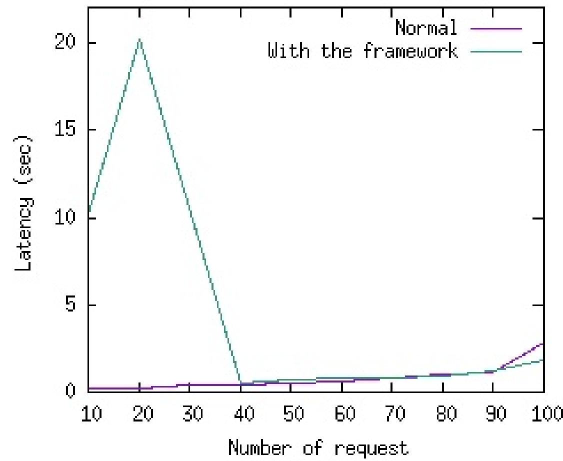

Figure 4.17 – Latency with and without our framework

It can be seen in this Figure that the graph representing the latency before the approach is strictly increasing. The graph representing the launch of the approach is not monotonous. In fact, between 10 and 40 requests, the graph has a peak and there is a stabilization from 50 to 100 requests. This peak is due to the initial recovery overhead. After 50 requests the two graphs are almost similar. On the contrary, the similarity between these two graphics after 50 requests is due to several reasons. The first reason is that we have statically compiled the variants. The latency then doesn't include compilation overheads. Secondly we believe that this similarity is due to the principle of temporal locality in which processors tend to reuse the data and instructions used in the recent past. Moreover, the choice of CGI as the deployment allows us to reduce the switching time between the variants. All variants are called from the same endpoint.

Using very large numbers of requests (cf. Figure 4.18), one sees that the trend is the

same.

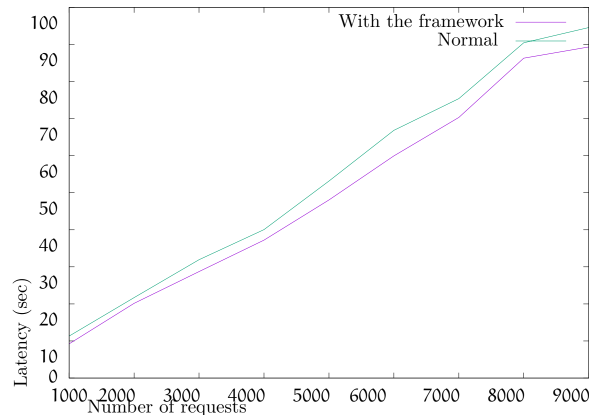

Figure 4.18 – Latency with and without our framework

The overhead is almost insignificant. In real life, Web services receive thousands of requests per second. Since the overhead of our approach is insignificant for very large customer requests, our framework is suitable for real applications.

Dos Attack

As we said in the previous section, to tolerate attacks, either we detect them by monitoring, or we anticipate the occurrence of an attack. We validate the variants by testing them with the SoapUI[7] framework. XML-bomb attack was launched against the services variants. We found that our variants are not vulnerable to XML-based attacks. Consequently we will test our Web service with a DoS (Denial of Service) attack, which can be defined as an explicit attempt by attackers to prevent legitimate users of a service from using that service. The detection of this attack is enabled by MMT. With MMT, to detect an attack, a rule describes the patterns that if they are or not verified help to conclude the presence or absence of the attack. We analyze and record all HTTP requests targeting our endpoint in order to check if the attacks have been successful. The are two cases: (i) If in a period of one second the metrics (chapter 3) of the traffic are greater than the corresponding thresholds, then we may have an attack.(ii) Otherwise, there is no attack. We launched several times the DoS attack against our Web service.

The DoS attack were launched against our Web service using LOIC (Low Orbit Ion Cannon) From Table 4.4 and 4.5, the framework seems highly reactive. In fact, the time to detect a DoS attack is roughly 2.5 seconds. After the detection, we switch to another implementation of our web service by changing the endpoint of the service in the CGI folder. We also counter the effects of the attack for example by closing the open ports and restarting the server. Since web services may not have the same diversity parameters, there should be some sort of wrapper that connects the customer to the right service.

[7]https://www.soapui.org

Table 4.4 – Time elapsed to detect dos attacks

Dos Attack	Detection time
1	2.504751 seconds
2	2.397282 seconds
3	2.426627 seconds
4	2.397596 seconds
5	2.424721 seconds
6	3.060523 seconds
7	2.931496 seconds
8	2.455809 seconds
9	2.571891 seconds
10	2.920244 seconds

Table 4.5 – Time elapsed to complete a client request in the presence of a DDoS attack

Nb. of packets/s	Before (seconds)	After (seconds)
1000	4.45	5.93
2000	5.25	6.91
3000	8.09	9.05
4000	8.34	9.51
5000	9.72	11

4.3 Discussion

We claimed that the problem of attack tolerance of Web services and more particularly the services stored in the cloud is an open challenge. The conventional techniques alone can not be enough to protect Web services against increasingly intrusive attacks. It is not enough to only detect known or new intrusions; system need to react against attacks and repel them or at least mitigate their effects, while keeping the system running. In this chapter, we leveraged diversity in order to enable attack tolerance with two approaches. We began by model-based attack tolerance approach in which there is one model and several equivalent generated models from that main model. Then we presented, implementation-oriented diversity that reduces the shortcomings of the first approach and extends it. During all the steps of the construction of our Web services, i.e., from modeling up to the concrete instantiation, we have integrated diversification. This allows us to ensure end-to-end security at all levels of our services. We have shown based on our preliminary work on a simplified version of the CLARUS project use case, that our approach tolerates certain types of attacks with a relatively low latency. This confirms that diversification is effective for attack tolerance of Web services. Nevertheless, diversity is not the unique approach for achieving attack tolerance. In the next chapter, we will see another attack tolerance technique. The results of this chapter have been published in the proceedings of the International Conference on Advanced Information Networking and Applications

Workshops (WAINA 2017) [Ouffoué et al., 2017], in the proceedings of the peer-reviewed International Conference on Web Services (ICWS 2017) [Ouffoué et al., 2017a] and in the peer-reviewed International Conference on Services Computing (SCC 2017) [Ouffoué et al., 2017b].

5

Software reflection based attack tolerance

Contents

4.1 Model-based diversity for attack tolerance 58
4.1.1 Overview 58
4.1.2 Authentication example 59
4.1.3 Experimentations 62
4.1.4 Discussion 68
4.2 Implementation-based diversity for attack tolerance 68
4.2.1 Definition of key concepts 69
4.2.2 Overview of the approach 69
4.2.3 Experiments and discussion 75
4.3 Discussion 79

The risk-based monitoring approach requires effective detection and response techniques. This is particularly true for so-called internal attacks which are difficult to detect attacks. This chapter presents a methodology using software reflection to prevent, detect, and mitigate internal attacks to a running Internet Web services. We think this methodology is very suitable to design such systems as secure by default, that is, when designing the software some parts are marked as secured, and any change/modification of these parts will be an unexpected behavior that needs to be analyzed. If these changes turn out to be attacks, then some remediation techniques are activated, in order to guarantee that the system will continue to work even in the presence of an attack. In addition of providing the methodology, we show how this technique has been used as the basis to develop a real information system. The main sub-contributions are then as follows:

- We briefly reviewed the literature and explored the scientific work and techniques about reflection.

- We propose and describe a new approach exploring the usage of Software Reflection as a mean of detection and mitigation of insider attacks.
- We describe the approach through a RESTfull e-health Web service.
- We evaluate it with this Web service the e-health Web service using a realistic testbed. We explain how the attacks are detected and how we use our methodology.

5.1 Background

Software reflection is a way of implementing meta programming techniques (programs manipulating themselves) and was envisaged to enable the construction of programs that require the ability to examine or modify their execution behaviors. The foundations of software reflection were first proposed by [Maes, 1987] that brought some additional features of reflection and described an original experiment to introduce and show how reflective architecture can be incorporated in object-oriented languages such as Python [Generowicz et al., 2004], C++ [Roiser and Mato, 2005], JAVA [Forman et al., 2004], etc. In practice, there are two types of activities in reflection: introspection and intercession. Introspection is called read-only reflection in the sense that a component learns the metadata (Classes, methods, attributes, types) of itself but do not change them. Intercession, on the other hand, is called read and write reflection in the sense that the component can look up the metadata of itself or another component and may change them. Software reflection is nowadays implemented in most of the existing programming language (Python, Java...). The developers of Java provided a rich reflection API. Reflection can allow applications to perform operations that otherwise would be impossible, for example, access to some protected fields. This reflection API is used usually for introspection in the context of Java beans which is a sort of component model in Java. Introspection enables a Java bean to get the properties, methods and events of other beans at runtime. This helps the developers to design and develop their beans without knowing the details of other beans.

At the beginning, reflection techniques were used for adaptability [Affonso and Nakagawa, 2013], debugging, self-optimization, integrity verification [Spinellis, 2000], Remote Method Invocation (Java RMI),... However, in this chapter we present how reflection can be used as a secure by design technique, having as a consequence to improve the *security* of an application and enable attack tolerance for insider attacks. Insider intrusions on a company are the most difficult to detect because they require extensive means of defense. Insider attackers are whether, (i) traitors who are legitimate users that sometimes misuse the access privileges given to them, or (ii) masqueraders who steal identities of some legitimate users [Salem et al., 2008]. Insider attacks can affect the proper functionality of a program or corrupt the data used by the programs. Insider intrusions on a company being the most difficult to detect require extensive means of defence. The survey [Gheyas and Abdallah, 2016] presents some existing approaches proposing insider threat detection and prediction algorithms (IDPA) that aim at detecting such attacks.

Furthermore [Sun, 2016] proposed a way of detecting untrusted access to software source code. Their analysis was carried out on access logs to the SVN repository. Firstly, the log data was parsed and stored into a Log database. Further, leveraging the data obtained, they performed Cluster analysis based on the VAT algorithm [Bezdek and Hathaway, 2002]. Data clustering is one of the methods used in machine learning to analyse data. They assumed that the data overwhelmingly represent normal behaviour and any deviation of this normal behaviour is considered as an abnormal behaviour. Besides, [Lin, 2016] explained the lacks of current commercial products addressing malware and insider threat attacks. He claimed that role-based access control policies are useless in preventing policy abuse attacks. He then proposed to store the user's authentication status in an Active Directory log and to establish a profiling model to capture the normal behaviour. A deviation of this normal behaviors triggers an alarm.

Although all of these methods are interesting from a theoretical point of view, we think they are not sufficient to properly detect internal attacks and malware due to the following reasons: a) In all existing machine learning methods, detection involves a learning step in which the normal behaviour is described. However, in an internal attack, the usurper's behavior often has very little difference compared to the behavior of legitimate one. This can lead to several false positives in the final detection. b) None of these approaches offer remediation sketches in order to allow applications to continue functioning even in the presence of insider attacks. A new approach to insider attack detection and tolerance is needed. In this chapter, we rather propose to design a generic attack-tolerant methodology for insider attacks. Our approach integrates the reflection techniques mentioned above as well as the monitoring of the log files.

5.2 Framework

In any methodology, it is important to explicitly define the assumptions one makes. So we introduce the assumptions and the definition of basic concepts that will be used in our methodology.

1. We consider that the software of the client is located in a safe environment for example in a DMZ (Demilitarized Zone). For this reason no attacker can access the trusted zone (the case of CLARUS).

2. Some potential attacks that can take place are *internal ones*. That is, coming from internal spies. The aim of the attacker is to usurp the actions i.e. to modify the methods of the platform API.

3. Even if the environment is safe, we also assume that the unsuspecting use of employees (e.g. the unknowing click of an email attachment) can lead to malware exposures. A particular case of viruses are ramsonwares for example [Cert, 2017], an attack that had affected hundreds of thousands of computers worldwide in the first half of 2017,

paralyzing some public services and businesses. Here we only consider some class of malware. Our taxonomy of malware is depicted in the following table 5.1 [Sans, 2017].

Table 5.1 – Taxonomy of malware

Attack	Description
Virus	A virus spreads itself by infiltrating its code into an application. The virus is generally not limited to its spread, which makes the host software unusable, but also launches malicious routines.
Worm	Computer worms are similar to viruses in that they replicate functional copies of themselves. In contrast to viruses, which require the spreading of an infected host file, worms are standalone software.
Trojan	A Trojan is a form of malware disguised as useful software. Its purpose: to be executed by the user, which allows him to control the computer and to use it for his own purposes.
Spy	Spyware does what its name says: it is a spy that collects various data about the user without the latter realizing it.

Definition 3. *An attack is any external or internal interaction with the system that modifies the behavior or changes some parts of the code making them unsafe. An attack can also be the presence of malware that has the purpose of spreading and infecting the machines of the system.*

Definition 4. *An API is a set of methods and tools that can be used for building software applications and we can consider them as safe.*

Definition 5. *Given an API of our system, any change of this API can be considered as an internal attack.*

As presented previously we define a layered framework depicted in Figure 5.1:

- **Layer 1: Web Firewall Service.** This is the entry-point of the framework.
- **Layer 2: Specialized operations.** This layer contains the business operations (for example patients management for an e-health software) of the running application.
- **Layer 3: Authentication.** We will have a multi-factor authentication namely a user-password authentication followed by an SMS authentication. This multi-factor authentication obviously increases the security since the attacker needs much more time to access to the system.

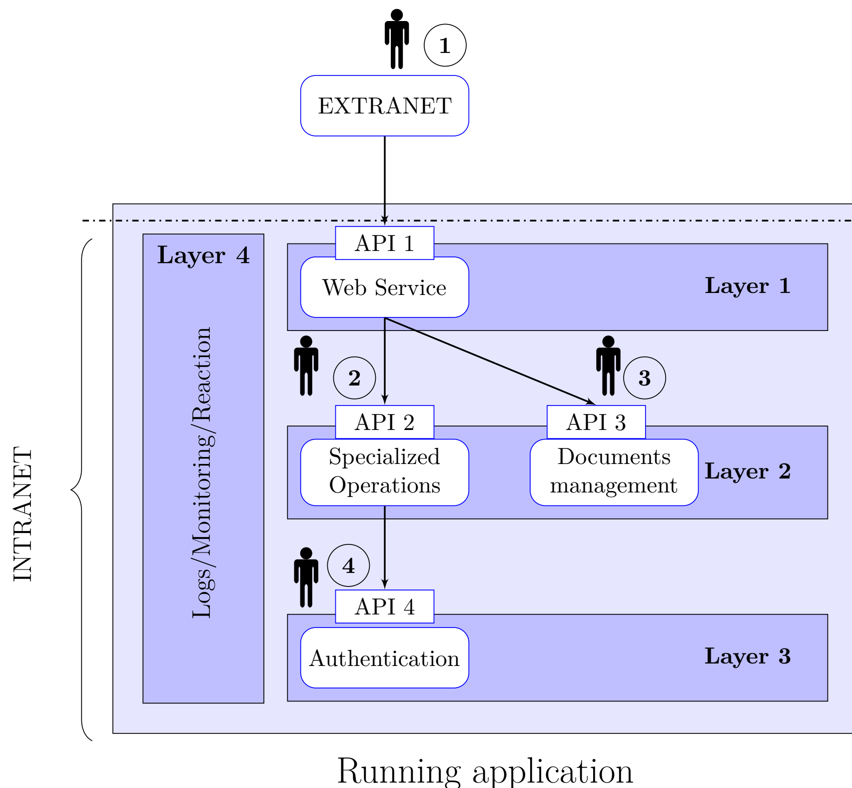

Figure 5.1 – Attack tolerance framework

- **Layer 4. Logs, monitoring and reaction.** This layer spans the other layers presented above. It is responsible of the detection of misbehaviors and the reaction against such threats in each layer of the running application. The detection of attacks is based on the monitoring of the log files. Each operation or method invocation is stored in a log located in each part of the system (application, server and cloud). The monitoring tool analyses these logs in order to find misbehaviors or attacks according to some security rules. When the logs or the network probes testify the presence of an attack or a misbehavior, a classification is made for deploying the best countermeasures or remediation to these threats. The classification leverages a vulnerability DB, where the hashes codes of known vulnerabilities are stored.

Any software component (Web Service, Specialized Operations, Documents Managements and Authentication) has it own API. The whole framework aims at providing attack tolerance by design for any user of the platform. For example, lets us consider the user 1 who wants to use the Web service. If the Web Service does not work because of an attack, the user can still access to the documents using the extranet. In the same manner, the users 2 and 3 can still perform some specific operations (Layer 2) even if some operations

of Layer 2 stop working. But any component needs the authentication component (Layer3). Also, let us note that user 4 wants to authenticate himself and that the authentication component is not available due to an attack, user 4 is not able to access the database or critical parts of the running application.

Once the assumptions and the general framework described, a detailed description of the detection and reaction methodology is presented (Layer 4 above). We propose a methodology that will ensure an efficient attack tolerance. To better tolerate attacks, first they must be detected. Consequently, detection is an important part of our approach.

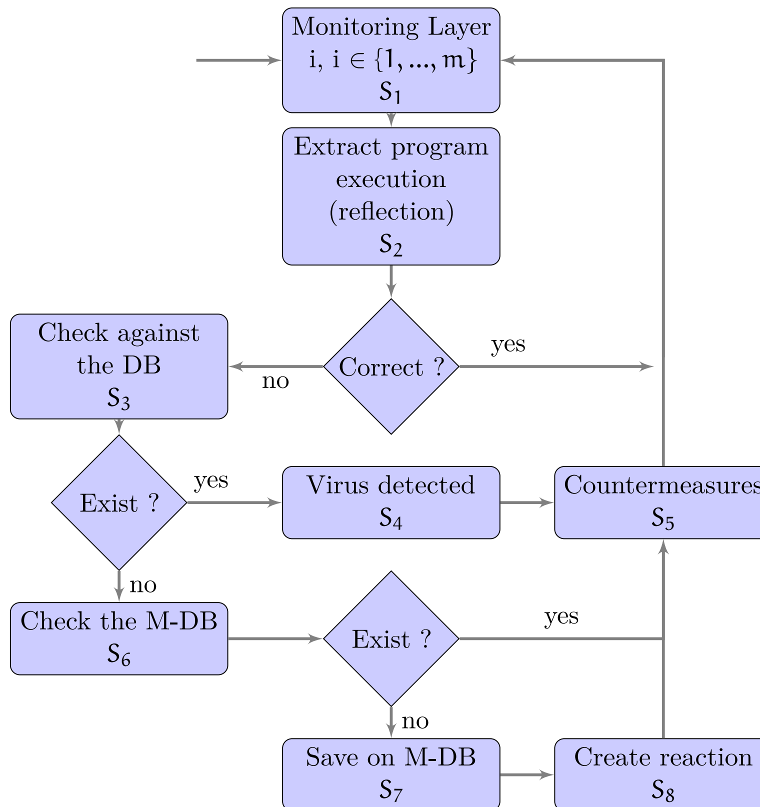

Figure 5.2 – Detection and Reaction model

The whole detection and reaction framework works as follows:

- **Monitoring and Detection (States S_1 and S_2)** : We begin with the detection step. Monitoring and detection are possible thanks to the tool MMT. The programs are checked at runtime using reflection. The system can be in the normal working conditions, i.e., the security policies are respected, in this case we have nothing to do. Or on the other hand, if something abnormal is found, there may be a virus attack, an API modification or an unknown attack.

- **Check against the database of virus (State S_3):** We analyse the hashed fingerprints of the program and compare that value in the database of attacks. If the attack appears in the vulnerability database, it means that the attack is a virus (State S_4) or an attack resulting by the modification of a method of the API. We provide a set of countermeasures (State S_5). If the attack is a malware, the system launches the corresponding patch. If the threat is a modification of one of the methods of the API, a countermeasure can be to replace a software component/layer or to change to the initial API method. These countermeasures will be described later.

- **Check against the M-DB (State S_6):** If the attack that happens is not known in the vulnerability DB, we check in our own DB, called Montimage DB (M-DB). If the attack exists, we provide the same countermeasures as mentioned above.

- **Save on M-DB (State S_7):** This case corresponds to the situation in which the attack has no hash either in the vulnerability DB nor in the M-DB. The hash is then stored in the M-DB and we define a new countermeasure (State S_8).

We can also have security level. Following, we define the security level of the attack according to the criticality levels as well as the reaction we are going to provide. When an attack occurs in one of the methods of a given API, the countermeasure corresponding to the criticality level of this API is triggered. The new corresponding reaction is returned to mitigate the effect of the attack (State S_5).

According to the Security Level (SL) we have: 1) SL Normal means a correct functioning of the system. The events in the logs are noncritical. Reaction: Nothing to do. 2) SL Warning means that these events indicate that a component is not in an ideal state and that other actions may cause a critical error. Reaction: We stimulate the layer or component in order to check its response to some predefined inputs. According to the responses obtained, we can locate and correct the misbehavior. 3) SL Attack means: The events indicate that a component/layer of the system has been affected and that the component/layer failed or stopped responding due to an attack. The component/layer and the attack are then identified. Reaction: The layer or component is automatically disconnected from the network and the other components/layers. It will be quarantined and we will remove the malicious code. Thus, the removal is a very effective mechanism for avoiding the production of more viruses. This is to prevent the virus from spreading to other unaffected layers. We replace the affected component/ layer with new ones. 4) SL Critical Attack means: The Events demand the immediate attention of the system administrator. They are generally directed at the global level. The events indicate that one or several components or layers of the system have been affected at the same time due to an attack. This is also the case where any critical part(databases, storage) of the system has been accessed. Reaction: We react like in the case above. In the case a critical part like the storage has been destroyed, there is a recovery step in which the data are restored with backup data. The security levels mentioned above will be incorporated in the future extension of this work.

5.3 Case studies

5.3.1 Overview

We propose an e-health Web application example that is a software for the management of patients and doctors of an hospital. In the field of health, data are ultra sensitive and critical. An incorrect modification can lead to the admission of inappropriate treatment that can lead to premature death of the patient. On the other hand, any leakage of information relating to the state of health of a patient, for example an AIDS patient may result in the patient being excluded from society. This is the main reason why, we chose the field of health to illustrate our approach. The simplified API consists of 4 methods, that are presented in Figure 5.3.

HealthOperation
`getConnected`(login, password) : token `listPatient`() : boolean
`createPatient`() : boolean `updatePatient`(idPatient, token) : boolean

Figure 5.3 – API of the `HealthOperation` Center

It is assumed that this API is by definition *safe*, i.e., it is only accessible by authorised people. Let's suppose the following code is used as the implementation of the `updatePatient` method (Figure 5.4):

```
def updatePatient(self, idPatient, token):
  '''
  This function updates the patient.
  Only authorized users, can do this operation.
  '''
  log=checkSource();
  if self.validateToken(token):
  with open('patients.txt') as f:
    for line in f:
      if  idPatient in line:
          ...
      logging.info("updatePatient "+log)
      break
  log=checkSource();
  return result
```

Figure 5.4 – Correct implementation of *updatePatient*.

Let us note that both: a) In this scenario, as protection measure, a user wanting to perform an action must first obtain a token provided by the super-administrator. b) This

token should be validated before updating the database.

Leveraging this fact, let us suppose that the malicious attacker extends the class by redefining the *updatePatient* method as follows (Figure 5.5):

```
def updatePatient(self, idPatient, token):
  '''
  This function updates the patient.
  There is no validation of the token.
  '''
  with open('patients.txt') as f:
    for line in f:
      if  idPatient in line:
  ...
      break
```

Figure 5.5 – Unexpected implementation of *updatePatient* function.

These lines of code are functionally similar to the first implementation of the *updatePatient* method but the length of the code is not the same. The insider attacker got a token from one of his colleagues who has more privileges than him. There is no verification of that token. The attack can, for example, modify the API and insert fake values into the system or can retrieve confidential data. In this way, the requests of the users of the application do not return correct results. An attack can also be the presence of a malware that has the purpose of spreading and infecting the machines of the system.

Following our methodology, this attack is detected by using software reflection. This is a Meta programming technique. It is possible in many programming languages to be able to dynamically get the code and even the execution trace of a method, class, module. One can also modify the class at runtime. In Python the *inspect* module provides functions for learning about live objects, including modules, classes, instances, functions, and methods. Functions can be used in this module to retrieve the original source code for a function, look at the arguments of a method on the stack, and extract the sort of information useful for producing library documentation for your source code.

We suppose that the API is the one described previously. For detecting attacks, we will use logs located on two endpoints: on premises, on the server (proxy). We then will store them in addition to the general information such as the date and time, the hash of the stack of any running code. These hashes are made possible by the method *checkSource()* defined in Figure 5.7.

Consequently, any line in all logs has the following format:

Date Hour Operation hash . We also consider that, every request made by a user using the API is followed by an answer from the server. Any request has then two traces in the logs: *outbound* and *inbound*. For instance, if the user sends a *getConnected* request, it will produce the corresponding *getConnected outbound* in the log file of the application. When the server responds to that request, it will produce *getConnected inbound* in the log file of the application. Consider now the case of the *updatePatient()* method that interests

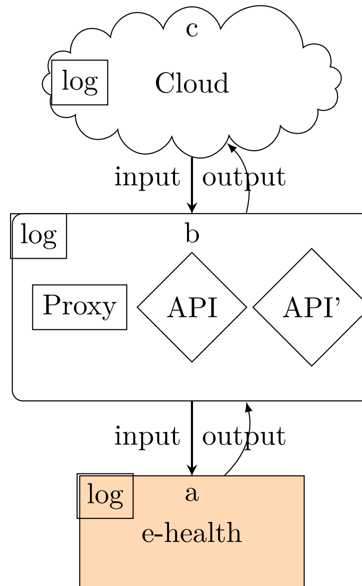

Figure 5.6 – Setup of the running system.

```
def checkSource():
  a = inspect.getsource(inspect.stack()\
    [1][0])
  m = hashlib.md5()
  m.update(a)
  return format(hashlib.sha224(a)\
    .hexdigest())
```

Figure 5.7 – Implementation of the *checkSource* method.

us in particular. Let's detail how the client can use that method in a correct way and also in cases where the attack is manifest. As we showed in the first implementation of *updatePatient* method (Figure 5.4), the *checkSource()* method is called before and after the operation itself. We suppose that the hash of the *outbound* operation corresponds to the first *checkSource()* call while the *inbound* operation one's corresponds to the second call of *checkSource()*. Both hashes should obviously match because they are obtained from the same method. The log file of the application will be:

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	! updatePatient(Outbound)	2224d35250e...	a
07/11/2017	10:15:00 AM	? updatePatient(Inbound)	2224d35250e...	b

Reciprocally upon receiving the *updatePatient* request, *updatePatient inbound* is written

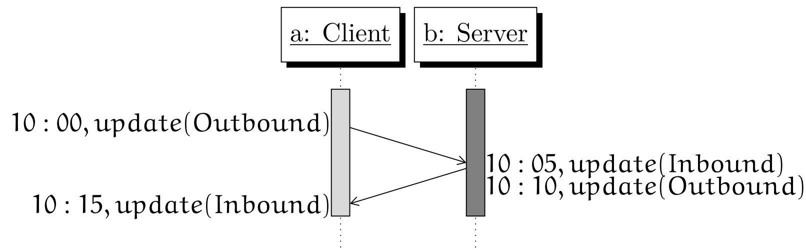

Figure 5.8 – Normal case

in the log file located on the server. When the server responds to that request, it will produce an *updatePatient outbound* in the server's log file (Figure 5.8). The server's log will be:

Date	Hour	Methode	Hash	Host
07/11/2017	10:05:00 AM	?updatePatient(Inbound)	2224d35250e......	b
07/11/2017	10:10:00 AM	!updatePatient(outbound)	2224d35250e......	a

Let's see the case where an attacker has succeeded to launch his attack against the API. We describe then the situation when an attack occurs. Any of these cases seem to show that there is an attack: someone has modified the API and overridden one or several methods:

- No hash: This is the case where we see some information of the methods but there is no Hash. This happens when the attack overrides the method but do not implement it correctly (Figure 5.5).

  Let us note that on the client side we have (note that the server side is the same):

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	!updatePatient(Outbound)		a
07/11/2017	10:15:00 AM	?updatePatient(Inbound)		b

  It is also possible to get only one hash for the *outbound* operation and nothing for the *inbound* operation.

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	!updatePatient(Outbound)	2224d35250e...	a
07/11/2017	10:15:00 AM	?updatePatient(Inbound)		b

- Hashes not equal: Here we consider that we got some relevant information in the logs but the hashes of both *Outbound* and *Inbound are not the same*. This can appear on the client side, the server side or both. This happens when the attacker uses the same core algorithm used on the correct code but the instructions or the implementation of the rest of the code is not the same.

  We have on the client side:

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	!updatePatient(Outbound)	2224d35250e...	a
07/11/2017	10:15:00 AM	?updatePatient(Inbound)	2504d35222e...	b

  On the server side:

Date	Hour	Methode	Hash	Host
07/11/2017	10:05:00 AM	?updatePatient(Inbound)	0c251145317...	b
07/11/2017	10:10:00 AM	!updatePatient(outbound)	53171140c25...	a

- Inconsistency: Here we can get some inconsistencies in the logs. For instance on the client log (respectively server log) we got an *Inbound* (respectively *Outbound*) operation before an *Outbound* (respectively *Inbound*).

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	?updatePatient(Inbound)	0c251145317...	b
07/11/2017	10:15:00 AM	!updatePatient(Outbound)	0c251145317...	a

  On the server side:

Date	Hour	Methode	Hash	Host
07/11/2017	10:05:00 AM	!updatePatient(Outbound)	0c251145317...	b
07/11/2017	10:10:00 AM	?updatePatient(Inbound)	0c251145317...	a

  - **Operations are not ordered:**

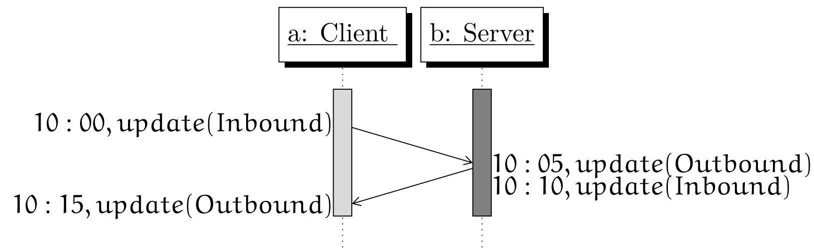

Figure 5.9 – Inconsistent messages diagrams.

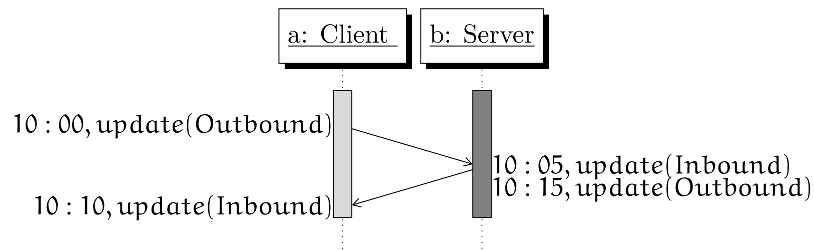

Figure 5.10 – Inconsistent messages diagrams.

- **Timestamps are not ordered:**

- We can also have the inconsistency with the dates or hours in the log files. (Above Figure 5.10) We show on the client side:

Date	Hour	Methode	Hash	Host
07/11/2017	10:00:00 AM	!updatePatient(Outbound)	0c251145317...	b
07/11/2017	10:10:00 AM	?updatePatient(Inbound)	0c251145317...	a

On the server side :

Date	Hour	Methode	Hash	Host
07/11/2017	10:05:00 AM	?updatePatient(Inbound)	0c251145317...	b
07/11/2017	10:15:00 AM	!updatePatient(Outbound)	0c251145317...	a

The inconsistency lies in the fact that the client receive *updatePatient(Inbound)* before the server sent *updatePatient(Inbound)*.

Let us remark that we could have any combination of the inconsistencies mentioned above.

5.3.2 Detection and mitigation

For detecting these attacks, regarding the methodology presented in section 5.2, we applied the following security policies (rules). On the client application:

- **Rule 1**: Any update request should have hashes for its operations (outbound and inbound) on the log files and these hashes must correspond.
- **Rule 2**: Any outbound operation should be followed by an inbound operation.
- **Rule 3**: If the outbound and the inbound operations have the same hashes, the inbound one shouldn't appear before the outbound one.

The rules on the server side are the same as those of the client seen above. On the server side:

- **Rule 4**: Any update request should have hashes for its operations (outbound and inbound) on the log files and these hashes must correspond.
- **Rule 5**: Any inbound operation should be followed by an outbound operation.
- **Rule 6**: If the outbound and the inbound operations have the same hashes, the outbound one shouldn't appear before the inbound one. Aggregating the two logs we will have these news rules:
- **Rule 7**: For any outbound operation that appears in the log of the application, there must be a corresponding inbound operation in the log of the server coming from the application and the outbound (from the application) operation must occurred (clock indication) before the inbound operation(from the server).
- **Rule 8**: For any inbound operation that appears in the log of the server, there must be a corresponding outbound operation in the log of the application whose source is the server; and the outbound (from the server) operation must occurred (clock indication) before the inbound operation (from the application).

In the case of attack against the API, by comparing the hashes of both outbound and inbound operations, one can see that a lambda instruction has been called without being the right one. Thus the attack is detected. In the same way we can detect the external attacker. For a good and efficient monitoring of and how to specify those rules, we will use the MMT tool and investigate how we can enhance the *Complex Event Processing* engine. The central concept in the CEP field is therefore the event. It must be noted that the events that we take into account within MMT-Security properties are related to observable system/network communications. In the case of a telecommunication network, they refer to traffic packets and flows. In other contexts, they can relate to any action that can be stored in a server/database/software log file. MMT runs on a modular architecture. Any additional features or protocol is incorporated by a development of a specific plugin. So

Code Listing 5.1 – Example of MMT's specification of Rule 1

```
<beginning>
<property
        value="THEN"
        delay_units="s"
        delay_min="0+"
        delay_max="2"
        property_id="1"
        type_property="ATTACK"
        description="Detection of the insider attack">

        <event
                value="COMPUTE"
                event_id="1"
                description="reception of the outbound operation in the log"
                boolean_expression="((log.op == 1)
                        &amp;&amp; ((#strcmp(log.op, updatePatient(outbound)) != 0)
                        &amp;&amp; (log.hash!= '' ))"/>

        <event
                value="COMPUTE"
                event_id="2"
                description="reception of the inbound operation in the log"
                boolean_expression="(((log.op== 1)
                        &amp;&amp; (#strcmp(log.op, updatePatient(outbound)) == 0)
                        &amp;&amp; (log.hash!=''))
                        &amp;&amp; (#strcmp(log.hash, log.hash.1) != 0))"/>
</property>
</beginning>
```

Where: Event 1 (e_1): reception of the outbound operation (log.op) in the log with a hash (log.hash!= "), and Event 2 (e_2): reception of the inbound operation in the log and we compare the hash with the previous hash (the built-in C function *strcmp* was used for the comparison).

Figure 5.11 – Security Rule representation in MMT.

we have developed a plugin that allows us to parse the available information on log files so that MMT can infer rules from the information contained in these files.

We then specified the rules describe above. For example Rule 1: Any update request should have hashes for its operations (outbound and inbound) on the log files and these hashes must correspond, as depicted in Figure 5.11. This XML document expresses the rule 1 in the formalism of the MMT tool. This is an attack detection rule (type_property="ATTACK"). In this example, if the context and the trigger are verified, then an attack/evasion has been detected. Note that *&* is equivalent to logical *AND* and *strcmp* is the classical string comparison function in C.

In summary, we proposed some rules for the efficient detection of an insider attack using MMT. After the detection, we must react in order to ensure the attack tolerance capability of the framework. The mitigation and the remediation techniques are as follows. To mitigate an insider attack, there are three ways of reacting. a) The first way is to dynamically change the implementation of the class at runtime. In this way any further attacks leveraging that issue will be thwarted. This is called *Behavioral Reflection* i.e. reification of execution. b) The second way is to disable the overridden function. This is called *Structural Reflection* i.e. reification of structure. c) Finally we can also change the class at runtime according to a period of time to increase the randomness. We will

illustrate the first method in the next section through experimentation.

These are made possible by using metaclass and reflection in python. A metaclass is defined as "the class of a class". Any class whose instances are themselves classes, is a metaclass. Reflection is the process by which a program can observe and modify its own structure and behavior at runtime. Let's show how we can change a class at runtime in the following example [Dougblack, 2017].

```
import types

'''Overrides the implementation of the method updatePatient '''
def newUpdatePatient(idPatient, token):
   ......
    print ' update completed '
  return True;

class MetaClass(type):
  def __new__(cls, name, bases, attrs):
    for name, value in attrs.items():
      if type(value) is types.MethodType:
        if attack=True:
          attrs[name] = newUpdatePatient(value)
    return super(MetaClass, cls).__new__(name, bases, attrs)

class HealthOperation(object):
  __metaclass__ = MetaClass

  def updatePatient(self, idPatient, token):
      ....
      return True;
```

Figure 5.12 – Example of usage of metaclasses

We assume that the main class is the *HealthOperation* and we consider that if an attack has been suspected, we change the implementation of the *updatePatient* method with the new one *newUpdatePatient* at runtime (Figure 5.12). In this snippet, we create a metaclass named MetaClass. Our metaclass parses all the methods and attributes of the class *HealthOperation* and when the attack has been detected (by evaluating the attack variable for example), the implementation of the *updatePatient* method is dynamically changed at runtime by the new method *newUpdatePatient*. This example is naive but explains how we can use reflection.

5.4 Experiments and results

To implement and test our approach, we developed a python RESTful Web-service with the FLASK [Ronacher, 2017] framework. This Web service implements and extends the example of the hospital seen previously. The preference of REST above SOAP is obviously because REST is easier to use and is more flexible. It has the following advantages over SOAP as it is fast, effective, and there is no need of expensive tools to interact with the Web service.

The service has two main databases, a database of viruses (Virus DB) and a database for MMT(MMT DB) which contains the meta-data of the methods (name, module, source code). To be sure that these databases can not be corrupted, standard database protection techniques have been used. These databases have been encrypted and the data they contain as well. For the whole framework, the methodology is the one explained in the previous section. The operations related to both the patients and the doctors (creation, list, update, deletion) were implemented as REST requests (POST, GET, PUT, DELETE).

Figure 5.13 – RESTful API of the e-health Center.

The use-case conducted is the following. We assume that one or more operations of the service have been compromised by injecting the code with a known virus. This may be the case, for example, if one of the project partners has clicked on a malicious link received in an e-mail. The requests launched by the different users are intercepted and analyzed with the detection tool (MMT) before executing the corresponding method. In this step, there is a comparison between the hash code of the methods invoked and the hash code we have in MMT DB. If the hash is equivalent to the hash of the method that is in the DB, it means that nothing malicious happens. If the hash does not correspond to the hash of the method that is in the DB, an alarm is issued and as a countermeasure, the safe code of the operation existing in the DB is dynamically executed so that the corrupted code can not spread. This also ensures continuity of the service for the users. By doing so, all subsequent attempts by the attackers will not succeed.

We evaluated the framework in the presence of virus samples of VirusShare [1]. Experiments have been launched. We showed below these experiments. In the first experiment we investigated the overhead(overall time needed to respond to the requests of the clients) generated by the framework when an attack is detected. The second experiment aims at comparing the accuracy (time to detect an attack) of our detection tool with the classical detection commercial off-the-shelf (COTS) tools.

Experiment 1: We measured the average time to make a client request without attack and when the attack is detected. The results are recorded in the following table 5.14(a). We find that these values are very close. It can be concluded that the approach does not induce much overhead and that this is transparent to the user.

Experiment 2: In this part, the ability of the framework to detect viruses attack in comparison to a conventional anti-virus. A virus was injected into the Web Service. For security and simplicity issues, a new virus has been proposed. This virus modifies all the codes of the classes, methods or functions of the python modules of a given directory tree. We established a signature of the new virus. We added this signature to our virus database as well as that of the anti-virus ClamAv [2] a well known anti-virus for all operating systems. We used a logical signature (a logical signature allows combining of multiple signatures in extended format using logical operators) as well as a hash-based signature(namely md5). The sample of the logical signature is the following.

sample;Engine:0-20,Target:0;((3&4)|(0|1|2|3|4));
66696c65746f696e666563740a;696e6665637465640a;
66696c656c6973740a;69662054727565650a;73797356d0a746

Our local database is based on the site' virusshare database. This site contains all the hashes (md5) of known malware. For fairness we only launched ClamAv in the folder containing our web service implementation. We run the virus and try to detect it with both antivirus. The results in terms of detection time are recorded in the table 5.14(b). From the table 5.14(b) we can conclude that our framework is twice faster than ClamAv.

Without detection	**With detection**
0,0545 seconds	0,0553 seconds

((a)) Latency measurements

Our framework	**ClamAv**
5,534 seconds	12,870 seconds

((b)) Detection time

Figure 5.14 – Experiments 1 & 2 results

Moreover, we think our framework is more suitable than the conventional anti-viruses for the following reasons. MD5-based anti-malware only works against static-infections that

[1] https://virusshare.com
[2] https://www.clamav.net

never change. However, there are also polymorphic malwares that change continuously their source code. So whether it's static signatures or dynamic signatures, attackers can still do zero-days attack. But with our methodology, any attack that will take place will necessarily be detected because we base our detection on the sources of our modules and not on the sources of viruses. This is fundamental for attack tolerance. The only condition is to make sure that this database can not be easily compromised.

5.5 Discussion

In this chapter we have presented a generic secure methodology to detect and remediate insider attacks on Web-based services. Our approach is based on reflection and monitoring techniques. We have also presented a multi-layer architecture and show how we are able to detect at runtime changes of the kernel application that might be considered as attacks. In addition, to be able to detect attacks our methodology allows to enable/disable different layers of the system in order to stop the attack and allow the system to continue working. Our goal is to have always a part of the system always running, even in the presence of an attack and be able to provide some ”critical” services in any situation. As conclusion, software reflection is suitable for attack tolerance. So, in the next chapter, we will see how we can leverage reflection as well as the other methods of attack tolerance presented previously to build a more complete attack tolerance framework. The results of this chapter have been published in the proceedings of the peer-reviewed International Conference on Web Services (ICWS 2018) [Cavalli et al., 2018].

6

An attack tolerance framework for Web-based applications in the cloud.

Contents

5.1 Background 82
5.2 Framework 83
5.3 Case studies 88
5.3.1 Overview 88
5.3.2 Detection and mitigation 94
5.4 Experiments and results 97
5.5 Discussion 99

In the previous chapters, as part of the implementation of our risk-based monitoring methodology, we presented different approaches of attack tolerance. These approaches were based on the concepts of diversity and software reflection. In addition, we shown that Web services are increasingly being used to deploy applications in the cloud. We think that it would be interesting on the one hand, to find a way to consider and express these applications to allow them to benefit optimally from our attack tolerance methods. On the other hand we believe that the methods of attack tolerance we presented, although different can be used in a complementary way to ensure an additional and complete tolerance. Based on these two observations, we propose in this chapter, a complete attack tolerance framework for Web services deployed in the cloud. For this aim, we first explain why we express such cloud applications as a choreography of services that must be continuously monitored and tested. In fact, very often individual Web services are not sufficient to meet the requirements of end users. It may therefore be necessary to compose Web services together to meet complex needs. Choreography one approach for achieving composition is an unambiguous way of describing the relationships between services in a global peer-to-peer collaboration and has the advantage of requiring no centralized actor. Each participant of

that choreography is deployed in a container. Thus, our services will leverage the features of cloud platforms such as elasticity and scalability. Then we will leverage SChorA, a formal Web service testing framework by incorporating some of the attack tolerance mechanisms in order to take advantage of the benefits of each of them. As a result, our approach is unique in the sense that it offers a complete attack tolerance for Web services, from both formal and practical points of view. We will present the theoretical framework of this approach in the following.

6.1 Web services and cloud applications

Service-oriented architectures (SOA) are primarily intended to facilitate the integration of new applications into information systems of companies or organizations by optimizing exchanges and operations. The advantage of SOA is that these services are standardized. They can be interpreted in a simple way by other applications sharing the same standards and potentially exploitable by all the entities of the target system. Besides, it is important to note that Web services and the cloud converge. With the expansion of cloud-based online services or simply the interconnection of Information Systems of companies, the need to expose services to the outside is growing. As we saw in the chapter 2, cloud computing facilitates access to computing resources. These resources are provisioned for public convenience and benefit the availability, the elasticity of cloud computing. The advent of cloud computing has made it possible to provide innovative Information Technology (IT) services. [Yang and Zhang, 2012] claim that SOA and cloud computing are actually complementary since SOA can help realize Software as a Service (SaaS) applications rapidly. So since SOA supports the processing of services and the cloud allows the provision of these services quickly, the combination of these two principles is relevant for businesses giving them more flexibility.

Moreover cloud applications are distributed applications. A distributed application can be defined as a set of concurrently running and interoperable software processes. In addition, it should be noted that there are three models of deployment of distributed applications in the cloud ([Etchevers, 2012]):

1. **Infrastructure-oriented solutions**: Infrastructure-based solutions envision the deployment of an application in the cloud through the implementation of a set of virtualized hardware resources. They come in the form of a public or private cloud. This type of cloud application can benefit from several services provided in the cloud such as database storage, virtual machine cloning, or memory ballooning.

2. **Service oriented solutions**: Service-oriented solutions are platforms where applications are often deployed in the form of composition of high-level services and the administration as the orchestration of these services according to User-defined policies through Service Level Agreements (SLAs).

3. **Application-oriented solutions**: This third category of solutions aims at combining the service-oriented approach, in which a distributed application is defined as a composition of high-level services, and the infrastructure-oriented approach, which explains how an application breaks down within a set of virtual resources. Application-oriented solutions thus offer a high degree of parameterization for the user to define the application to be implemented.

In addition, Cloud Technology Partners (CTP), a Hewlett Packard Enterprise company, claimed that cloud applications are best deployed as a collection of cloud services [Cloudtp, 2018]. The idea is to build up services and then combine those services into composite services or complete composite applications. According to them the benefits of such are manifold. First, the separation of the application services physically, executing on the proper machine instances, can help to track and maintain the services of the application. Additional benefits may include re-usability. One can break up applications into hundreds of underlying services that have value when used by other applications.

In our case, to fully benefit from the benefits of the cloud we will consider our applications as a composition of SOAP Web services in the cloud. The applications will be a choreography of Web services and they will be deployed in the form of containers. The reasons of these considerations are the following:

- **Choreography over Orchestration**. Generally service compositions are classified into two styles: orchestrations and choreographies. Orchestration always represents control from one participant's perspective, called orchestrator. Unlike the orchestration, there is no privilege entities in the choreography. Orchestration provides a good way for controlling the flow of the application when there is synchronous processing. But orchestration incurs a performance overhead due to the additional layer of the orchestration platform itself. A choreography description specifies the interactions between the roles of a collaboration. An implementation of a choreography is a set of services that perform role behaviors. Choreography enables faster end-to-end processing as services can be executed in parallel. In choreography adding or updating services is easier as they can be plugged in/out of the collaboration. However, complexity is shifted in choreography. Each service would have its own flow logic. We choose to deploy our applications as choreography because we want our services to be autonomous and tolerant to attack. The absence of a central orchestrator therefore has the advantage of allowing us to increase reactivity. Furthermore [Kopp and Breitenbücher, 2017] envision to modelling the provisioning of distributed multi-Cloud applications as a choreography model, wherein each participant flow executes the provisioning of one part of the application. They claimed that since composite cloud applications have to be provisioned across multiple different private Clouds, a single centralized provisioning engine or workflow is not possible. [Gomes et al., 2015] also claimed that service choreography would become the norm for internet applications. Indeed, the authors argue that applications are increasingly developed especially based on existing services. Moreover the interconnection of services is facilitated by the use of distributed cloud platforms. [Furtado et al., 2014] argued that Web

services composition, in particular choreography is a suitable solution used to build application and systems on the cloud. They built a middleware solution that is capable of automatically deploying and executing Web services on the cloud. We agree with them that choreography is a good approach for deploying cloud applications based on Web services. In our case we will deploy our application on the cloud as service choreographies that integrate attack tolerance features.

- **SOAP over REST**. It is true that more and more cloud services and cloud applications are deployed using REST APIs. As we saw in the previous chapter, REST "REpresentational State Transfer" is an architectural style and is based on different HTTP methods: GET, POST, PUT and DELETE. The advantage is the simplicity of use and exploitation on the Web. SOAP (Simple Object Access Protocol) is a messaging protocol. It allows programs that run on separate operating systems XML (Extensible Markup Language). Although frequently associated with the HTTP protocol, SOAP supports other protocols such as SMTP. It is usually much slower than REST.

  However, REST APIs can expose to new risks. REST does not implement any specific security patterns like SOAP. Even though some authentication and access control mechanisms are provided, the following bad examples when implementing APIs can induce data loss:

  - HTTPS protected API without any authentication: HTTPS alone may be not sufficient for ensuring security of APIs.
  - Unprotected identity and keys and weak API keys: authentication and identity platforms are used to grant access to services on behalf of a user. Weak semantics of such identity platforms may expose the system to identity impersonating.

  So protecting REST API is of paramount importance. REST APIs developers need to follow specific good practices in order to build strong APIs. In contrast, SOAP uses the Web Services Security (WS-Security [Oasis, 2006]) communications protocol to apply security to Web services. WS-Security allows one to sign SOAP messages to ensure integrity, confidentiality, and security tokens to ensure the identity of the issuer. From a security standpoint, WS-Security (Web Service Security) protocol, which provides end-to-end message level security using SOAP messages, is widely applied in cloud computing to protect the security of most cloud computing related Web services. In addition, cloud providers like Salesforce and Oracle offer SOAP APIs to expose their customer relationship management (CRM) services. This kind of application needs in fact a high level of reliability and security. All these facts justify our choice of SOAP over REST.

In conclusion we propose to develop our distributed applications as a choreography of SOAP Web services and deployed in the cloud. We believe that applications developed in the form of service choreography and deployed in the cloud are ideal because:

- There is a single central component so no strong coupling between services

- A failed service can be started quickly. One can increase the computing and storage capabilities of these services on demand in a transparent way. One can stop these services easily when he doesn't need them anymore.
- One can benefit from the latest security patches implemented by cloud service providers (CSP).
- Cloud providers are proposing redundant mechanisms in several regions of the world.

However, before and when deploying such choreography one should ensure that this choreography is realizable and the participants of this choreography act according to the requirements. We need also reliable tools and frameworks. For this goal, we will leverage SChorA ([Nguyen, 2013]), a conformance and testing framework for choreographies. We will present this framework in the next section.

6.2 SChorA

SChorA [1]is a Web service choreography conformance and testing environment proposed by [Nguyen, 2013]. This framework aims to solve the key issues in choreography-based top-down development; i) Conformance: Does a set of peers conform to the choreography? ii) Realizability: Whether a choreography is realizable *i.e* ensuring that a choreography can be practically implemented. iii) Projection: Ability to derive local models of a global choreography on peers.

To solve these issues, the framework has the following features that we will see quickly:

- A symbolic model and an integrated environment for specifying and analyzing service choreographies;
- Passive testing of Choreographies.

6.2.1 A symbolic model and an integrated environment for specifying and analyzing service choreographies.

A description language for modeling choreography has been proposed. Its semantics are given by Symbolic transition Graph (STG) [Hennessy and Lin, 1995]. It can be used to specify either global view or local view by using global or local events. A STG is a transition system. Each transition of STG is labelled by a guard and a basic event. So the choreography is expressed as a STG. When the models on the roles are available, the conformance is then verified. To verify such conformance, the models of the roles are transformed into STG and then combined to form an overall STG. To verify that this overall STG obtained is equivalent to the STG of the global choreography, the techniques of branching bi-simulation is used. In theoretical computing, a bisimulation is a binary relation between state transition systems, checking whether they behave in the same way.

[1] http://SChorA.lri.fr

A	$::=$	BA	(basic activities)
	$\mid$	$A;A$	(sequential)
	$\mid$	$A \sqcap A$	(choice)
	$\mid$	$A \parallel A$	(parallel)
BA	$::=$	skip	(no action)
	$\mid$	$c^{[i,j]}$	(communication)
Basic:			$[\![\mathsf{skip}]\!] \triangleq \{\langle\rangle\}$
			$[\![c^{[i,j]}]\!] \triangleq \{\langle c^{[i,j]}\rangle\}$
Sequential:			$[\![A_1;A_2]\!] \triangleq [\![A_1]\!]^\frown[\![A_2]\!]$
Choice:			$[\![A_1 \sqcap A_2]\!] \triangleq [\![A_1]\!] \cup [\![A_2]\!]$
Parallel:			$[\![A_1 \parallel A_2]\!] \triangleq [\![A_1]\!] \bowtie [\![A_2]\!]$

(a) Syntax & Semantics

Figure 6.1 – Chor choreography language [Qiu et al., 2007]

In other words, two systems are bisimilar if they are capable of imitating each other. A logical formula is obtained from this bisimulation step. This formula is given as input to a SMT solver that checks for satisfiability. A positive verdict shows conformance. For the realizability and projection issues, the framework works as follows. From the description of the choreography, if there are non-realizable parts, some additional messages are incorporated to the graph to allow all the transitions to be realizable. Once the realizability is verified, there are projected on the different roles or peers.

6.2.2 Passive testing

When role models are not available, the conformance between the specification and the implementation is made possible by passive testing. The author chose the Chor [Qiu et al., 2007] language because it is expressive and abstract enough to enable one to specify collaborations. Figure 6.1 presents the semantic of that language. In order to formally verify the choreography, the author proposed another language ChorD which is an extension of the Chor language with data. The requirements for the roles projected are described in a sublanguage of Chor called role language. The semantics of a Chor (local or global) specification C is given in terms of its set of all specification traces, *trace set* for short, that represent all possible run of the specification [Qiu et al., 2007]. To obtain the requirements for each role of a choreography, there is a projection (nproj) that hides the interactions that do not concern the role itself.

The passive testing approach proposed in [Nguyen, 2013] was the following. From a choreography C, a trace set is obtained. Then, the set of local requirements of its roles is obtained by using the natural projection function as we mentioned previously. From each R_i also a trace set exists. He obtains l_i, from each role. From n collected logs, he synthesizes a global log, denoted as L. A pre-order operator was used to see if one of the traces has some patterns present in the other. When a message is sent or received by a peer of the IUT, an *observation* is recorded in the log file. At the role level, interactions are represented with "!" that indicates a sending and "?" that indicates a reception. Observations are then

annotated accordingly in logs. To ease the reconstruction of the order between observations of different logs l_i, the author incorporated the time of the observation in the log. Formally, the observation is defined as follows. Given an Implementation Under Test (IUT) which consists of n peers $P = \{P_1, \ldots, P_i, \ldots, P_n\}$, an *observation* is a tuple $ob = (act, id, t, s, r, m)$ where $act \in \{SEND, RECEIVE\}$ indicates a sending or a reception, id is a message identity, t is the reception or sending time, $s, r \in P$ with $s \neq r$ are the sender and the receiver, and m is the message. A log $l_i = \langle ob_1, ob_2, \ldots, ob_m \rangle$ for a peer P_i is a sequence of all the observations of messages which are sent/received by P_i to/from others peers of the IUT.

The previous conformance relation (pre-order relation) while useful, was not strong enough to detect complex misbehaviours. To improve the detection, another complementary approach was proposed. This new approach can detect non conformance by the verification of properties provided by the standards or by the choreography experts. In order to express such properties some terminologies were proposed: message and event, candidate events local and global properties. The messages and event exchanged between the members where formalized as follows.

Definition 6 (Message and Event). *Given a finite set of action names Ω, of labels , and of atomic data values , a message m takes the following form:*

$$o(l_1 = v_1, \ldots, l_n = v_n)$$

where $o \in \Omega$ represents the action. The composite data of the message is represented by a set $\{l_1 = v_1, \ldots, l_n = v_n\}$, in which each field of this data structure is pointed by a label $l_i \in$ and its value is $v_i \in dom(v_i)$.

For example a message can be:

$$Request(location = Paris, people/name = Georges, people/job = researcher)$$

A message is an instance of an event. It means that an event expresses a set of messages which have the same operation name. For deriving a sub-class of an event in which the messages respect a particular condition the term candidate event was proposed. A candidate event is then a pair event/predicate. Formally candidate events are defined as follows:

Definition 7 (Candidate Event). *A candidate event (CE) is a pair $o(l_1 = v_1, \ldots, l_n = v_n)/\phi(x_1, \ldots, x_n)$, denoted by $o()/\phi()$, where $o(l_1 = v_1, \ldots, l_n = v_n)$ is a message and $\phi(x_1, \ldots, x_n)$ is a predicate. The predicate can be omitted if it is true.*

Example 1. *An example of CE is:*

$$CE_1 = \mathit{People}\ (age = x)/(x >= 18)$$

Which represents any adult people i.e. whose age is greater than 18.

For detecting misbehaviours one should express properties both at the role and the choreography level. As consequence there are two types of property, local and global. The local properties of a member of a choreography are expressed as follows.

Definition 8 (Local Property)**.** *A local property* P *is a 2-tuple (cont, conseq) composed of a Context* $cont$ *AND/OR a Consequence* $conseq$ *AND/OR an Operation* op*. It is denoted as:*

$$P ::= cont \xrightarrow{d} conseq$$

where:

- d *is an integer,* $d > 0$
- *Context is a sequence of CEs,* $o_1/\phi_1, o_2/\phi_2, \ldots, o_n/\phi_n\rangle$
- *Consequence is a set of CEs,* $\{CE_1, \ldots, CE_m\}$*.*

On the other hand, the global properties, the properties of the choreography, should verify are defined as follows.

Definition 9 (Global Property)**.** *A global property is described as:*

$$G ::= SET \rightarrow SET'$$

where SET *and* SET' *are two sets of local properties.*

These properties are then checked on the execution traces of the IUT which are collected at running time.

Summary: We chose to leverage SChorA in this chapter for the following reasons. First, we explained in section 6.1 that we wanted to take advantage of the properties offered by the cloud such as the elasticity to deploy our Web services-based applications. In addition we showed why we prefer choreography rather than orchestration. SChorA by its formal richness is a tool that is perfectly suited for verification of our Web services choreographies. SChorA also supports the passage of values in choreographies. This is particularly useful for checking complex interactions. Secondly, SChorA has some similarities with the approaches we proposed in previous chapters. This is particularly the case of the reflection-based attack tolerance approach. As a reminder, the detection of attacks by reflection was done by the use of logs files deployed in each of the components of our application. We developed a plugin to analyze properties that we defined through the formalism of MMT. In comparison SChorA also uses logs for passive tests. Moreover, the semantics of the properties of the passive testing tool are similar to those of MMT. We believe that using SChorA in conjunction with our attack tolerance methodology can only be beneficial. Thus we will see in the following section how concretely this can be done.

6.3 Attack tolerance in the cloud

This section will present in detail how we will leverage SChorA as well as our previous work for building a full attack tolerance framework. Adding mechanisms of detection and reaction on the fly to these applications verified correct, ensures optimal attack tolerance. Our framework consists of two main parts (Figure 6.2). The first part that deals with modeling, verification and code generation and the second part where the system is deployed, tested and monitored. A part of the methodology in this section has been illustrated through a vote choreography in Appendix A.

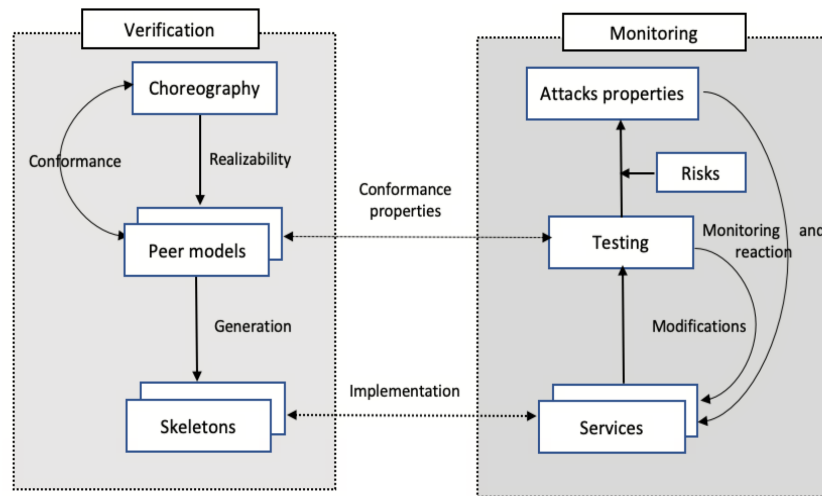

Figure 6.2 – SChorAcloud architecture and components.

6.3.1 Part 1: Verification and code generation

We proposed to model any Web-based application as a choreography of Web services using the description language of SChorA. We check our choreographies from a formal point of view. From the overall choreography, one verifies the realizability thanks to SChorA. Once this stage is over, we project the choreography on the different roles. We then obtain local models on the peers. After having formally verified the conformance of the models, *w.r.t* the choreography, it is advisable to be able to implement these projected models on the peers in a way that the services obtained are correct by construction. The goal is to automate code generation. We propose for this aim to use a new Domain Specific Language (DSL) for our choreography called *ChorGen*. DSLs are language families dedicated to a particular domain (SQL for databases is an example, unlike GPLs (Java) that can potentially handle all domains). DSLs have been promoted because users or Experts in a given business domain may not have the skills to write programs in traditional languages. However they can more easily understand terms related to their target domains, for example a baker will

understand the key words, cake, cookies than classes (Java). The *ChorGen* language has the following grammar:

```
grammar org.xtext.example.Chor with org.eclipse.xtext.common.Terminals

generate chor "http://www.xtext.org/example/Chor"
Model:
(choreographies+= Choreography)+;

Choreography:
'choreography' name=ID '{'
(roles+=Roles)*
'}'
;
Roles:
'role' name=ID '{'
  operations+=Operation
'}'
;
Operation:
'operations' '{'
   (methods+=Function)*
'}'
;
Function:
name=ID '('(params+=Param)* ')'
;
Param:
name=ID type=ID ',' |name =ID type=ID
;
```

This means that a choreography contains several roles that expose some operations to interact with the other roles. An example of a well defined implementation is the following:

We used model-driven engineering technologies (Xtext[2] for grammar analysis, Xtend[3]) for code generation) for the specification, the semantic, the compilation and code generation from the *ChorGen* language to our three target languages: Wsdl, Python and PHP. Indeed these tools allow one to easily create associated compiler/ interpreter and even a suitable environment (code editor with syntax highlighting, autocompletion, etc.). So once the local models are projected from the global choreography, we write the corresponding codes using the syntax of the *ChorGen* language taking into account the interactions added to the specifications. This is the case for example when in the verification phase, interactions are added to the models to make the choreography realizable. After we generate the skeletons

[2]https://www.eclipse.org/Xtext/
[3]https://www.eclipse.org/xtend/

```
choreography chor{
    role client{
      operations {
        send(IP String)
        receive(Data string)
    }
    }
    role server{
      operations {
        verify(IP String)
        ack(IP string)
    }
    }
}
```

Figure 6.3 – Example of definition of a choreography with 2 roles

of the services that will implement our choreography thanks to the functionalities offered by our language. In particular, we generate the Wsdl files (interface file of our Web services) as well as the skeletons of the implementations of these services in the Python, PHP and Java languages. In addition to the skeletons of these services we have the possibility to generate JUnit (if the developer chose Java) test cases that will allow us to test the services once implemented (functional testing).

The advantage of doing such code generation is the reduction of service development time. This allows us to be more efficient when implementing the services. This is useful, for example, for choreographies containing a very large number of peers. Another advantage is that since all interactions are taken into account in *ChorGen*, we are sure that developers will not forget to implement them since their signatures are available. This is very suitable for top-down approaches of choreography development. After having these skeletons generated, the developer completes the implementations. The next step consists in describing some properties in the formalism we presented above. The properties are check by SChorA's test engine in order to see if the implementations are conform to the choreography. The purpose is to detect non conformance *w.r.t.* the requirements and correct them before deploying the services. Furthermore, the current version of SChorA test engine can not explain the backward properties. From the vote choreography presented in Appendix A, backward properties are: *i)* every citizen must be registered before he/she can vote, *ii)* the list of candidates must be sent only to a registered citizen. This kind of properties can be implemented in MMT.

Algorithm 2 Observer or manager execution

```
1:  detected ← false
2:  while true do
3:      if detected then
4:
5:          setLocalCountermeasures()
6:          chosenVariant ← random(1, n_variants)
7:          chosenVariant.state ← active
8:          notifyAllMembers(IP)
9:          nextInstruction ← buffer.pop()
10:         setTimer(Timer)
11:
12:         while Timer do
13:             buffer ← buffer ∪ ack()
14:         end while
15:
16:         for ( do i=0 To N)
17:             if !ack[i] then
18:                 notifyMember(IP, i)
19:             end if
20:         end for
21:         bind(chosenVariant)
22:         sendInstruction(chosenVariant, nextInstruction)
23:     else
24:         if receiveChange[IP,i] then
25:             sendAck(mac(IP), i)
26:         end if
27:     end if
28: end while
```

6.3.2 Part 2: Deployment, monitoring and reaction

To ensure redundancy, as we described in the previous step, we suppose that there are diversified implementations in three target languages, Java, Php and Python. Added to this, we diversify the data structures and the sequences of the instructions so as to have different AST for the variants of the implementations obtained. So we will have at least 6 different implementations for each of the peers in the choreography.

In accordance with our attack tolerance methodology, one implementation at a time is chosen. The others are started but inactive. In addition, we assume that the implementations are localized in a safe environment that is to say are not connected to the outside and that only the active implementation can be attacked. When the current implementation is attacked, it is replaced by one of the variants. We assume that the communication channels between the choreography members are reliable. So, only the different roles can emit false messages when they are attacked or compromised. Each member of the choreography is deployed in a container on a public cloud platform. They are observation probes (algorithm 2) available for each role in the choreography. These agents are in charge of monitoring and detecting attacks. They will also be given the ability to implement the replacement of the container when the attack is detected. These agents do not normally interact directly with the members of the choreography. They are therefore not visible from the outside and are located in a safe environment. They are considered safe and they do not crash. The members of the choreography have each a log file that will be used for detecting attacks. We stored Date, Hour, Operation ,hash on these logs.

When a member is attacked, with respect to the rules described in the MMT tool, the monitoring agent interrupts the connection of the attacked container while saving the last unexecuted requests. It replaces the current damaged container with another container (line 5,6 in algorithm 2). Afterwards, he notifies the other members of the choreography and send them the information about the new active implementation (line 7). He expects to receive acknowledgments from the other members during a certain period of time (line 9 to 13). For example, it sends the new IP address of the new deployed member to the other members of the choreography. Acknowledgments should be received with the correct hash of the IP address to ensure that members have obtained the correct address. After the end of the timer, if he does not receive the acknowledgments from all members, he sends again the message to the members (Line 15 to 19). Otherwise if there is no detection of attack, the manager keeps working. If he receive a message of a new member (line 23) he answers back (line 24).

In the same way in algorithm 3 we describe how the members of the choreography act. They are initially in the inactive state. Then, one of the members is chosen after a detection of attack or misbehaviour. This member becomes active (line 6 algorithm 2), launches the list of remaining non executed instructions he receives from the manager (line 4 algorithm 3) and continues to perform his normal tasks (algorithm 3 line 7). We see the value of using these applications in cloud computing because you can deploy/disconnect containers quickly

Algorithm 3 Choreography member execution

```
1: while True do
2:     if receiveActivation then
3:         if receive(instruction) then
4:             execute(instruction)
5:         end if
6:
7:         continueExecution()
8:     end if
9: end while
```

Since the implementations of our services verified conform to the description of the choreography, the next step is to ensure that when these services are deployed, there will be no other misbehaviours such as attacks or viruses. The idea is to leverage the risk-based monitoring presented in chapter 3. Recall that in this methodology, we first identify the assets we want to preserve and then we analyze the risks (attacks, vulnerabilities) that these assets may be confronted with. In our case we will derive properties from the risk analysis phase. These new properties will be added to those in the previous section to continuously monitor the services. The monitoring is done by using the reflection methodology as presented in chapter 5. Thus, at runtime when a misbehaviour is detected, we react by applying the attack tolerance methodology (rejuvenation and recovery of the implementation and reflection).

In summary, the framework is useful for the following reasons. First of all, our services are in a sense correct by construction since the code generation is made from the projected models of the choreography which is verified realizable. Then tests will be performed before and when the services are deployed to detect possible deviations of the implementation *w.r.t.* the choreography. To detect attacks that can happen, after risk analysis, we have a continuous monitoring and reaction that mitigate the effects of those attacks. The introduction of the reflection methodology ensures tolerance to a wide variety of attacks and threats including malware. Last but not least, deploying the choreographies in the cloud allows us to take advantage of the elasticity and continuous delivery capabilities offered by cloud services. This enhances the attack tolerance of such choreographies since the time to recover from an attack is short. We will then have a full attack tolerance loop for services-based applications deployed in the cloud. This comprehensive approach is therefore different from existing approaches that aim to simply deploy choreographies in the cloud without focusing on robust means of verification and monitoring.

6.4 Discussion

We proposed in this chapter an approach of attack tolerance of applications deployed in the cloud and developed from Web services. We first expressed these applications in the

cloud as service choreographies. We have detailed and explained the interest of modeling these applications in this way. These choreographies that can be verified (conformance, realizability and projection) using SChorA a formal environment for choreography testing and verification. In addition, we developed a DSL that allowed us to automatically specify and generate skeletons of our Web services in the target programming languages. The idea was to diversify the implementations as we mentioned in chapter 4. This is one contribution of this chapter. According to the methodology, once these services are developed, the next step is to deploy the choreography with the peers launched on clouds. For the detection of attacks or misbehaviours, the idea was to leverage the reflection-based attack tolerance presented in chapter 6 5 in conjunction with SChorA property-based detection of misbehaviours. This will result in the specification of some specific rules that aimed at monitoring the whole framework, locally (at the peer level) and globally (at the collaboration level).

Moreover, we have at our disposal, the following software components. The MMT plugin that parses information contained in log files and write security properties on them is operational. The format of the properties of this module is the same as the format of the properties of MMT presented in the chapter 3. The modeling and verification components of the SChorA framework as well as *ChorGen* for code generation in Wsdl, Php, python and Java are available. Some parts should be refined (SChorA property testing) so that we can validate this framework experimentally. As a perspective, we think that in order to fully implement this approach, one should follow this experimental protocol:

1. Specifying the choreography in the ChorD language and verifying the realizability, conformance and projection of that choreography using SChorA conformance tool.

2. Generating the Wsdl files and the skeletons of the peers in Java, Python and Php and implementing them.

3. Specifying some properties in order to check the conformance of the implementation *w.r.t* the choreography and expressing them in the formalism of SChorA passive test engine. Specifying the remaining properties using the formalism of MMT.

4. In line with our risk-based monitoring approach, checking of the assets of the choreography in order to anticipate potential vulnerabilities and attacks.

5. Specifying some MMT properties for monitoring the peers and detecting attacks.

6. Deploying the peers on different cloud providers to enhance diversity.

7 Conclusion

Contents

6.1 Web services and cloud applications 102
6.2 SChorA 105
6.2.1 A symbolic model and an integrated environment for specifying and analyzing service choreographies. 105
6.2.2 Passive testing 106
6.3 Attack tolerance in the cloud 109
6.3.1 Part 1: Verification and code generation 109
6.3.2 Part 2: Deployment, monitoring and reaction 113
6.4 Discussion 114

7.1 Synthesis of results

In this thesis, we proposed a new attack tolerance framework based on formal monitoring techniques as well as software engineering techniques. We claim that a good tolerance requires attack detection and continuous monitoring on the one hand; and reliable reaction mechanisms on the other hand. We obtained the following results;

Risk-based monitoring. We leveraged the traditional risk management loop to build a risk-based monitoring that integrates risks into monitoring. We claimed that the detection and prevention of attacks require a good knowledge of the risks that these systems face. As such, it is mandatory to include risk management in the monitoring strategy in order to reduce the probability of failure or uncertainty. This methodology involves the following aspects: i) assets identification to define what is necessary to protect. ii) Threats and vulnerability analysis, to evaluate the potential flaws the system may suffer. iii) Risk analysis to categorize the threats that can exploit the system vulnerabilities. iv) System

monitoring to detect potential occurrences of attacks, and. v) Remediation strategies to repel or mitigate the impact of the attacks.

Diversity-based attack tolerance. First, we presented a model-based diversity for attack tolerance. We investigated attack tolerance at the design and specification phase. The idea was the following. From a formal model of a system, *i.e.* a representation of the system namely as a Timed Extended Finite State Machine (TEFSM), we derive equivalent models that are functionally equivalent and that are supposed to be more resistant. From these models, we derived the corresponding implementations. We ensure that these models has been validated using, for instance model checking and testing techniques. As an example, we can consider an authentication Web application, which is based on a password mechanism and we defined other models using more complex authentication such as n-factors authentication. The experiments show that the proposed mechanisms could tolerate attacks such as brute-force attacks. However, the principal difficulty of this method led in the derivation of the equivalent models. Moreover, one of the remaining questions that needed to be solved was: how can we synthesize these variants?

Secondly, we proposed a complementary approach. This contribution aimed at extending and solving the issues raised by the former approach. During all the steps of the construction of our software, *i.e.*, from modeling to the concrete instantiation, we have integrated diversification. The idea was still the same but we only have one model and several implementations. Our intuition laid on the fact that having diverse layers and points of failure increase the security as it allows tolerance to a greater variety of attacks. We illustrated the approach with a Web service that simplifies the management task in a hospital. First of all, we defined a Feature Model that describes the variability points of the service. Three variability patterns (Encoding style, Encoded type, Language) and the corresponding WSDL files were defined. After this specification step, the derivation of the variants was made possible in a semi-automatic manner. In fact, we used gSOAP, a C/C++ compiler, to generate the corresponding skeletons of the code of our Web services and implemented them. To add more randomness we diversified the binaries of the services to obtain different equivalent implementations of the Web service. We tested the framework with a DoS attack to check the attack tolerance. The experiments show that our approach tolerates certain types of attacks with a relatively low latency.

Reflection-based attack tolerance. Our aim was to address attack tolerance for insider attacks. We investigated meta-programming techniques in particular software reflection. Reflection helps a program to monitor or modifying its components and behavior at run-time. We relied on reflection to build our method of attack tolerance. The basic idea was therefore the following. We considered that the software of the client is located in a safe environment. Some potential attacks that can take place are internal ones. That is, coming from internal hackers. The goal of the intruder is to usurp the actions, *i.e.*, to modify the methods of the API of the platform. We designed a layered framework. By reflection we obtained all the hash of the source code of any methods of the API. Any deviation

at runtime of that hash value means the presence of a misbehavior. Such misbehavior could be whether an insider attack or a virus attack. We stored date, hour, operation ,hash, host in the file. Any request has then two traces in the logs: outbound(request) and inbound(response). We described the situation when an attack occurs, *i.e.* someone has modified the API and overridden one or several methods. First, this is the case where we see some information of the methods but there is no hash. It is also possible to get only one hash for the outbound operation and nothing for the inbound operation. The hashes of both Outbound and Inbound could not correspond in the log files, if there is an attack. Finally, we can get some inconsistencies in the logs: timestamps incoherence, method inconsistencies (answer before request), or combination of inconsistencies. For the monitoring part of the framework, the programs are checked at runtime using reflection as we mentioned earlier. For detecting attacks, we use logs located on the two endpoints: on premises, on the server. We developed a new plugin for this kind of detection in the monitoring tool MMT. We applied some security policies (rules). We conducted two different but complementary experiments :

- We measured the average time to make a client request without attack and when the attack is detected.
- We evaluated the ability of the framework to detect viruses attack in comparison to a conventional anti-virus.

The experiments show that our attack tolerance is effective.

Attack tolerance framework for services-based applications. We proposed a theoretical correct-by construction and correct-at runtime attack tolerance framework for Web services-based application in the cloud. For this goal, we first express any application deployed in the cloud as a choreography of services which must be continuously monitored and tested. Choreography is an unambiguous way of describing the relationships between services in a global peer-to-peer collaboration and has the advantage of requiring no centralized actor. Each participant of that choreography is deployed in one virtual machine or container. Then, we extend a formal framework for choreography testing by incorporating the tolerance methods for detecting and mitigating attacks presented. Adding mechanisms of detection and reaction on the fly to these applications verified correct by design, will ensure optimal attack tolerance.

Throughout this thesis, we have proposed innovative mechanisms for attack tolerance for Web services deployed in the cloud. We believe that these techniques may be applied for other types of applications. Now let's discuss improvements and open directions.

7.2 Perspectives

We defined in this thesis attack tolerance or intrusion tolerance as the capability of a system to continue to function properly with minimal degradation of performance, despite

intrusions. The aim was to detect the known and unknown attacks and if not possible to reduce their impact on the system. Although we obtained satisfactory results, we believe that we could improve the tolerance to attacks if we could somehow anticipate or predict these attacks. So in addition to detection and remediation, it would be necessary to be able to predict and anticipate future attacks. We think that these two following axes would be interesting to investigate.

Diagnosticability and predictability. [Ibrahim, 2016]. Diagnosis consists in designing and implementing algorithms for verifying the formal properties of the system, ensuring that a model, which is known in advance of observable events, allows the detection and the discrimination of a set of possible failures. Similarly, predictability is the ability to predict a future occurrence of a fault using the observable events preceding. We think that if we can predict the occurrence of a fault, it would be interesting to prevent it from taking place and therefore to tolerate attacks effectively. However, an important step for using diagnosticability and predictability for attack tolerance will be therefore the formalization of faults that can occur from attacks.

Big data and machine learning. Recently Machine Learning have emerged as a mean to enhance security ([Buczak and Guven, 2016]). The authors describes the literature review of Machine Learning (ML) and data mining (DM) methods for intrusion detection. This study evaluated the different existing algorithms. They pointed out that the most effective methods for cyber detection must be established and adapted to the specificity of the attacks. Furthermore, adding big data to machine learning ([Suresh et al., 2016]) can improve cyber security. The introduction of Big Data processing led to a new era in the design and development of large-scale data processing systems. The idea is that data in raw format makes it possible to create statistical baselines to identify normality. Subsequently, it is possible to instantly determine when the data deviates from this standard. This historical data also makes it possible to create predictive and statistical models. While some supervised and unsupervised learning algorithms are already available for Big Data, there is big room for improvement. It has be recognized that the false-positive rate of machine learning algorithms, are too high and the alerts generated are not always sufficiently interpretable to enable their exploitation. In summary, there is research avenue for the application of such techniques for attack tolerance. [Suresh et al., 2016].

The result of using either/both predictability or ML and Big data in conjunction to our attack tolerance methods will be the design and the implementation of a framework for software systems, that are attack tolerant in the sense they have the possibility to continue to deliver their services even when after a successful attack, and be able to recover quickly and learn from the past.

A
Appendix

A.1 Example: Vote application

For illustrating the approach of chapter 6, we propose an electronic voting application for the election of the president in a certain country. This application allows citizens to register on the electoral lists and to vote electronically. The application is described by the *VoteElecService* choreography. It is composed of three basic members: *Inscription*, *Vote* and *Citizen*. The first member of that choreography allows to register a citizen on the electoral lists by providing the personal information (surname, first name, date of birth, address, ...). Another service called *Inscription* supports the Vote service. Once registered, this citizen can vote electronically after verification of his registration by the member *Vote*. Subsequently, this service will provide the list of associated candidates and their identification number (1, 2, ...) in addition to the number zero that is associated with the blank ballot. A registered citizen will vote by selecting one or more voting numbers (including the blank ballot) and submitting his/her choices.

A.1.1 Verification and code generation

The vote choreography is modelled in Chor as follows: $\mathsf{inscription}^{[1,2]}$;$\mathsf{voteRequest}^{[1,3]}$; $\mathsf{resultVerifInfo}^{[3,2]}$; $\mathsf{rejection}^{[2,1]}\sqcap(\mathsf{confirmation}^{[3,1]};\mathsf{liste}^{[3,1]};\mathsf{vote}^{[1,3]})$ where the Citizen member, the Inscription member, the Vote member and the result member correspond to respectively 1, 2 and 3. In order to formally verify the choreography, we adapted this specification in the ChorD language which is an extension of the Chor language with data. The resulting specification is: $\mathsf{inscription}[c,i].<\mathsf{info}>;\mathsf{voteRequest}[c,v].<y>;\mathsf{resultVerifInfo}[v,i].<x>;([x=0]|>\mathsf{rejection}[v,c]+[x!=0]|>(\mathsf{confirmation}[v,c];\mathsf{liste}[v,c];\mathsf{ote}[c,v]))$. The results of the formal analysis of the choreography are depicted in the following figure A.1.

As one can observe, the choreography is fully realizable without the need of adding new interactions. It is therefore generated on the different roles. We also describe some implementations models in order to check the conformance of the locals models, *w.r.t* to the choreography (figure A.2). And from these descriptions, we generate skeletons of the roles (figure A.3) that the developer should complete later.

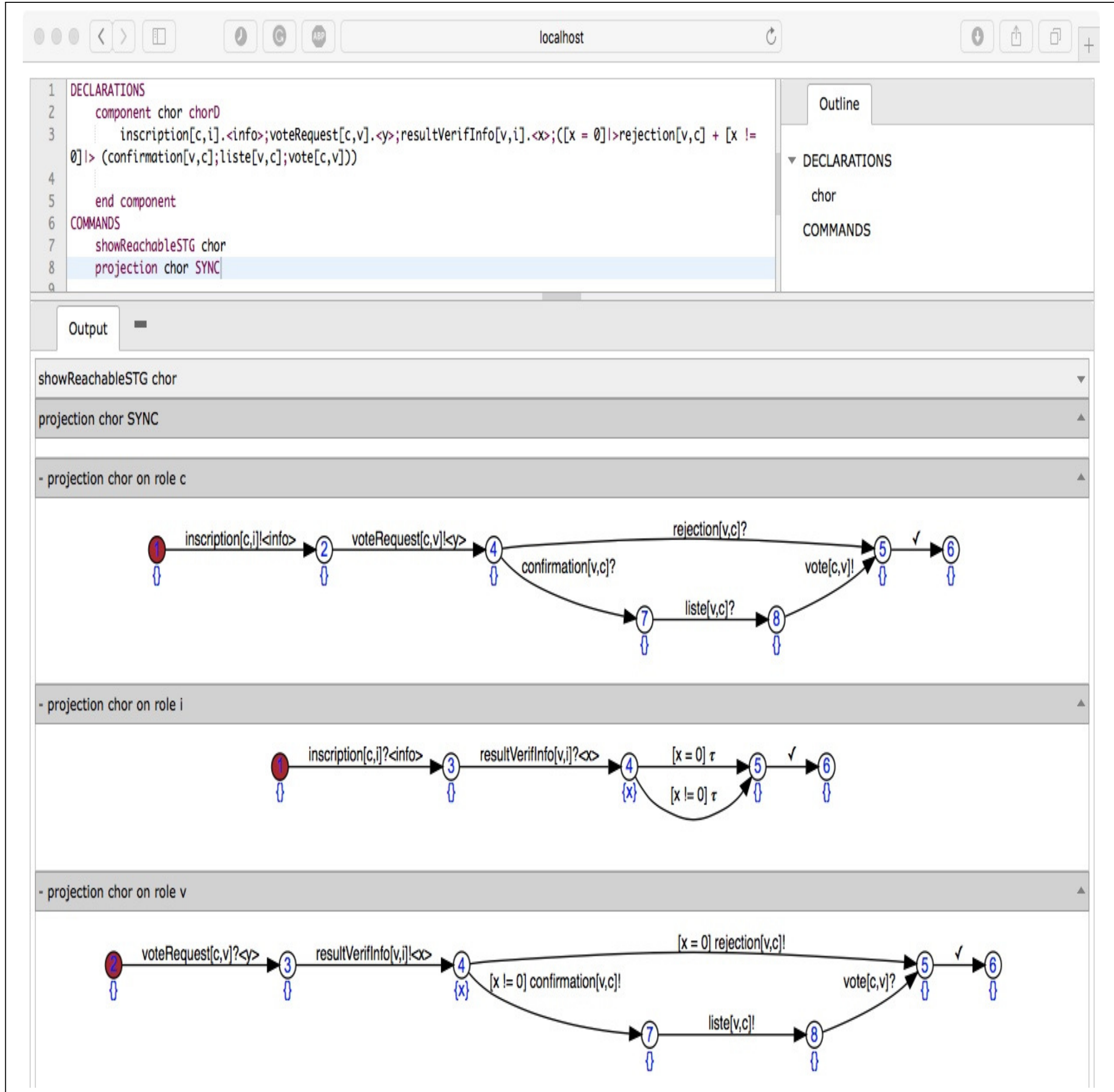

Figure A.1 – Verification of the Vote choreography

A.1.2 Testing

For the choreography conformance, we define the following rules:

1. Every citizen must be registered before he/she can vote;
2. When the citizen does not exist in the electoral files, we must send him a refusal;
3. The service must answer all the requests of the citizens;

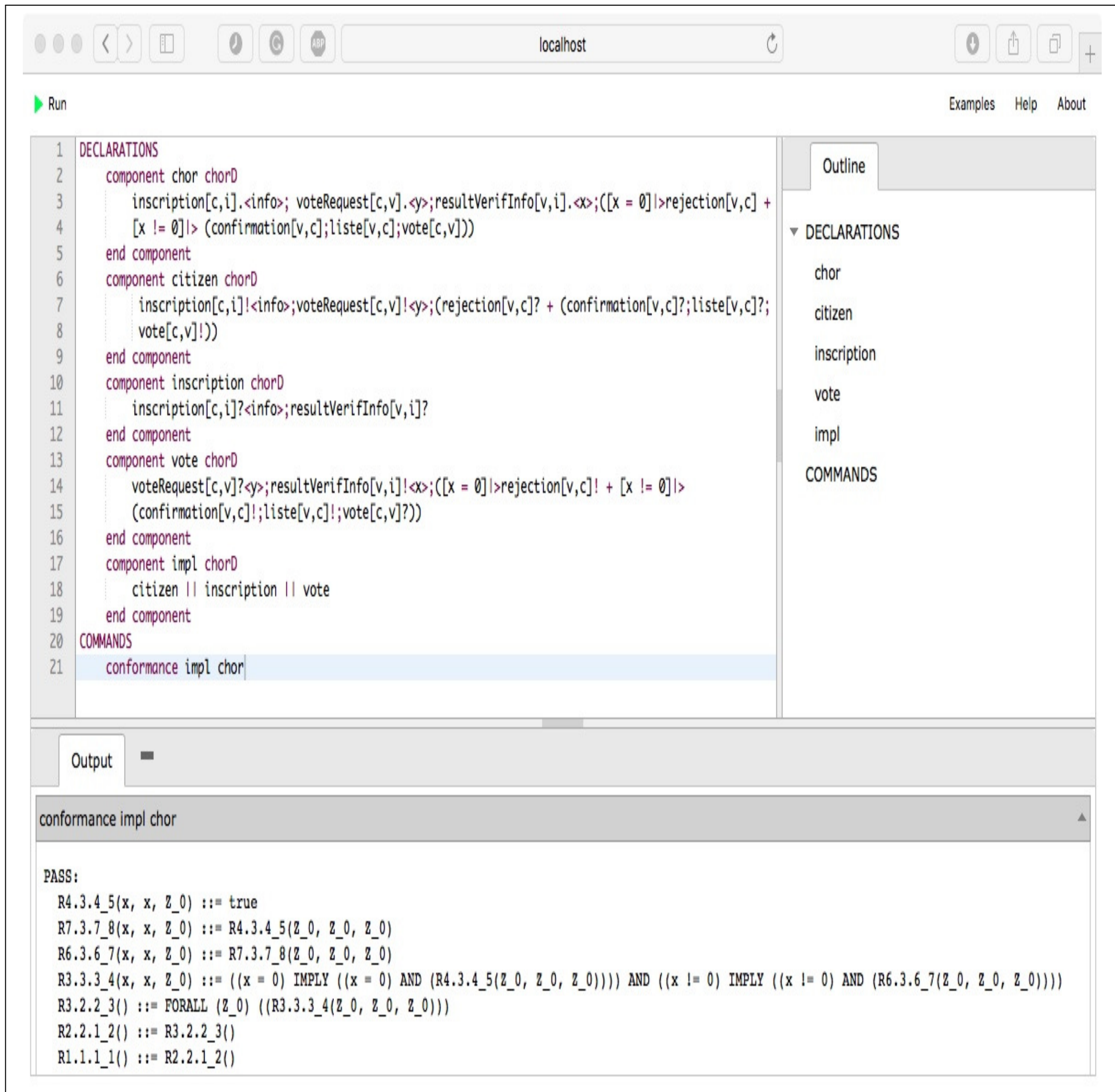

Figure A.2 – Conformance Checking

4. The service must validate the exact choice of the candidate;

5. The service must send the correct list of candidates to a citizen;

6. A citizen can only vote once;

7. The list of candidates must be sent only to a registered citizen;

8. A citizen must receive the list of candidates before making his choice;

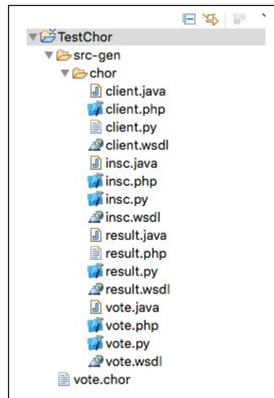

Figure A.3 – Generated skeletons

9. A citizen should be able to vote whenever he/she wants the first time;

10. The service should register any citizen.

The properties i) the service must validate the exact choice of the candidate; ii) The service must answer all the requests of the citizens. iii) the service should register any citizen can be specified by a local property of Citizen role: P1 = voteRequest[c,v]!(-)> -(2)-> rejection[v,c]?(-), confirm[v,c]?(-) .

For the global properties, as there is a local order of events, the local ones are enough to verify the conformance of the whole choreography. The global properties are only necessary in the case there are no local order of the events.

Moreover, in line with our risk-based monitoring approach the main assets remain the votes of the citizens and the availability of the platform. These citizens must be able to vote at any time of the election day. The main attacks that can be expected are the usurpation of the citizens identity, the modification of the votes (by a human being, by a program or a virus), the denial of services (XML DoS). For detecting such attacks the following rules/or prevention mechanisms can be checked:

- XML Dos Attacks: the attack is executed by sending a very large SOAP message to the attacked web service. The countermeasure is to use strict schema validation in the Wsdl files *i.e.* limiting the range of the data types.

- Usurpation of the identity of the user: The implementation of strict authentication mechanisms such two factors authentication and very strong authentication passwords are sufficient.

- Modification of the source code of the voting service: We assume that a member of the development team can modify the algorithm of the vote method in order to help a candidate or a political party of his choice to win the elections. A property (figure A.4) consists in ensuring that every method called should have the same hash as previously before deployment as we did in chapter 5.

```
<beginning>
    <property value="THEN" delay_units="s" delay_min="0+" delay_max="2"
      property_id="10" type_property="ATTACK" description="Detection of
      the insider attack: Any update request should have hashes for its operations
      (outbound and inbound) on the log files and these hashes must correspond.">
        <event value="COMPUTE" event_id="1" description="reception of the
          inboud operations in the log" boolean_expression="((log.hash == '1')
          &amp;&amp; ((#strcmp(log.method, 'vote(inbound)') != 0)&amp;&amp;
          (log.hash != ''))"/>
        <event value="COMPUTE" event_id="2" description="reception of the
          outbound operation in the log" boolean_expression="(log.hash!='')
          &amp;&amp;(#strcmp(log.method, 'vote(outbound)') == 0) &amp;&amp;
          (log.hash!='') &amp;&amp; (#strcmp(log.hash, log.hash.1) != 0))"/>
    </property>
</beginning>
</beginning>
```

Figure A.4 – Security rule in MMT: The hashes of the called vote method should be equal to the hash existing in the database before that call

A.2 Résumé de la thèse en Français

A.2.1 Contexte

Les systèmes informatiques sont actuellement au coeur de toutes les fonctions de l'entreprise (comptabilité, production, facturation...) et plus généralement sont au coeur des activités habituelles du quotidien. Ces systèmes sont très souvent composés d'applications et de données hétérogènes. Ces composants doivent être donc décrits par des architectures modulaires qui permettent la composition et l'intégration de ces composants pour satisfaire les besoins des organisations. Les architectures orientées services (SOA en anglais) ont été proposées ces dernières années pour atteindre cet objectif de modularité.

Une architecture orientée service est un paradigme qui permet aux organisations d'avoir une infrastructure informatique répondant à leurs besoins métier. Ces architectures sont distribuées et facilitent la communication entre des environnements hétérogènes. Le principal élément d'une architecture orientée service est le service Web. Un service Web est une collection de protocoles et de standards pour l'échange de données sur le Web. Cela facilite donc la communication de différentes applications conçues avec des techniques et des langages différents. Ces services peuvent être situés localement ou distribués géographiquement sur des environnements virtualisés tels que le cloud.

En effet, avec les avancées technologiques dans les réseaux de communication en particulier Internet, et l'expansion des services en ligne, le besoin d'exposer les services pour attirer des clients ou pour asseoir sa compétitivité sur Internet s'est considérablement accru ces dernières années. Les plateformes Cloud par exemple permettent le partage des ressources informatiques (stockage, calcul, réseaux...) à la demande au travers d'Internet.

Ces services sont souvent déployés sous la forme de composants unitaires (machines virtuelles, conteneurs, machine sans serveur). Avec la démocratisation d'Internet et du Cloud, bon nombre d'organisations ont réalisé d'énormes bénéfices en déployant des architectures orientées service. Ces bénéfices sont notamment perçus en terme de réduction des coûts, d'agilité et un time-to-market qui continue de croitre.

Cependant, les services Web, par leur interorpérabilité et le fait qu'ils soient parfois exposés sur Internet fait d'eux des cibles potentielles des attaques ou autres comportements malveillants. Les services Web aussi bien que les autres technologiques tirant profit d'Internet, sont soumis à des attaques visant la disponibilité, l'intégrité et la confidentialité des plateformes et des utilisateurs. De plus les services Web déployés dans les environnements Cloud héritent des vulnérabilités de ces derniers. Récemment de nouvelles attaques exploitant les vulnérabilités du cloud on émergées (attaques par canaux auxiliaires, détournement de machines virtuelles, usurpation d'identité...). Ces attaques ont réduit l'efficacité des outils de détection et de prévention classiques disponibles sur le marché. Très souvent ces attaques sont perpétrées soit par des organisations ou entreprises pour infliger des dommages aux entreprises concurrentes, pour leur infliger des pertes financières ou pour voler de la propriété intellectuelle; soit par des individus en manque de reconnaissance et avides d'argent. En substance, les systèmes d'information des entreprises sont aujourd'hui sous la menace constante d'attaques malveillantes. En parallèle, la recherche dans la communauté des services Web s'est focalisée sur la modélisation, la composition et la vérification. Certes ces recherches ont amélioré la confiance et la fiabilité des services Web, néanmoins elles peuvent s'avérer insuffisantes pour assurer complètement les besoins en sécurité des infrastructures orientées service. De plus, peu de travaux proposent des solutions de détection pratiques et ces derniers sont limités par leur degré de détection. Les techniques proposées sont bien souvent restreintes à un seul type d'attaque. A cause de toutes ces limitations, les services Web peuvent ne pas pouvoir accomplir parfaitement les tâches qui leur sont assignées. Nous pensons qu'il n'est pas suffisant de seulement détecter les attaques. Les services Web doivent également disposer de moyens pour faire face à ces attaques. Leur continuité de service est donc cruciale. Ceci étant, les outils traditionnels de détection des intrusions devraient être améliorés et de nouveaux mécanismes de détection plus sophistiqués devraient être proposés. Le but de notre thèse est donc essentiellement de répondre à la question : comment rendre les services Web tolérants aux attaques?

A.2.2 Contributions

Nous définissons la tolérance aux intrusions/attaques comme la capacité d'un système informatique à assurer un fonctionnement normal avec une dégradation minimale des performances malgré les intrusions. Le but étant de détecter aussi bien les attaques classiques et si cela n'est pas possible, réduire leur impact sur le système. Nous pensons néanmoins que pour assurer une tolérance effective des attaques une détection en amont suivie d'une réaction en aval est plus que primordiale. D'autre part, il est aussi opportun de tenir compte des contraintes fonctionnelles aussi bien que des contraintes non-fonctionnelles et ce, dans toutes les phases de développement du système à savoir, la conception, le

développement, la recette et le déploiement. Les contraintes fonctionnelles sont en fait des contraintes qui définissent une fonction du système à développer tandis que les contraintes non-fonctionnelles sont des contraintes qui caractérisent une qualité ou un attribut du système telle que la performance, la robustesse, l'adaptabilité, la disponibilité... Dans cette thèse, nous adoptons une approche de bout-en bout de la tolérance aux attaques en se basant essentiellement sur les mécanismes de diversification et sur les méthodes formelles. C'est pourquoi nous proposons une méthodologie de supervision formelle qui tient compte des risques pour l'évaluation de nos services Web. Pour mettre en oeuvre cette approche, notre idée de base est d'avoir des variantes des composants de notre service Web. Ces variantes réagissent et sont remplacées quand l'une d'entre eux est compromise par une attaque. Plus précisément, les contributions proposées sont les suivantes :

Le monitorage basé sur les risques (chapitre 3

Nous nous sommes appuyés sur la boucle traditionnelle de gestion des risques pour en ressortir une approche de monitorage basée sur les risques. Nous faisons l'hypothèse que la détection et la prévention d'attaque requièrent une bonne connaissance des risques auxquels les services Web peuvent être confrontés. Il est donc opportun voire obligatoire d'inclure la gestion des risques lorsqu'on veut surveiller ces services dans le but principal de réduire la probabilité des attaques ou des comportements malveillants. Cette méthodologie possède quatre piliers fondamentaux :

- l'identification des parties ou des éléments dont la compromission peut être préjudiciables au bon fonctionnement du sytème ou du service;
- l'analyse des menaces et des vulnérabilités pour évaluer les potentielles failles du système;
- l'analyse des risques pour évaluer la probabilité que ces vulnérabilités puissent conduire à des attaques effectives,
- le monitorage du système pour détecter l'occurrence de potentielles attaques;
- le déploiement de mécanismes de remédiation pour permettre au système de continuer à fournir son service en présence d'attaques.

Cette méthodologie a été appliquée tout au long de cette thèse. Elle a donc été mise en oeuvre dans les contributions ci-dessous suivantes.

Tolérance aux attaques basée sur la diversification (chapitre 4)

Premièrement, nous avons investigué la tolérance aux attaques dès les premières phases de conception de nos services. Nous avons proposé un modèle du système exprimé sous forme d'une machine à états étendue. A partir de ce modèle, nous dérivons des variantes fonctionnellement équivalentes. Nous nous assurons que ces modèles sont validés notamment en les testant. De ces modèles, nous dérivons les implémentations correspondantes. Lorsqu'une

attaque est détectée par le monitorage, nous changeons dynamiquement l'implémentation. Ce qui signifie que l'attaquant est confronté à un nouveau système dont il n'a pas forcément connaissance des vulnérabilités de celui-ci. Nous avons proposé comme exemple, une application Web d'authentification qui permet en se basant sur un mécanisme de mots de passe plus simple de passer à des mécanismes plus robustes mais plus contraignants pour l'utilisateur tels que l'authentification à deux facteurs. Nous avons évalué cette approche en injectant une attaque de type brute-force dans l'application d'authentification. Ces expérimentations ont montrée que l'approche permettait de tolérer ce type d'attaque. Les limites de cette première approche résidaient dans la difficulté à dériver automatiquement les variantes.

C'est pourquoi dans une approche complémentaire nous avons étendu la première. Cette seconde approche vise essentiellement à répondre aux insuffisances de la première approche. L'idée de base est la même, mais la mise en oeuvre est un brin différente. Notre intuition résidait dans le fait qu'avoir divers points d'échec permettait d'améliorer la sécurité puisque cela induit la tolérance à une grande variété d'attaque. Nous avons donc dans cette nouvelle approche inclut la diversification dans toutes les étapes; de la phase de modélisation à la phase de déploiement des services Web. Nous avons illustré cette approche par un service Web qui simplifie les opérations de gestion dans un hôpital . Comme dans la première méthode, nous définissons un modèle du service. Cette fois ci nous avons choisi un modèle permettant d'exprimer les parties variables de notre service. Ces points de variation (style d'encodage, language, type d'encodage) assurent la diversification des variantes au niveau des implémentations. Après cette étape de spécification, la dérivation des variantes se fait d'une manière semi-automatique. Pour ajouter davantage de diversification, nous avons diversifié les exécutables de nos services. Nous avons évalué l'approche avec une attaque de type déni de services. Cela montre que l'approche induit peu de surcouts de performance pour une réaction rapide à ces attaques.

Tolérance aux attaques basée sur le mécanisme de réflexivité (chapitre 5)

Dans cette contribution, nous avons abordé la tolérance aux attaques des services Web d'une manière différente de celles des deux approches que nous avons mentionnées plus haut. En effet dans ces approches, la détection se faisait pour les attaques externes aux services. De plus, leur capacité de tolérance était complètement décrite à la phase de conception des services. Ici, nous proposons une tolérance qui couvre un large spectre d'attaques en incluant les attaques dites internes qui sont réputées difficiles à détecter. Cette nouvelle contribution intègre les techniques de réflexivité et de monitorage pour une détection efficiente de ce genre d'attaques. La réflexivité est une technique qui permet aux programmes de s'analyser, et d'ajuster leur comportement dynamiquement. Nous considérons que le logiciel du client se trouve dans un environnement sûr. Les potentielles attaques qui peuvent survenir sont seulement celles qui proviennent de l'intérieur. En utilisant les techniques de réflexivité, nous déterminons les hash de toutes les méthodes de notre service Web. Toute déviation lorsque le système est en cours de fonctionnement traduit une comportement malicieux. Ce comportement malicieux peut être une modification du code par un tiers ou une infection

de virus. Nous stockons les informations suivantes : Date, heure, opération, hash, origine dans nos fichiers de journalisation. Toutes les requêtes contiennent deux traces (requête et réponse) dans ces fichiers de journalisation. Nous avons développé un nouveau plugin permettant à notre outil de monitoring MMT de ce type d'attaque. Nous avons évalué cette approche en déployant un service Web modulaire. Nous avons injecté un virus dans ce service et nous avons en particulier comparé la détection de notre approche à celle des antivirus classiques du marché. Ces expériences ont montré que notre approche était plus efficace dans la détection de ce virus. Ce qui montre le bien fondé de réflexivité pour détecter ce genre d'attaque.

Une approche de tolérance pour les applications basées sur les services Web et déployées dans le cloud chapitre (6)

Nous avons proposé une méthodologie pour la tolérance aux applications dans le cloud qui sont construites à base de services Web. Pour ce faire, nous avons étendu un cadre formel de vérification et de test de services Web en incorporant les contributions précédentes. Dans cette approche, les services Web sont modélisés sous la forme de chorégraphies de services, chacun exécutant une fonction clé de la collaboration. Une chorégraphie est un moyen de décrire une collaboration point à point entre les services et a l'avantage de ne pas avoir un acteur central de décision tels que rencontrés dans les systèmes distribués actuels. Une fois que cette chorégraphie est vérifiée formellement par exemple pour s'assurer qu'elle est réalisable, elle est projetée sur les différents pairs. Nous vérifions également que ces modèles locaux sont conformes à la chorégraphie globale. Par ailleurs, à partir de ces modèles locaux, nous dérivons les squelettes des interfaces du service ainsi que des implémentations en divers langages cibles(Java, Python et PHP) afin que les programmeurs puissent choisir le langage qui leur sied et compléter l'implémentation. Nous avons pour ce faire, proposer un langage intermédiaire qui permet de produire ces squelettes à partir d'une description textuelle des chorégraphies. Après cette phase de diversification, une parmi ces implémentations est choisie. Nous disposons de deux manières de monitorage de l'application. La première manière consiste à tester les implémentations avant qu'elles ne soient déployées. Cela à pour but de détecter les fautes de conception. La seconde manière consiste à utiliser les techniques de réflexivité énoncées dans la section précédente. Cela a pour but de détecter les comportements anormaux tels que les attaques. Nous réagissons également en utilisant la réflexivité. En conclusion, le fait d'ajouter ces mécanismes de détection et de réaction en ligne à cette chorégraphie vérifiée formellement assure une tolérance complète aux attaques.

A.2.3 Publications

Les principaux résultats de cette thèse ont été publiées dans des conférence internationales avec comité de lecture ainsi que lors de séminaires nationaux suivants :

Workshops

1. G. Ouffoué, A. M. Ortiz, A. R. Cavalli, W. Mallouli, J. Domingo-Ferrer, D. Sánchez,and F. Zaïdi. *Intrusion detection and attack tolerance for cloud environments : The clarus approach.* In 2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW), pages 61–66. IEEE, 2016.22.

2. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *Model-based attack tolerance.* In 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), pages 68–73. IEEE, 2017.

Conférence internationales

1. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *How web services can be tolerant to intruders through diversification?* ICWS 2017 24th IEEE International Conference on Web Services, pp.436 - 443, 2017,

2. G. Ouffoué, F. Zaïdi, A. R. Cavalli, and M. Lallali. *An Attack-Tolerant Framework for Web Services.* In 2017 IEEE International Conference on Services Computing (SCC), pages. IEEE, 2017.

3. Ana R. Cavalli, Antonio M. Ortiz, Georges Ouffoué, Cesar A. Sanchez, and Fatiha Zaïdi. *Design of a Secure Shield for Internet-based Services using Software Reflection.* ICWS 2018 International Conference on Web Services,

Présentations

1. *How web services can be tolerant to intruders through diversification?* lors Journées du GDR MTV2/MFDL track. Dec 2017.

A.2.4 Posters

1. *Attack Tolerant Cloud* lors de la conférence nationale en sécurité C&ESAR 2017.

Bibliography

[Rsa, 1977] (1977). RSA algorithm. https://people.csail.mit.edu/rivest/Rsapaper.pdf.

[api, 2016a] (2016a). The top cloud computing threats and vulnerabilities in an enterprise environment. http://www.cloudcomputing-news.net/news/2014/nov/21/top-cloud-computing-threats-and-vulnerabilities-enterprise-environment/.

[api, 2016b] (2016b). Vulnerable apis continue to pose threat to cloud. http://www.darkreading.com/risk/vulnerable-apis-continue-to-pose-threat-to-cloud/d/d-id/1138983.

[Affonso and Nakagawa, 2013] Affonso, F. J. and Nakagawa, E. Y. (2013). A reference architecture based on reflection for self-adaptive software. In *2013 VII Brazilian Symposium on Software Components, Architectures and Reuse*, pages 129–138.

[Ahamed et al., 2013] Ahamed, F., Shahrestani, S., and Ginige, A. (2013). Cloud computing: Security and reliability issues. pages 1–12.

[Allier et al., 2015] Allier, S., Barais, O., Baudry, B., Bourcier, J., Daubert, E., Fleurey, F., Monperrus, M., Song, H., and Tricoire, M. (2015). Multi-tier diversification in web-based software applications. *IEEE Software*, 32(1):83–90.

[Anagnostakis et al., 2002] Anagnostakis, K., Ioannidis, S., Miltchev, S., Greenwald, M., Smith, J., and Ioannidis, J. (2002). Efficient packet monitoring for network management. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 423–436.

[Arsenault et al., 2007] Arsenault, D., Sood, A., and Huang, Y. (2007). Secure, resilient computing clusters: Self-cleansing intrusion tolerance with hardware enforced security (scit/hes). In *The Second International Conference on Availability, Reliability and Security (ARES'07)*, pages 343–350.

[Attiogbe, 2007] Attiogbe, C. (2007). *Contributions aux approches formelles de développement de logiciels : intégration de méthodes formelles et analyse multifacette.*

[Aung et al., 2005] Aung, K., Park, K., and Park, J. S. (2005). A rejuvenation methodology of cluster recovery. In *CCGrid 2005. IEEE International Symposium on Cluster Computing and the Grid, 2005.*, volume 1, pages 90–95 Vol. 1.

[Baudry and Monperrus, 2015] Baudry, B. and Monperrus, M. (2015). The multiple facets of software diversity: Recent developments in year 2000 and beyond. *ACM Comput. Surv.*, 48(1):16:1–16:26.

[Bezdek and Hathaway, 2002] Bezdek, J. C. and Hathaway, R. J. (2002). Vat: a tool for visual assessment of (cluster) tendency. In *Neural Networks, 2002. IJCNN '02. Proceedings of the 2002 International Joint Conference on*, volume 3, pages 2225–2230.

[Bozga et al., 2002] Bozga, M., Graf, S., and Mounier, L. (2002). If-2.0: A validation environment for component-based real-time systems. In *Proceedings of the 14th International Conference on Computer Aided Verification*, pages 343–348. Springer-Verlag.

[Buczak and Guven, 2016] Buczak, A. L. and Guven, E. (2016). A survey of data mining and machine learning methods for cyber security intrusion detection. *IEEE Communications Surveys Tutorials*, 18(2):1153–1176.

[Cao et al., 2010] Cao, T. D., Phan-Quang, T. T., Felix, P., and Castanet, R. (2010). Automated runtime verification for web services. In *2010 IEEE International Conference on Web Services*, pages 76–82.

[Cavalli et al., 2010] Cavalli, A. R., Cao, T., Mallouli, W., Martins, E., Sadovykh, A., Salva, S., and Zaïdi, F. (2010). Webmov: A dedicated framework for the modelling and testing of web services composition. In *IEEE International Conference on Web Services, ICWS 2010, Miami, Florida, USA, July 5-10, 2010*, pages 377–384. IEEE Computer Society.

[Cavalli et al., 2018] Cavalli, A. R., Ortiz, A. M., Ouffoué, G., Sanchez, C. A., and Zaïdi, F. (2018). Design of a secure shield for internet and web-based services using software reflection. In *Web Services – ICWS 2018*. Springer International Publishing.

[Cert, 2017] Cert (2017). "wannacry". http://cert-mu.govmu.org/English/Documents/WhitePapers/WhitePaper-The20WannaCryRansomwarAttack.pdf.

[Chen et al., 2009] Chen, L., Li, Z., Gao, C., and Liu, L. (2009). Dynamic forensics based on intrusion tolerance. In *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, pages 469–473.

[Chen and Hu, 2002] Chen, T. M. and Hu, L. (2002). Internet performance monitoring. In *Proceedings of the IEEE*, pages 1592 –1603.

[Cloudtp, 2018] Cloudtp (2018). Cloud-ready application development: Step-by-step guide. https://www.cloudtp.com/doppler/5-steps-building-cloud-ready-application-architecture/.

[Collberg et al., 2012] Collberg, C., Martin, S., Myers, J., and Nagra, J. (2012). Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 319–328. ACM.

[Constable et al., 2011] Constable, R., Mark, M. B., and Robbert, V. R. (2011). Investigating Correct-by-Construction Attack-Tolerant Systems. Technical report, Department of Computer Science, Cornell University.

[Cousot and Cousot, 1977] Cousot, P. and Cousot, R. (1977). Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, New York, NY.

[Curtis, 2000] Curtis, J. (2000). Passive measurement. http://wand.cs.waikato.ac.nz/old/wand/publications/jamie_420/final/node9.html.

[Deswarte et al., 2002] Deswarte, Y., Abghour, N., Nicomette, V., and Powell, D. (2002). Intrusion-tolerant authorization for internet applications. In *Supplement of the IEE/IFIP International Conferencce on Dependable Systems and Networks*, DSN' 2002.

[Deswarte and Powell, 2004] Deswarte, Y. and Powell, D. (2004). Intrusion tolerance for internet applications. In *Building the Information Society*, pages 241–256. Springer US.

[Dillon et al., 2010] Dillon, T., Wu, C., and Chang, E. (2010). Cloud computing: Issues and challenges. In *2010 24th IEEE International Conference on Advanced Information Networking and Applications*, pages 27–33.

[Dougblack, 2017] Dougblack (2017). Metaclasses. https://dougblack.io/words/metaclasses.html.

[Engelen and Gallivan, 2002] Engelen, R. A. V. and Gallivan, K. A. (2002). The gsoap toolkit for web services and peer-to-peer computing networks. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 128–128.

[Etchevers, 2012] Etchevers, X. (2012). *Déploiement d'applications patrimoniales en environnements de type informatique dans le nuage*. PhD thesis, Université de Grenoble.

[Ficco and Rak, 2011] Ficco, M. and Rak, M. (2011). Intrusion tolerant approach for denial of service attacks to web services. In *Proceedings of the 2011 First International Conference on Data Compression, Communications and Processing*, CCP '11, pages 285–292. IEEE Computer Society.

[Forman et al., 2004] Forman, R., Forman, N., and Ibm, J. (2004). Java reflection in action.

[Franz, 2010] Franz, M. (2010). E unibus pluram: Massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 7–16. ACM.

[Furtado et al., 2014] Furtado, T., Francesquini, E., Lago, N., and Kon, F. (2014). A middleware for reflective web service choreographies on the cloud. In *Proceedings of the 13th Workshop on Adaptive and Reflective Middleware*, ARM '14, pages 9:1–9:6. ACM.

[Gabbay et al., 1994] Gabbay, D. M., Hodkinson, I., and Reynolds, M. (1994). *Temporal Logic Mathematical Foundations and Computational Aspects*. Clarendon Press.

[Ganger et al., 2001] Ganger, G. R., Khosla, P. K., Bakkaloglu, M., Bigrigg, M. W., Goodson, G. R., Oguz, S., Pandurangan, V., Soules, C. A. N., Strunk, J. D., and Wylie, J. J. (2001). Survivable storage systems. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 184–195 vol.2.

[Generowicz et al., 2004] Generowicz, J., Lavrijsen, W. T., Marino, M., and Mato, P. (2004). Reflection-based python-c++ bindings. *Lawrence Berkeley National Laboratory.*

[Georgiev et al., 2012] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. (2012). The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security*, pages 38–49.

[Gheyas and Abdallah, 2016] Gheyas, I. A. and Abdallah, A. E. (2016). Detection and prediction of insider threats to cyber security: a systematic literature review and meta-analysis. *Big Data Analytics*, 1:6.

[Gomes et al., 2015] Gomes, R., Lima, J., Rocha, F. C. R. D., and Georgantas, N. (2015). A Model-Based Approach to Pragmatic Service Choreography Deployment. In *Proceedings of Second Workshop on Seamless Adaptive Multi-cloud Management of Service-based Applications.*

[Hansche et al., 2003] Hansche, S., Berti, J. C., and Hare, C. (2003). Official (isc) 2 guide to the cissp exam.

[Hennessy and Lin, 1995] Hennessy, M. and Lin, H. (1995). Symbolic bisimulations. *Theor. Comput. Sci.*, 138(2):353–389.

[Hwang et al., 2009] Hwang, I., Lallali, M., Cavalli, A. R., and Verchère, D. (2009). Modeling, validation, and verification of PCEP using the IF language. In *Formal Techniques for Distributed Systems, Joint 11th IFIP WG 6.1 International Conference FMOODS 2009 and 29th IFIP WG 6.1 International Conference FORTE 2009, Lisboa, Portugal, June 9-12, 2009. Proceedings*, volume 5522, pages 122–136. Springer.

[Ibrahim, 2016] Ibrahim, H. (2016). *SAT-Based Diagnosability and Predictability Analysis in Centralized and Distributed Discrete Event Systems.* PhD thesis, Université Paris-Saclay.

[Karande et al., 2011] Karande, V., Vishal, M., Pais, M., and Alwyn, R. (2011). A framework for intrusion tolerance in cloud computing. In *Advances in Computing and Communications*, volume 193, pages 386–395.

[Kazim and Zhu, 2015] Kazim, M. and Zhu, S. Y. (2015). A survey on top security threats in cloud computing. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 6.

[Knight et al., 2001] Knight, J., Heimbigner, D., Wolf, A. L., Carzaniga, A., Hill, J., Devanbu, P. T., and Gertz, M. (2001). The willow architecture : Comprehensive survivability for large-scale distributed applications.

[Kocher et al., 1999] Kocher, P., Jaffe, J., and Jun, B. (1999). Differential power analysis. In *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*, pages 388–397. Springer-Verlag.

[Kondratyeva et al., 2013] Kondratyeva, O., Kushik, N., Cavalli, A. R., and Yevtushenko, N. (2013). Evaluating quality of web services: A short survey. In *2013 IEEE 20th International Conference on Web Services, Santa Clara, CA, USA, June 28 - July 3, 2013*, pages 587–594. IEEE Computer Society.

[Kopp and Breitenbücher, 2017] Kopp, O. and Breitenbücher, U. (2017). Choreographies are Key for Distributed Cloud Application Provisioning. In *ZEUS*, volume 1826 of *CEUR Workshop Proceedings*, pages 67–70. CEUR-WS.org.

[Kuyoro et al., 2012] Kuyoro, S., Ibikunle, F., and Okolie, S. (2012). Security issues in web services.

[La, 2016] La, V. H. (2016). *Security Monitoring for Network Protocols and Applications*. PhD thesis, Université Paris-Saclay.

[Lala, 2003] Lala, J. (2003). Foundations of intrusion tolerant systems. In *Foundations of Intrusion Tolerant Systems, 2003 [Organically Assured and Survivable Information Systems]*, pages i–vii.

[Lin, 2016] Lin (2016). Detecting insider security threats. https://content.pivotal.io/blog/a-data-science-approach-to-detecting-insider-security-threats.

[M. Raju, 2014] M. Raju, B. L. (2014). Survey about cloud computing threats. *(IJCSIT) International Journal of Computer Science and Information Technologies*, 5:384–389.

[Madan and Trivedi, 2004] Madan, B. B. and Trivedi, K. S. (2004). Security modeling and quantification of intrusion tolerant systems using attack-response graph. *Journal of High Speed Networks*, 13(4):297–308.

[Maes, 1987] Maes, P. (1987). Concepts and experiments in computational reflection. *SIGPLAN Not.*, 22(12):147–155.

[Mateescu et al., 2012] Mateescu, R., Poizat, P., and Salaün, G. (2012). Adaptation of service protocols using process algebra and on-the-fly reduction techniques. *IEEE Trans. Software Eng.*, 38(4):755–777.

[Meixner et al., 2016] Meixner, C., Develder, C., Tornatore, M., and Mukherjee, B. (2016). A survey on resiliency techniques in cloud computing infrastructures and applications. *IEEE Communications Surveys and Tutorials*, 18(3):2244–2281.

[Mell and Grance, 2001] Mell, P. and Grance, T. (2001). Nist special publication 800-145 the definition of cloud computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf.

[Mishra et al., 2017] Mishra, P., Pilli, E. S., Varadharajan, V., and Tupakula, U. K. (2017). Intrusion detection techniques in cloud environment: A survey. *J. Network and Computer Applications*, 77:18–47.

[Morales et al., 2010] Morales, G., Maag, S., Cavalli, A. R., Mallouli, W., de Oca, E. M., and Wehbi, B. (2010). Timed extended invariants for the passive testing of web services. In *IEEE International Conference on Web Services, ICWS 2010, Miami, Florida, USA, July 5-10, 2010*, pages 592–599. IEEE Computer Society.

[Morin et al., 2009] Morin, B., Barais, O., Jezequel, J., Fleurey, F., and Solberg, A. (2009). Models@ run.time to support dynamic adaptation. *Computer*, 42(10):44–51.

[Moy and Wallenburg, 2010] Moy, Y. and Wallenburg, A. (2010). Tokeneer: Beyond formal program verification. In *2010 International Congress on Embedded Real Time Software and System*.

[Nguyen, 2013] Nguyen, H. N. (2013). *Une Approche Symbolique pour la Vérification et le Test des Chorégraphies de Services*. PhD thesis, Université Paris-Sud.

[Nguyen et al., 2013] Nguyen, H. N., Poizat, P., and Zaïdi, F. (2013). Automatic skeleton generation for data-aware service choreographies. In *IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, Pasadena, CA, USA, November 4-7, 2013*, pages 320–329. IEEE Computer Society.

[Nguyen et al., 2014] Nguyen, H. N., Zaïdi, F., and Cavalli, A. R. (2014). A framework for distributed testing of timed composite systems. In *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*, pages 47–54. IEEE.

[Nguyen et al., 2016] Nguyen, H. N., Zaïdi, F., and Cavalli, A. R. (2016). Effectively testing of timed composite systems using test case prioritization. In Gou, J., editor, *The 28th International Conference on Software Engineering and Knowledge Engineering, SEKE 2016, Redwood City, San Francisco Bay, USA, July 1-3, 2016.*, pages 408–413. KSI Research Inc. and Knowledge Systems Institute Graduate School.

[Nguyen and Sood, 2010] Nguyen, Q. and Sood, A. (2010). A comparison of intrusion-tolerant system architectures. *IEEE Security & Privacy*, 9(4):24 – 31.

[Nicomette et al., 2011] Nicomette, V., Powell, D., Deswarte, Y., Abghour, N., and Zanon, C. (2011). Intrusion-tolerant fine-grained authorization for internet applications. *Journal of Systems Architecture*, 57(4):441 – 451.

[Oasis, 2006] Oasis (2006). Oasis web services security (wss) tc. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss.

[Obelheiro et al., 2006] Obelheiro, R. R., Bessani, A. N., Lung, L. C., and Correia, M. (2006). How practical are intrusion-tolerant distributed systems? Technical report, Departamento de Informática Faculdade de Ciências da Universidade de Lisboa Portugal.

[O'Brien et al., 2003] O'Brien, D., Smith, R., Kappel, T., and Bitzer, C. (2003). Intrusion tolerance via network layer controls. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 1, pages 90–96 vol.1.

[Ouffoué et al., 2016] Ouffoué, G., Ortiz, A. M., Cavalli, A. R., Mallouli, W., Domingo-Ferrer, J., Sánchez, D., and Zaïdi, F. (2016). Intrusion detection and attack tolerance for cloud environments: The clarus approach. In *2016 IEEE 36th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, pages 61–66. IEEE.

[Ouffoué et al., 2017] Ouffoué, G., Zaïdi, F., Cavalli, A. R., and Lallali, M. (2017). Model-based attack tolerance. In *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*, pages 68–73. IEEE.

[Ouffoué et al., 2017a] Ouffoué, G., Zaïdi, F., Cavalli, A. R., and Lallali, M. (2017a). How web services can be tolerant to intruders through diversification. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 436–443.

[Ouffoué et al., 2017b] Ouffoué, G. L. A., Zaïdi, F., Cavalli, A. R., and Lallali, M. (2017b). An attack-tolerant framework for web services. In *2017 IEEE International Conference on Services Computing (SCC)*, pages 503–506.

[Partha et al., 2006] Partha, P., Rubel, P., Atighetchi, M., Webber, F., Sanders, W., Seri, M., Ramasamy, H., Lyons, J., Courtney, T., Agbaria, A., Cukier, M., Gossett, J., and Keidar, I. (2006). An architecture for adaptive intrusion-tolerant applications: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.*, 36(11-12):1331–1354.

[Pen et al., 2014] Pen, K., Huang, C., Wang, P., and Hsu, C. (2014). Enhanced n-version programming and recovery block techniques for web service systems. In *Proceedings of the International Workshop on Innovative Software Development Methodologies and Practices*, pages 11–20. ACM.

[Prasad et al., 2013] Prasad, M. R., Ramavathu, L., and Bapuji (2013). Cloud computing : Research issues and implications. 2:134–140.

[Qiu et al., 2007] Qiu, Z., Zhao, X., Cai, C., and Yang, H. (2007). Towards the Theoretical Foundation of Choreography. In *Proc. of WWW'07*.

[Raj and Varghese, 2011] Raj, S. and Varghese, G. (2011). Analysis of intrusion-tolerant architectures for web servers. In *2011 International Conference on Emerging Trends in Electrical and Computer Technology*, pages 998–1003.

[Reiser and Kapitza, 2007] Reiser, H. P. and Kapitza, R. (2007). Hypervisor-based efficient proactive recovery. In *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 83–92.

[Reynolds et al., 2003] Reynolds, J. C., Just, J., Clough, L., and Maglich, R. (2003). On-line intrusion detection and attack prevention using diversity, generate-and-test, and generalization. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pages 8 pp.–.

[Ristenpart et al., 2009] Ristenpart, T., Tromer, E., Shacham, H., and Savage, S. (2009). 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*, pages 199–212. ACM.

[Roiser and Mato, 2005] Roiser, S. and Mato, P. (2005). The seal c++ reflection system.

[Ronacher, 2017] Ronacher, A. (2017). The flask framework. http://flask.pocoo.org.

[Sadegh and Azgomi, 2015] Sadegh, B. and Azgomi, M. A. (2015). A new architecture for intrusion-tolerant web services based on design diversity techniques. *Journal of Information Systems and Telecommunication (JIST)*.

[Saha, 2007] Saha, G. K. (2007). Understanding dependable computing concepts. *Ubiquity*, 2007(November):1:1–1:1.

[Saidane et al., 2009] Saidane, A., Nicomette, V., and Deswarte, Y. (2009). The design of a generic intrusion-tolerant architecture for web servers. *IEEE Transactions on Dependable and Secure Computing*, 6(1):45–58.

[Salem et al., 2008] Salem, M., Hershkop, S., and Stolfo, S. (2008). *A Survey of Insider Attack Detection Research*, pages 69–90. Springer US.

[Sánchez and Domingo-Ferrer, 2015] Sánchez, D. and Domingo-Ferrer, J. (2015). Clarus - a framework for user centred privacy and security in the cloud. In *Position paper at Cloudscape VII*.

[Sans, 2017] Sans (2017). Glossaries. http://www.sans.org/security-resources/glossary-of-terms/.

[Schneier, 1999] Schneier, B. (1999). Modelling security threats. https://www.schneier.com/academic/archives/1999/12/attack_trees.html.

[Science, 2013] Science, U. C. (2013). System administration database. “blocking brute force attacks". http://www.cs.virginia.edu/~csadmin/gen_support/brute_force.php,.

[Sharma et al., 2011] Sharma, R., Sood, M., and Sharma, D. (2011). Modeling cloud saas with soa and mda. In *Advances in Computing and Communications*, pages 511–518. Springer Berlin Heidelberg.

[Singhal et al., 2007] Singhal, A., Winograd, T., and Scarfone, K. (2007). Recommendations of the national institute of standards and technology.

[Sliti et al., 2009] Sliti, M., Hamdi, M., and Boudriga, N. (2009). Intrusion-tolerant framework for heterogeneous wireless sensor networks. In *2009 IEEE/ACS International Conference on Computer Systems and Applications*, pages 633–636.

[Sousa et al., 2008] Sousa, P., Bessani, A., Neves, N. F., and Obelheiro, R. (2008). The forever service for fault/intrusion removal. In *Proceedings of the 2Nd Workshop on Recent Advances on Intrusiton-tolerant Systems*, WRAITS '08, pages 5:1–5:6. ACM.

[Sousa et al., 2007a] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2007a). Resilient intrusion tolerance through proactive and reactive recovery. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 373–380.

[Sousa et al., 2007b] Sousa, P., Bessani, A. N., Correia, M., Neves, N. F., and Verissimo, P. (2007b). Resilient intrusion tolerance through proactive and reactive recovery. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, pages 373–380.

[Spinellis, 2000] Spinellis, D. (2000). Reflection as a mechanism for software integrity verification. *ACM Transactions on Information and System Security*, 3(1):51–62.

[Stankovic and Strigini, 2009] Stankovic, V. and Strigini, L. (2009). A survey on online monitoring approaches of computer-based systems.

[Stroud et al., 2004] Stroud, R., Welch, I., Warne, J., and Ryan, P. (2004). A qualitative analysis of the intrusion-tolerance capabilities of the maftia architecture. In *International Conference on Dependable Systems and Networks ICDSN*, pages 453–461.

[Studnia et al., 2012] Studnia, I., Alata, E., Deswarte, Y., Kaâniche, M., and Nicomette, V. (2012). Survey of security problems in cloud computing virtual machines. *Computer and Electronics Security Applications Rendez-vous (C&ESAR). Cloud and security: threat or opportunity*, 5:61–74.

[Sun, 2016] Sun (2016). Behaviour in accessing a source code repository. http://www.arcserve.com/us/~/media/Files/AboutUs/CATX/analysis-of-user-behaviour-in-accessing-a-source-code-repository.pdf.

[Suresh et al., 2016] Suresh, A. T., Yu, F. X., McMahan, H. B., and Kumar, S. (2016). Distributed mean estimation with limited communication. *CoRR*, abs/1611.00429.

[Tipton, 2009] Tipton, H. F. (2009). Official (isc) 2 guide to the cissp cbk.

[Tittle et al., 2006] Tittle, E., Stewart, J. M., and Chapple, M. (2006). Cissp: Certified information systems security professional study guide.

[Totel et al., 2006] Totel, E., Majorczyk, F., and Mé, L. (2006). Cots diversity based intrusion detection and application to web servers. In *Proceedings of the 8th International Conference on Recent Advances in Intrusion Detection*, RAID'05, pages 43–62. Springer-Verlag.

[Valdes et al., 2002] Valdes, A., Almgren, M., Cheung, S., Deswarte, Y., Dutertre, B., Levy, J., Saïdi, H., Stavridou, V., and ás E. Uribe, T. (2002). An architecture for an adaptive intrusion-tolerant server. In *Security Protocols, 10th International Workshop, Cambridge, UK, April 17-19, 2002, Revised Papers*, pages 158–178.

[Valdes et al., 2004] Valdes, A., Almgren, M., Cheung, S., Deswarte, Y., Dutertre, B., Levy, J., Saïdi, H., Stavridou, V., and Uribe, T. (2004). An architecture for an adaptive intrusion-tolerant server. In *Security Protocols*, pages 158–178. Springer Berlin Heidelberg.

[Veríssimo et al., 2003] Veríssimo, P., Neves, N., and Correia, M. (2003). Intrusion-tolerant architectures: Concepts and design. In *Architecting Dependable Systems*, pages 3–36, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Verissimo et al., 2006] Verissimo, P. E., Neves, N. F., Cachin, C., Poritz, J., Powell, D., Deswarte, Y., Stroud, R., and Welch, I. (2006). Intrusion-tolerant middleware: the road to automatic security. *IEEE Security Privacy*, 4(4):54–62.

[Wang et al., 2003] Wang, F., Raghavendra, U., and Killian, C. (2003). Analysis of techniques for building intrusion tolerant server systems. In *IEEE Military Communications Conference (MILCOM)*, volume 2, pages 729–734.

[Wang and Upppalli, 2003] Wang, F. and Upppalli, R. (2003). Sitar: a scalable intrusion-tolerant architecture for distributed services - a technology summary. In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 2, pages 153–155 vol.2.

[Weinstein and Lepanto, 2003] Weinstein, W. and Lepanto, J. (2003). Camouflage of network traffic to resist attack (contra). In *Proceedings DARPA Information Survivability Conference and Exposition*, volume 2, pages 126–127 vol.2.

[Xu et al., 2016] Xu, H., Zhou, Y., and Lyu, M. (2016). N-version obfuscation. In *Proceedings of the 2Nd ACM International Workshop on Cyber-Physical System Security*, pages 22–33. ACM.

[Yan et al., 2016] Yan, Q., Yu, F. R., Gong, Q., and Li, J. (2016). Software-defined networking (sdn) and distributed denial of service (ddos) attacks in cloud computing environments: A survey, some research issues, and challenges. *IEEE Communications Surveys and Tutorials*, 18(1):602–622.

[Yang and Zhang, 2012] Yang, X. and Zhang, H. (2012). Cloud computing and soa convergence research. In *2012 Fifth International Symposium on Computational Intelligence and Design*, volume 1, pages 330–335.

[Yarom and Falkner, 2014] Yarom, Y. and Falkner, K. (2014). Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732.

[Zangrilli and Lowekamp, 2004] Zangrilli, M. and Lowekamp, B. B. (2004). Using passive traces of application traffic in a network monitoring system. In *Proceedings of the 13th IEEE International Symposium on, High performance Distributed Computing*, pages 77 – 86.

[Zhang et al., 2005] Zhang, T., Zhuang, X., and Pande, S. (2005). Building intrusion-tolerant secure software. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 255–266. IEEE Computer Society.

[Zhang et al., 2012] Zhang, Y., Juels, A., Reiter, M., Michael, K., and Ristenpart, T. (2012). Cross-vm side channels and their use to extract private keys. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, pages 305–316. ACM.

[Zhang et al., 2014] Zhang, Y., Juels, A., Reiter, M. K., and Ristenpart, T. (2014). Cross-tenant side-channel attacks in paas clouds. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM.

[Zhang and Reiter, 2013] Zhang, Y. and Reiter, M. K. (2013). Düppel: retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *Proceedings of the 2013 ACM SIGSAC conference on Computer 's communications security*, pages 827–838. ACM.

ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Tolérance aux attaques pour les applications orientées services Web dans le cloud

Mots clés : Tolérance aux attaques, services Web, supervision, cloud, tests passifs, réflexivité

Résumé :
Les services Web permettent la communication de systèmes hétérogènes sur le Web. Ces facilités font que ces services sont particulièrement adaptés au déploiement dans le cloud. Les efforts de formalisation et de vérification permettent d'améliorer la confiance dans les services Web, néanmoins des problèmes tels que la haute disponibilité et la sécurité ne sont pas entièrement pris en compte. Par ailleurs, les services Web déployés dans une infrastructure cloud héritent des vulnérabilités de cette dernière. En raison de cette limitation, ils peuvent être incapables d'exécuter parfaitement leurs tâches. Dans cette thèse, nous pensons qu'une bonne tolérance nécessite un monitoring constant et des mécanismes de réaction fiables. Nous avons donc proposé une nouvelle méthodologie de monitoring tenant compte des risques auxquels peuvent être confrontés nos services. Pour mettre en œuvre cette méthodologie, nous avons d'abord développé une méthode de tolérance aux attaques qui s'appuie sur la diversification au niveau modèle. On définit un modèle du système puis on dérive des variantes fonctionnellement équivalents qui remplaceront ce dernier en cas d'attaque. Pour ne pas dériver manuellement les variants et pour augmenter le niveau de diversification nous avons proposé une deuxième méthode complémentaire. Cette dernière consiste toujours à avoir des variants de nos services; mais contrairement à la première méthode, nous proposons un modèle unique avec des implantations différentes tant au niveau des interfaces, du langage qu'au niveau des exécutables. Par ailleurs, pour détecter les attaques internes, nous avons proposé un mécanisme de détection et de réaction basé sur la réflexivité. Lorsque le programme tourne, nous l'analysons pour détecter les exécutions malveillantes. Comme contre-mesure, on génère de nouvelles implantations en utilisant toujours la réflexivité. Pour finir, nous avons étendu un environnement formel et outillé de services Web en y incorporant de manière cohérente tous ces mécanismes. L'idée est de pouvoir combiner ces différentes méthodes afin de tirer profit des avantages de chacune d'elle. Nous avons validé toute cette approche par des expériences réalistes.

Title : Attack tolerance for services-based applications in the Cloud

Keywords : Attack tolerance, Web services, monitoring, cloud, passive tests, reflection

Abstract : Web services allow the communication of heterogeneous systems on the Web. These facilities make them particularly suitable for deploying in the cloud. Although research on formalization and verification has improved trust in Web services, issues such as high availability and security are not fully addressed. In addition, Web services deployed in cloud infrastructures inherit their vulnerabilities. Because of this limitation, they may be unable to perform their tasks perfectly. In this thesis, we claim that a good tolerance requires attack detection and continuous monitoring on the one hand; and reliable reaction mechanisms on the other hand. We therefore proposed a new formal monitoring methodology that takes into account the risks that our services may face. To implement this methodology, we first developed an approach of attack tolerance that leverages model-level diversity. We define a model of the system and derive more robust functionally equivalent variants that can replace the first one in case of attack. To avoid manually deriving the variants and to increase the level of diversity, we proposed a second complementary approach. The latter always consists in having different variants of our services; but unlike the first, we have a single model and the implementations differ at the language, source code and binaries levels. Moreover, to ensure detection of insider attacks, we investigated a new detection and reaction mechanism based on software reflection. While the program is running, we analyze the methods to detect malicious executions. When these malicious activities are detected, using reflection again, new efficient implementations are generated as countermeasure. Finally, we leveraged a formal Web service testing framework by incorporating these complementary mechanisms in order to take advantage of the benefits of each of them. We validated our approach with realistic experiments.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France