UNIVERSITÉ DU LUXEMBOURG

PhD-FSTC-2013-36
The Faculty of Sciences, Technology and Communication

Dissertation

Defense held on 20/12/2013 in Luxembourg

to obtain the degree of

Docteur de l'Université de Luxembourg

En Informatique

by

Benoît Bertholon
Born on June 12th, 1986 in Clermont Ferrand (France)

CertiCloud and JShadObf
Towards Integrity and Software Protection in Cloud Computing Platforms

Dissertation Defense Commitee

Abstract

A simple concept that has emerged out of the notion of heterogeneous distributed computing is that of Cloud Computing (CC) where customers do not own any part of the infrastructure; they simply use the available services and pay for what they use. This approach is often viewed as the next ICT revolution, similar to the birth of the Web or the e-commerce. Indeed, since its advent in the middle of the 2000's, the CC paradigm arouse enthusiasm and interest from the industry and the private sector, probably because it formalizes a concept that reduces computing cost at a time where computing power is key to reach competitiveness. Despite the initiative of several major vendors to propose CC services (Amazon, Google, Microsoft etc.), several security research questions remain open to transform the current euphoria into a wide acceptance. Moreover, these questions are not always tackled from the user's point of view. In this context, the purpose of this thesis is to investigate and design novel mechanisms to cover the following domains:

- **Integrity and confidentiality of Infrastructure-as-a-Service (IaaS) infrastructures**, to provide guarantees on programs and data running in a virtualised environment, either before, during or after a deployment on the CC platform.
- **Software protection on Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) architectures**, using code obfuscation techniques.

This dissertation details thus two main contributions. The first one is the development and implementation of CertiCloud, a CC framework which relies on the concepts developed in the Trusted Computing Group (TCG) together with hardware elements, *i.e.*, Trusted Platform Module (TPM) to offer a secured and reassuring environment within IaaS platforms. At the heart of CertiCloud reside two protocols: `TCRR` and `VerifyMyVM`. When the first one asserts the integrity of a remote resource and permits to exchange a private symmetric key, the second authorizes the user to detect trustfully and on demand any tampering attempt on its running VM. These protocols being key components in the proposed framework, their analysis against known cryptanalytic attacks has been deeply analysed and testified by their successful validation by AVISPA [1] and Scyther [66], two reference tools for the automatic verification of security protocols.
The second major contribution proposed in this manuscript is an obfuscation framework named JShadObf, designed to improve the protection of Javascript-based software running typically on SaaS and PaaS platforms. This framework combines obfuscation transformations, code complexity measurements and Multi-Objective Evolutionary Algorithms (MOEAs) to protect Javascript code, the most ubiquitous programming language at the heart of most modern web services deployed over those CC infrastructures such as Google Office Apps, Dropbox or Doodle.

.

To Kristine & Edouard-Gabriel

Acknowledgements

I would like to thank my advisers, Professor Pascal Bouvry and Doctor Sébastien Varrette for giving me the opportunity to do a PhD at the University of Luxembourg and for their help during these 4 years of studies. With their support I was able to investigate the issues I was interested in, while being well supervised and advised.

I was very happy to work in the team of Professor Bouvry and I thank all the members (former and current) of the team: Agata, Ana-Maria, Apivadee, Bernabe, Cesar, Frédéric, Grégoire, Guillaume, Hyacinthe, Jakub, Johnatan, Juanlu, Julien, Marcint, Mateusz, Patricia, Sébastien, Sune, Xavier, Yoann for their collaboration but more importantly for their friendship and all the good moments we spent together.

Thanks to Ton Van Deursen for his help with protocol verification, especially with Scyther, to Bernabe Doronsoro and Gregoire Danoy for the Evolutionary Algorithm insight and recommendations, and also to Frederic Pinel for his help with statistical analysis and the discussions on programming languages.

And finally, I would like to thank my thesis defence committee: Pascal Bouvry, Christian Collberg, Jean-Louis Roch, Yves Le Traon and Sébastien Varrette for their presence and their reviews of this dissertation.

I'm grateful to Kristine for being supportive in every moment, especially the difficult ones during the writing period and for the review of the manuscript, to Edouard-Gabriel for keeping me awake during nights ;), and to my parents for their support and love.

Supported by the Fonds National de la Recherche, Luxembourg (PHD-09-142)

Contents

Chapter 1
Introduction

Contents

This Chapter introduces the thesis context, develops the motivations for the work and presents the different contributions.

1.1 Context

At the beginning in the mid 1990s the term "cloud" was used to represent the computing space between the provider and the end user. The term Cloud Computing (CC) was mentioned in academia for the first time by Professor Ramnath K. Chellappa in 1997 [102] where he defined CC as a *computing paradigm where the boundaries of computing will be determined by economic rationale rather than technical limits alone.*

CC is therefore not only about computing but also about economics; in practice, the CC paradigm regroups many existing Information and Communications Technologies (ICTs) ranging from web-hosting to grid-computing and applies a pay-for-use model on these services. Users of CC platforms have access to many services and pay for what they actually use in a flexible way; this approach is often viewed as the next IT revolution, similar to the birth of the Web or the e-commerce.

Flexibility is the main keyword that characterizes the CC paradigm, it allows the user to modify on-demand many aspects of the service he receives. For instance, in the case of web-hosting it is possible to increase the memory size allocated to the user's web-site or in the case of an email service to increase the disk space for storing messages. All these modifications are transparent, increasing the quality of service and providing a better user experience of CC services.

Another important advance in computing is the virtualisation technology, allowing the creation of a new type of CC which is based on the provision of an infrastructure to the user. This technology allows the execution of multiple virtual environments, each having a dedicated Operating System (OS), on a single host. The user has full control over his virtual environment called a Virtual Machine (VM) on which he can install, remove, develop and manage any software. This environment allowed the CC provider to pool VMs on a single machine regardless of the interaction between users as they are evolving in logically isolated environments. In addition the VMs have virtualised disks allowing easy backup, copy and migration. A multitude of new services emerged from this new possibility going from natural disaster recovery to duplication of servers in combination with an elastic load balancers [36] to distribute the load across the servers in case of heavy usage (or even in case of Distributed Denial of Service (DDoS) attack).

However CC suffers from many security issues, as illustrated by the fact that 41% of respondents in the CDW 2011 tracking poll [59] are concerned about these issues, making security in CC the first reason for holding organizations back from adopting the Cloud. This thesis attempts to deliver a set of novel security mechanisms operating both at middleware and software level, providing users with a more secure experience of CC platforms.

1.2 Motivations

The security issues raised by the Cloud paradigm are not always tackled from the user's point of view. For instance, considering an Infrastructure-as-a-Service (IaaS) Cloud, it is currently impossible for the user to certify in a reliable and secure way that the environment he deployed (typically a Virtual Machine (VM)) has not been corrupted, whether by malicious acts or not. The following legitimate questions are not answered by current security protections dispensed by the CC companies:

- Can I be sure that my environment and its associated data remain confidential on such a shared platform?
- Is there a way to ensure that the cloud resources are not corrupted?
- Once deployed, is there a way to assert on demand and trustfully the integrity of my environment to detect undesired system tampering (typically by means of rootkit or malware)?

Yet having these features would enhance the confidence in the IaaS provider and therefore attract new customers.

The work done in the first part of the dissertation tries to answer these three questions within the CertiCloud framework. However in the proposed solution which stands at the IaaS level, the user still has to trust a trusted party that control the physical access to the machine (which can be CC provider). To offer a decent software protection at the other Cloud layers, namely PaaS and SaaS, this thesis investigates novel obfuscation techniques to grant security guarantees to the user at an application level. As of now, the vast majority of CC-based web-services (such as Google Office Apps, Dropbox or Doodle just to cite a few of them) are relying on the JavaScript programming language to interact with the user as all modern web browsers – either on desktops, game consoles, tablets or smart phones – include JavaScript interpreters making it the most ubiquitous programming language in history. The protection of the programs at the heart of these services becomes thus more and more crucial, especially for the companies making business on top of these services. We have therefore decided to focus on the protection Javascript-based software. In this context, we have designed an obfuscation framework named JShadObf which corresponds to the second main contribution of this thesis.

1.3 Contributions

This section details the contributions as well as the different publications produced during the thesis.

Integrity in IaaS Cloud Computing platform has been the first problem studied to address the lack of security of data and computation from the user's point of view. Indeed in current CC configurations and platforms, the user has to trust completely the Cloud Provider. This research led to the development of a framework called CertiCloud based on a cloud middleware.

CertiCloud relies on the concepts developed in the Trusted Computing Group (TCG) together with hardware elements, *i.e.*, Trusted Platform Module (TPM) to offer a secured and reassuring environment. These aspects are guaranteed by two protocols: TCRR and `VerifyMyVM`. When the first one asserts the integrity of a remote resource and permits to exchange a private symmetric key, the second authorizes a user to detect *trustfully* and *on demand* any tampering attempt on its running VM. These protocols being key components in the proposed framework, we take very seriously their analysis against known cryptanalytic attacks. This is testified by their successful validation by AVISPA and

Scyther, two reference tools for the automatic verification of security protocols. CertiCloud relying on the above protocols, provides the secure storage of users' environments and their safe deployment onto a virtualisation framework.

While the physical resources are checked by TCRR, the user can execute on demand the `VerifyMyVM` protocol to certify the integrity of its deployed environment. Experimental results operated on a first prototype of CertiCloud demonstrate the feasibility and the low overhead of the approach, together with its easy implementation on recent commodity machines. The CertiCloud concept and its protocols have been published in [48] and in [49].

This allowed to increase the user's security, but one still argues that the CC provider cannot be trusted. Then to hide the algorithm developed by the user another approach has been studied: Obfuscation. Even if proven impossible by [45], the aspect of time limited black box security intuitively stated that a security for a finite period of time might still be achievable (for example, it is possible to find a RSA private key from the public key but due to the size of the key used, it is impossible in a reasonable period of time considering actual computing power).

This leads ot JShadObf (Chapter 9), a JavaScript Obfuscator based on Multi-Objective Evolutionary Algorithm (MOEA) which takes into account different aspects such as execution time, size of code, number of predicates to make source code unintelligible. The Obfuscation of source code is a mechanism to modify a source code to make it harder to understand by humans even with the help of computing resources. More precisely, the objective is to conceal the purpose of a program or its logic without altering its functionality, thus preventing the tampering or the reverse engineering of the program. That's probably why a company such as Google heavily uses obfuscation for most of its web services (Gmail, Google Docs etc.) Measuring the obfuscation capacity within JShadObf is based on the combination of well known metrics, coming from Software Engineering, which are optimized simultaneously thanks to Multi-Objective Evolutionary Algorithms (MOEAs). The framework has been tested with a simple multiplication matrix program and a widely used library on the web: `JQuery.js`. The results regarding source to source obfuscation have been published in [51] and [50].

The `JavaScript` language can therefore be used to develop obfuscated server side applications running on a CC platform with CertiCloud to ensure the user that neither the physical node nor his VM have been compromised.

These two approaches have been presented in international conferences and published in journals with peered review. It is also worth to mention that in complement to the work presented in this manuscript and the associated publications, the candidate used to collaborate on several pedagogical contributions such as a book chapter on the security of distributed systems from a practical point of view [47]. Also due to space restriction, it was not possible to detail the work carried on during the master's training of the candidate [157] despite its direct influence on the choice for this thesis. This PhD thesis was carried out in the CSC, under AFR fellowship.

Publications

Whether directly *i.e.* linked to the work presented in this manuscript or through a more general context, this dissertation led to peer-reviewed publications in journals, conferences proceedings or books as follows:

- 1 book chapter [49];
- 1 journal article [47];
- 4 articles in international conferences with proceedings and reviews [50, 52, 48, 157];
- 4 articles in international workshops with reviews [26, 28, 25, 29];
- 2 articles in French national conferences with reviews [46, 31].

These publications are now detailed.

Peer-Reviewed Journal (1)

[49] B. Bertholon, S. Varrette and P. Bouvry. CertiCloud, une plate-forme Cloud IaaS sécurisée, Technique et Science Informatiques, Lavoisier, 2012 .

Book chapter (1)

[47] B. Bertholon, C. Crin, C. Coti, J.-C. Dubacq and S. Varrette. Distributed Systems (volume 1); Design and Algorithms. vol. 1 chapitre: "Practical Security in Distributed Systems", pages 243–306, Whiley and Son publishing, 2011.

International Conferences with proceedings and reviews (3)

[50] B. Bertholon and S. Varrette and P. Bouvry, JShadObf: A JavaScript Obfuscator based on Multi-objective Optimization Algorithms, Proc. of the IEEE Intl. Conf. on Network and System Security (NSS 2013), June 2013, Madrid, Spain, IEEE Computer Society.

[48] B. Bertholon, S. Varrette and P. Bouvry. CertiCloud: a Novel TPM-based Approach to Ensure Cloud IaaS Security. In: Proc. of the 4th IEEE Intl. Conf. on Cloud Computing (CLOUD 2011), IEEE Computer Society publishing, Washington DC, USA, July 4–9 2011.

[157] S. Varrette, B. Bertholon and P. Bouvry. A Signature Scheme for Distributed Executions based on Control flow Analysis.. LNCS In: Proc. of the 19th Intl. conference on Security and Intelligent Information Systems (SIIS 2011), Springer Verlag publishing, Warsaw, Poland, June 13–14 2011.

International Workshops with reviews (4)

[52] B. Bertholon, S. Varrette and S. Martinez, ShadObf: A C-source Obfuscator based on Multi-objective Optimization Algorithms, Proc. of the 16th Intl. Workshop on Nature Inspired Distributed Computing (NIDISC 2013), part of the 27th IEEE/ACM Intl. Parallel and Distributed Processing Symposium (IPDPS 2013), May 20–24 2013, Boston (Massachusetts), USA, IEEE Computer Society.

[28] B. Bertholon, S. Varrette and P. Bouvry. TPM-based Approaches to Improve Cloud Security. In: 2011 Grande Region Security and Reliability Day (SecDay 2011), Trier, Germany, Mar. 25 2011.

[29] B. Bertholon, S. Varrette and P. Bouvry, Using Evolutionary Algorithms to obfuscate code, Evolve 2011, May 25-27 2011, Luxembourg.

[26] S. Varrette, B. Bertholon and P. Bouvry. A Signature Scheme for Distributed Executions based on Macro-Dataflow Analysis. In: 2010 Grande Region Security and Reliability Day (SecDay 2010), Saarbrcken, Germany, March 18, 2010.

[25] S. Varrette, B. Bertholon and P. Bouvry. A Signature Scheme for Distributed Executions based on Macro-Dataflow Analysis. In: 2nd Intl. Workshop on Remote Entrusting (Re-Trust 2009), Riva del Garda, Italy, Sept 2009.

French National Conferences with reviews (2)

[46] B. Bertholon, S. Varrette and P. Bouvry. CertiCloud: une plate-forme Cloud IaaS scurise. In: Proc. des 20me rencontres francophones du paralllisme (RenPar'20), St Malo, France, May 10–13 2011.

[31] S. Martinez, S. Varrette and B. Bertholon. Optimisation d'obfuscation de code source au moyen d'algorithmes évolutionnaires multi-objectifs [Poster], In: Proc. des 20ème rencontres francophones du parallélisme (Compas'13), Grenoble, January 15–18 2013.

1.4 Dissertation Outline

The thesis is organised as follows: the first part (Part I) presents the basic information for reading the dissertation. The second part of the thesis (Part II) presents a way to increase security for **IaaS** platforms using hardware security equipments and security protocols which have been validated. The third part of the thesis (Part III) focuses on obfuscation techniques to increase security for **PaaS** and potentially **SaaS** platforms.
We now briefly review the outline of the successive chapters of this manuscript.

Part I

Chapter 2 presents Distributed Systems in general and Cloud Computing (CC) in particular with an overview of the different types of CC and of the existing IaaS middleware used to deploy Virtual Machines (VMs) into a Cloud environment.

Chapter 3 introduces the cryptographic primitives used in this thesis, specially in the CertiCloud framework. A brief presentation of the symmetric and asymmetric cryptosystems and how they can be used, combined with secured hash functions, to communicate in a secure way between parties.

Chapter 4 outlines hardware attacks on secured chips, and presents the TPM, an hardware based protection which has been used in CertiCloud.

Chapter 5 presents the existing software protections either at the OS level or at the Software level, impacting either the security in IaaS platform or in PaaS and SaaS platforms.

Part II – CertiCloud

Chapter 6 introduces briefly the different methods and tools used to verify the security protocols using automatic verification.

Chapter 7 presents one of the main contributions of the thesis which is the development and the implementation of CertiCloud, a TPM-based framework that increases the security of a user's VMs in **IaaS** platforms.

Part III – JShadObf

Chapter 8 contains an overview of the obfuscation techniques that are used in JShadObf. It provides as well some definitions and describes the metrics used to evaluate software complexity.

Chapter 9 presents the second main contribution of this thesis, namely the JShadObf framework, by detailing its implementation and explaining how the different transformations are combined with Evolutionary Algorithm (EA) to produce obfuscated `JavaScript` code. This allows to protect code running on **PaaS** of **SaaS** platforms.

At the end of the manuscript, a final part features the **Chapter 10** which summarizes the dissertation by presenting conclusions of the work performed together with outlines on future work and perspectives. The Figure 1.1 presents dependencies between the different chapters and therefore proposes a reading order for the thesis.

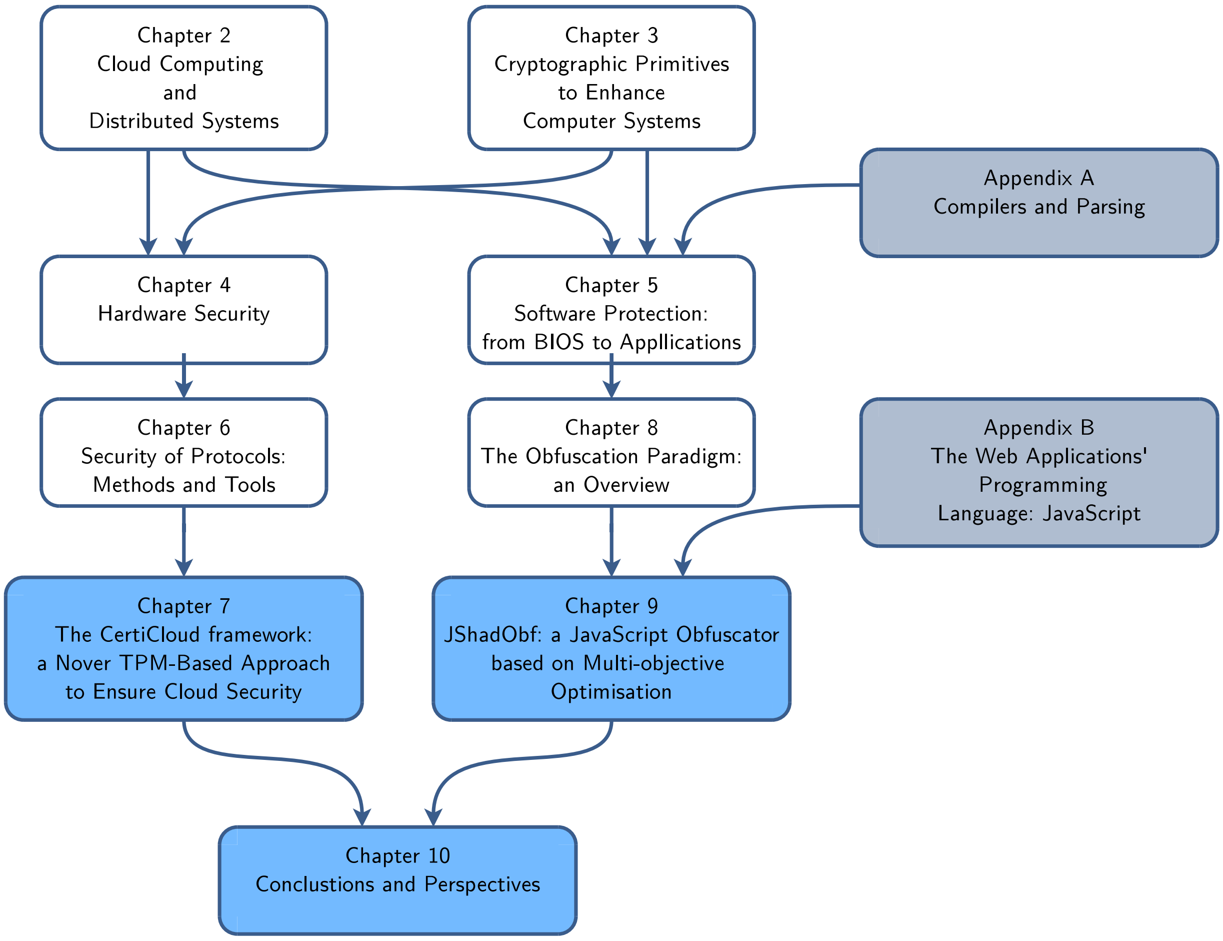


Figure 1.1: Graph of dependencies between the different chapters of the thesis.

Part I

Background

Chapter 2
Cloud Computing and Distributed Systems

Contents

The aim of the work presented in this thesis is to increase the security in Cloud Computing (CC) platform from the point of view of the user, it is therefore necessary to recall what is CC and where it comes from. This chapter is dedicated to the presentation of Distributed Systems in general and Cloud Computing in particular.

2.1 Distributed Systems

Prior to the emergence of the CC, the existence of distributed system already tried to tackle the problem of using multiple machines to perform computations. Computing platforms can be divided into two categories:

- Hard Coupled Systems: these systems communicate using interaction through shared memory, they have a centralized control and usually based on multiprocessor system. Examples of such systems are Massively Parallel Processors (MPPs) like BlueGene/Q [1] with more like 5 TFlop/s and 458752 cores.

- Loosely Coupled Systems: theses systems communicate via messages and have a more decentralized control, the global state of the computation is not known. They are as well referred as Distributed systems.

[1]http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/bluegene/

Definition 1 (Distributed Systems [147]) *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

2.1.1 Deployment types of Distributed Systems

There are two types of deployment of distributed systems:

- Client / server model where all the information is centralized on one or more servers, while the nodes (clients) connect to them to request information.
- Peer2peer model where every node is both a client and a server. The information is distributed within the peers ensuring a better load balancing.

In all cases a distributed system tries to aggregate the maximum number of nodes to increase its number of computations per second and therefore to decrease the time of the calculation.

2.1.2 Types of Distributed Systems

Distributed Systems can be classified into two types:

- Computation based: in this case a high number of machines with performant Central Process Units (CPUs) are required to increase the computational power of the system.
- Storage based: such system tries to increase the number of bytes it can store and later access.

The essential technical basic blocks necessary to build Cloud Computing (CC) platforms can be found in distributed system, specially for the IaaS type of Cloud that will be described later in Section 2.2.3.

2.2 The Cloud Computing Paradigm

Cloud Computing ([40], [156]) is a recent computing paradigm which is based on the on-demand access to computing services over the network and with its elastic-provisioning the user can modify his resources and request new ones at any time. This gives to the user an impression of an infinite number of resources, allowing small companies to increase their consumption to follow their needs. Combined with the pay as you go model of Cloud Computing, the service provider charges the user depending on the resources he requested. The CC paradigm is one of the fastest growing markets in ICT as shown in Figure 2.1 ([80]).

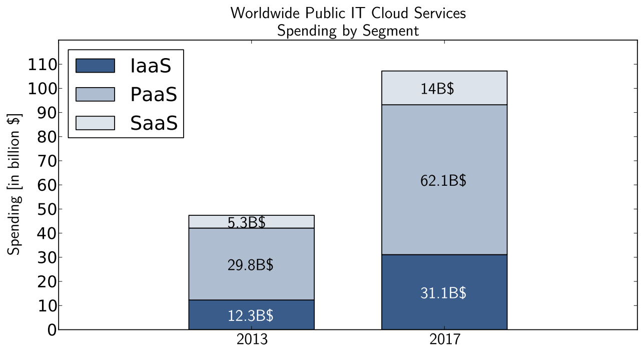


Figure 2.1: Cloud Computing: a fast growing market segment.

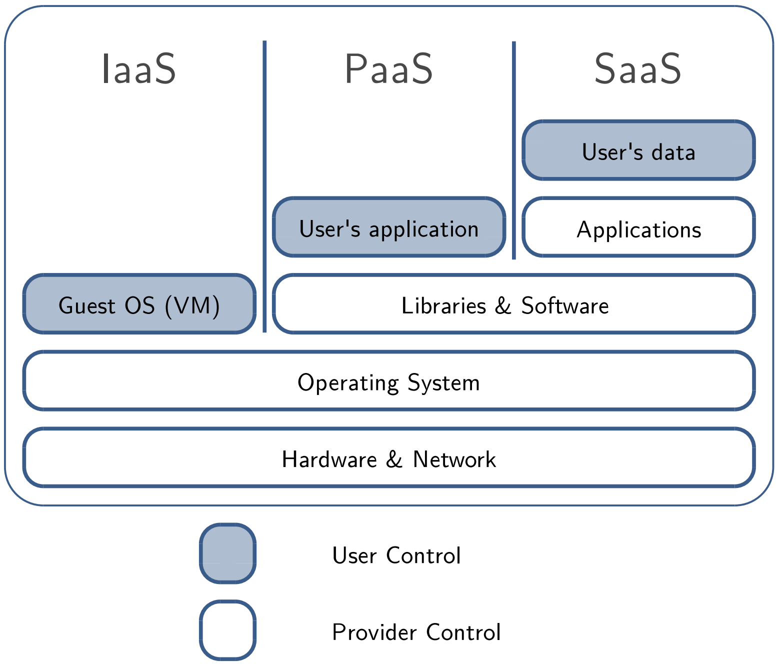


Figure 2.2: General classification of Cloud Computing platforms: a stack overview.

Cloud Computing services regroup different categories of services which range from access to virtual machines to web-hosting and redundant storage. A commonly used classification of the CC services is divided into three major categories [158], SaaS, PaaS and IaaS described in Figure 2.2. CC is however not limited to these three categories, one can cite for example Hardware as a Service (HaaS) [145] where the provider gives access to hardware such as graphic cards or special equipment.

The next sections offer a characterization of each class over three attributes: user, user's acquisition and payment method.

2.2.1 Software-as-a-Service (SaaS)

This category regroups the services where an end-user's software, which is running on the service provider's infrastructure, is accessed on-demand by the user. This category of cloud computing is the oldest, the term Software-as-a-Service (SaaS) is even anterior to the CC, it dates to the 1990s when the "Web services" emerged. The cloud provider manages and controls the application so that no intervention from the user is needed. In this case, the user does not need to own and update the software but rather pay for its use, for example through a web API.

Users:

- Employees accessing the organization software, *e.g.* webmail.
- Private individuals using online applications such as Google Apps [93], Google webmail Gmail [92], Facebook [165], the blog hosting service `wordpress` [155].
- Companies relying on SaaS to share and edit their documents *e.g.* using Google Docs [94].

User's acquisition:

Access of the online application.

Payment:

The payment method of this category depends on the service provided, it can be determined by the number of users, the network bandwidth, the quantity of storage used, it can be free but relying on the advertisements based on the user's data, etc.

2.2.2 Platform-as-a-Service (PaaS)

The PaaS category allows the user to have more control over applications but as well more obligations. The service provider chooses, installs and updates the programming languages and tools that the users will have access to. The users deploy their applications on the PaaS cloud using a remote control shell or a web interface. More customizable web-hosting allows the user to choose, to install and to maintain a website providing only scripting languages support and disk storage. This model is used by [97] or [105] which can run websites of multiple clients on a single server. The platform is responsible for allocating resources to the user's applications according to the user's demands. The user can for example ask for more storage, more Random Access Memory (RAM) or more network speed.

Users:

- Application developers, testers and deployers, who design, implement, test and deploy an application's software.
- Users of the developed application.

User's acquisition:

Access to the CC provider's tools and resources.

Payment:

Based on the number of users, storage, CPU time, network, software licences used.

2.2.3 Infrastructure-as-a-Service (IaaS)

Finally IaaS, the lowest category *i.e.* the closest to the hardware. IaaS authorizes the deployment and the execution of an environment fully controlled by the user – typically a Virtual Machine (VM) – on the Cloud resources. The user will have full control over its environment *i.e.* will be `root` on the machine, and will be able to install the software he needs. Instead of purchasing servers, software, data center resources, network equipment and the expertise to operate them, the user can buy these resources as an outsourced service delivered through the cloud. The main advantage of this category is its elasticity, as the users can scale their virtual infrastructures on demand. For example, a service provided by the user might have more and more clients, it can then be scaled up and request more CPU and memory. This is what Amazon EC2 proposes with the `Auto Scaling` of resources on its cloud [38].

Users:

System administrators, programmers.

User's acquisition:

Access to VMs, data storage and network bandwidth.

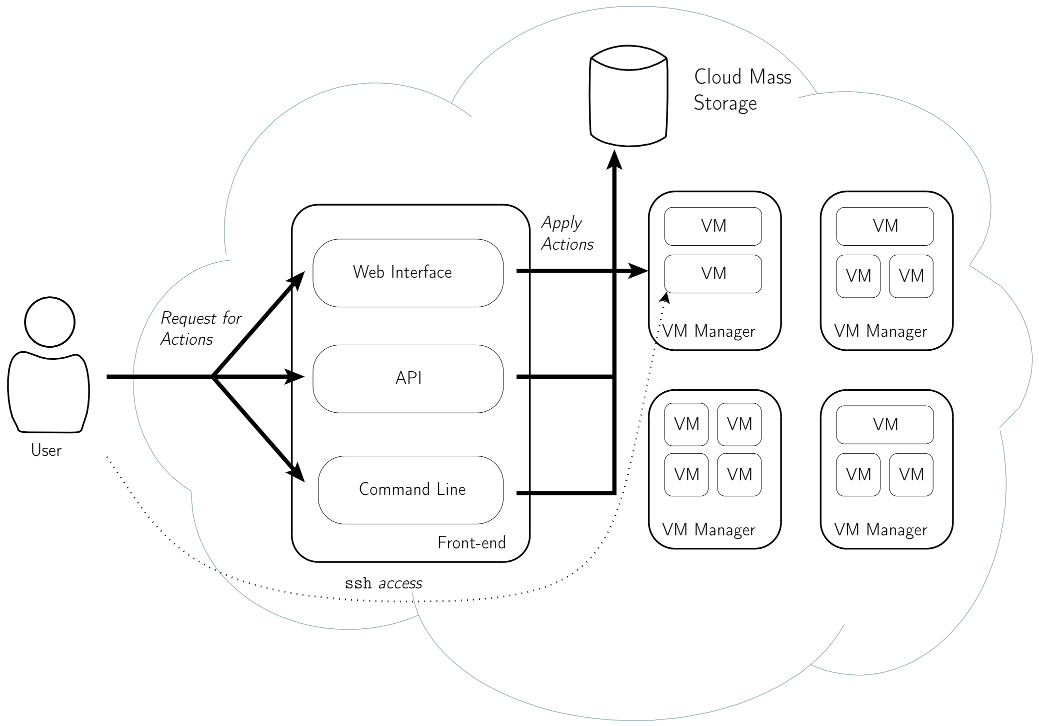


Figure 2.3: IaaS platform.

Payment:

Usually depends on the time spent by the user and the number and size of the VM requested.

IaaS platforms can be deployed on Public, Private of Hybrid Cloud. It is the type of Cloud which offers to the user the most control over the machine. It allows him to install the OS he desires and to have total control over the system. Since the release of Amazon EC2 in 2006, the number of open source CC middlewares increased with, for example, Eucalyptus [77], Nimbus [126], OpenNebula [10], and OpenStack [11]. These middlewares will be detailed in Section 2.3.

A general overview of an IaaS platform management is proposed in Figure 2.3. The user has access to a front end through a web interface, a command line or an Application Programming Interface (API), allowing him to perform actions involving VM images which are files containing a disk image, it is the representation of a raw disk or partition, therefore the size of these images can go from hundreds of megabytes to hundreds of gigabytes depending on the usage made by the user. The VM images are stored in the CC provider's mass storage, for example in the case of Amazon, the name of the service for the storing of Amazon Machine Image (AMI) is S3 [35].

The following actions are the basic ones available by default on any IaaS CC platform:

- Uploading VM images: the user prepares his virtual machine locally and sends it to the CC mass storage. The CC provider often supplies the user with existing images for widely used OS such as Windows, Ubuntu, CentOS or RedHat.

- Starting VM: the user can start a VM image by creating one or more instances that will use the image as reference *i.e.* it will copy the image as many times as the number of instances created.

- Terminate VM: the user can as well terminate a VM from the front end, allowing to bypass any shutdown problems.

- Saving VM: the user can save the VM image in order to commit all changes performed. This new image can be later used to create multiple instances of a customized environment.

The following actors present in the IaaS scenario can be found in every implementation of the CC platform under different names. The role of the front-end, the Virtual Machine Manager (VMM) and the storage are briefly explained here.

Front-end

The front-end is the access interface from which the user can connect to the IaaS. It is dealing with many important aspects of the service such as authentication, account management, billing, scheduling of the user instances, etc. If the user wants to perform any of the actions described earlier, he has to connect with the front-end endpoint. For big CC platform, this endpoint is of course composed of more than one server. The API used to contact the front end is very often following Amazon EC2 API. Indeed due to the size of Amazon and its early implantation in the IaaS world, the open-source CC middlewares often follow the same API (within the bounds of their possibilities). However other API such as Open Cloud Computing Interface (OCCI) [27] are in development within the open-source community and might in the future be part of CC middlewares.

Cloud storage

The Cloud mass storage or Virtual Machine Image (VMI) repository is used to store the default images as well as the customized environment created by the users. These customized environments might come from the modifications performed on a default VMI supplied by the CC provider and saved by the user or simply coming directly from a VMI created by the user and uploaded on the mass storage. For performance reasons the VMI has to be transferred and copied on the Virtual Machine Manager (VMM). Indeed any I/O access, *i.e.* writing or reading on the disk would require a network communication, this is very costly in terms of performance for the VMI containing the OS, as the OS is usually composed of many small files (configuration files, library ...). It would result in many network communications delaying the OS accesses and overloading the network. However extra persistent storage on the network is sometimes included in the platform.

Virtual Machine Manager node

The VMM nodes are responsible for the management of VMs. On these nodes, the hypervisor software responsible for the virtualisation of the hardware and the virtual network connections is installed. Many hypervisors exist such as Xen, KVM, VMWare ESX, Microsoft Hyper-V, OpenVZ, Virtualbox. They are able to simulate hardware (like CPU, Disks, Networks ...) and allow the creation of a virtualized environment. In this virtualized environment, an OS such as Linux, Unix or Windows can run, believing that it is running on a physical machine. But all the hardware it sees is only emulated by the hypervisor. In such case the user is completely free to operate his environment as he wishes because it does not have any effect on the host system. Another positive point for this architecture is the containment between the VM. As they think they are alone on the machine, they cannot (at least should not be able to [134]) interfere with other VMs simulated hardware.

2.2.4 Deployment models

After the different categories of CC, the types of deployment are as well important for the user. Indeed the Cloud Computing services that he will use can be deployed either on public cloud, on community cloud, on private cloud or on a hybrid version. These types of deployment are mainly for the IaaS category, but can be relevant as well for an SaaS service. Indeed the SaaS service provider does not have to own the resources, he can use an IaaS service to deploy his SaaS or PaaS services.

Public Cloud

The public cloud is provided by companies selling their infrastructure to allow clients to use it for outsourcing their computing hardware. This is the business model of companies like Amazon with EC2 [37] or Microsoft with Azure [122].

Community Cloud

The community cloud is an infrastructure dedicated exclusively to a specific community of consumers from organizations sharing the same goals, policy, security requirements, etc. It is owned by one or more of the organizations or by a third party or any combination involving these actors.

Private Cloud

A private cloud is managed and used exclusively by a single organization. This idea of resource sharing is not new (*e.g.* existence of computing grid) but the emergence of CC allows the use of virtualized environment and the tools provided by the middlewares such as migration, creation of snapshots or redundancy. The IT infrastructure of the organization is therefore enhanced with extra capabilities providing resilience and easier management. And even if this model does not use actual billing as it usually appears within an organization, it allows cost evaluation for reporting purposes.

Hybrid Solution

Common APIs allow hybrid solution for a user or a group of users within an organization to scale up their private cloud and request resources from a public cloud in case of high demand. The low price of private cloud is therefore combined with the feeling of infinity of resources provided by the public cloud.

2.3 Overview of the main Cloud Computing Middlewares

This section presents some of the well known middlewares available to deploy Virtual Machines (VMs) in an IaaS Cloud environment. We will see Eucalyptus [77], Nimbus [126], Open Stack [11] and Open Nebula [10] which are the four principal non-commercial CC platforms.

The commercial solutions such as Amazon EC2 [37] and Microsoft Azure [122] are not detailed due to the lack of information on their internal structure.

The Table 2.1 compares the most used open-source *middlewares* for Cloud Computing (CC) as well as vCloud. The open-source middleware has been tested to include in the table a parameter called *implementation feasibility* of CertiCloud, characterised by multiple criteria such as the accessibility of the source code, the languages used, the simplicity of the architecture, the robustness, the clarity of the log files. This metric is however subjective and has not been formalised using the criteria mentioned previously.

2.3.1 Eucalyptus

Eucalyptus is a CC middleware composed of five elements. They are all acting as standalone web services:

- **CLC**: The CLoud Controller is handling the different cluster controllers. It is acting as well as the front-end, therefore communicating with the users and administrators through a web-based

Middleware:	vCloud	Eucalyptus	OpenNebula	OpenStack	Nimbus
License	Proprietary	BSD License	Apache 2.0 License	Apache 2.0 License	Apache 2.0 License
Supported Hypervisor	VMWare/ESX	Xen, KVM, VMWare	Xen, KVM, VMWare	Xen, KVM, Linux Containers, VMWare/ESX, Hyper-V, QEMU, UML	Xen, KVM
Last Version	5.1.2	3.3	4.2	7 (Grizzly)	2.10.1
Programming Language	N.A.	Java / C	Ruby	Python	Java / Python
Host OS	VMX server	RHEL 5, ESX Debian, Fedora, CentOS 5, openSUSE-11	RHEL 5, Debian, Fedora, CentOS 5, openSUSE-11	Ubuntu, ESX Debian, RHEL, SUSE, Fedora	Ubuntu, Debian, RHEL, SUSE, Fedora
Guest OS	Windows (S2008,7), openSUSE, Debian, Solaris	Windows (S2008,7), openSUSE, Debian, Solaris	Windows (S2008,7), openSUSE, Debian, Solaris	Windows (S2008,7), openSUSE, Debian, Solaris	Windows (S2008,7), openSUSE, Debian, Solaris
Integration Possibility CertiCloud	- - -	+	+ +	-	+ + +

Table 2.1: Summary of differences between the main Cloud Computing (CC) *middlewares*.

interface or through command lines tools. It is as well responsible for high level scheduling of the work load and its API is using Amazon Standard.

- **Walrus**: The cloud storage controller is handling the transmission of the VM images (references and customized). It is as well responsible for storing persistent data for the user using *buckets* and objects. It is accessible from both inside the CC platform and outside.

- **CC**: The Cluster Controller is interfacing between the computing nodes and the cloud controller. It schedules the tasks to do within the cluster of computing nodes it is controlling.

- **SC**: The Storage Controller is providing the VM with storage volumes which are dedicated to a VM and cannot be shared between VMs. It is however possible to create snapshots of these volumes and share the snapshot within the availability zone. This concept is similar to Amazon Elastic Block Store (EBS).

- **NC**: The Node Controller is driving the hypervisor software handling the VM. It is as well responsible for retrieving VM images on the Walrus controller and maintaining the cache of these images.

The Eucalyptus virtualisation is using either Xen, Kernel-based Virtual Machine (KVM) or VMWare and has its API compatible with Amazon EC2 and Amazon S3.

At the security level, Eucalyptus groups and rules for the firewall similar to EC2 with a network isolation using Virtual Local Area Network (VLAN). As often the confidentiality and the integrity of communications is based on the WS-agreement protocol enforcing the use of timestamps to prevent replay attacks. Often similar measures are available in other middlewares but none of them is able to answer the questions raised by CertiCloud.

2.3.2 OpenNebula

OpenNebula is a simple CC middleware composed of two elements:

- Front-end: on which the OpenNebula services are installed and should have access to the Datastores as well as to every host. The installation includes the management services, the monitoring and accounting services the web interface and the API servers.

- Cluster nodes (Host): no OpenNebula installation is needed on the nodes, only an access from the front-end to them and rights well configured to be able to launch the hypervisor commands to run or terminate the VMs.

The installation on the front-end is very light ($\sim$ 10 MB), and no installation of the nodes is required. However the network has to be well configured to handle the VMs' communications.

2.3.3 OpenStack

OpenStack is the most recent open-source CC middleware of the selection, it has been developed originally by Rackspace [14] and NASA [8], then well known commercial companies such as AT&T, Nebula, RedHat, IBM, Ubuntu, Hp, Dell, Cisco, Citrix, Intel, VMware, joined the development.

OpenStack is divided into seven core components running as independent projects:

- Glance: proposes a catalogue of reference VM images for common OS installed as well as customized images. This service is used to reference the different available images (depending on the credentials of the user), but the storage itself can be done using a swift service, a normal file-system, Amazon S3 or a `http` server.

- Nova: it is installed on the computing nodes, this project drives the hypervisor installed on the system. The compatible hypervisors are KVM, LXC, QEMU, UML, VMware, Xen. This is the most complex and distributed component of OpenStack. It includes many programs and modules such as `nova-api` supporting different API, OpenStack Compute API, Amazon EC2 API and a special admin API used by administrator of the platform, `nova-compute` which is responsible for driving the hypervisors, `nova-schedule` responsible for scheduling the VMs on the node, `nova-network` for manipulating the virtual network and the bridges, `nova-novncproxy` for accessing the screen of the instances.

- Horizon: proposes a dashboard which is a web-based user interface to manage VMs. This service needs to have access to the same network as the user and to most of other services running on the platform.

- Keystone: provides the authentication for all other OpenStack services.

- Quantum: provides network connectivity between the applications managed by other OpenStack services. This service allows the users to create their own networks and then attach interface. The main interaction this service has is with the other nova services.

- Swift: proposes an object storage allowing user to store and retrieve files. It is distributed to prevent single point of failure, includes a proxy server `swift-proxy-server` which accepts requests using the OpenStack Object API, account servers for the rights, container servers for folders and object servers for actual files.

- Cinder: is the other service linked to storage, allows the users to store their volumes which is an additional persistent storage. The API allows the manipulation of volumes containing the VMI.

Due to the implication of many different well known companies, OpenStack is the most promising open-source project for Cloud Computing (CC). However due to its multiple projects, its installation and modification through the mean of plugins requires extensive knowledge of the middleware.

2.3.4 Nimbus

Nimbus is an open source middleware based on Globus [81] and composed of two elements:

- Service Node: managing the authentication of the user and its access to the cloud, plus the storage system called Cumulus.
- Virtual Machine Manager: which is the node controller driving the hypervisor.

Nimbus only relies either on Xen or KVM technologies. However it has been chosen for our work due to it simplicity regarding the number of actors and the usability. The concept is nevertheless extensible to other middlewares.

2.4 Promises and Open Issues in IaaS Cloud Computing

The promises of CC are presented in [44], it details what the users of CC services are expecting.

- Availability: the availability of resources is measured as percentage of resource uptime. This percentage usually appears within the Service Level Agreement (SLA) and is ranging generally from 99.5% to 100% of guaranteed uptime. However the computation method of the availability might be different depending on the service provider.
- Failure handling: this defines how the service provider should compensate the downtime, *i.e.* periods when the service is unavailable. The provider can for example offer some credit for further use of the CC platform.
- Data preservation: the data of the user has to be kept for a certain period of time even if the user has not fulfilled its payments.
- **Privacy of data**: one of the hardest promises to verify, the service provider has to guarantee not to sell, license or disclose any of the user's data and programs.

The limitations linked to the previously mentioned promises are the following:

- Scheduled outage: any outage which has been scheduled, announced and bounded in time is not counted as downtime.
- *Force majeure*: any outage due to an event outside the control of the provider is not counted as downtime.
- SLA changes: the user is responsible for checking any modifications in the SLA. These changes however have to be published with advance notice.
- **Security**: providers generally do not take responsibility for security breaches or for security in general. This means that in case of disclosure of the user's data due to a malicious activity the provider disclaims security responsibility. It is however hard to prove for a user that a disclosure was indeed the result of a malicious attack outside the service provider's control.
- API: due to the evolution of the middleware and the addition of new services, providers reserve the right to modify the API.

As an emerging technology, Cloud Computing (CC) has still numerous open issues. Often these issues were present in other domains but with the increase of popularity of the CC paradigm, they became more relevant. The next subsections will present some of these open issues, which are relevant for this work.

2.4.1 Computing performances

The cloud performances requirements are linked to the services provided by the user *e.g.* a web server which is generating web pages for human reader has less requirements than an High Performance Computing (HPC) computation. The computing performances characterised by the following constraints.

- Latency: the latency is the time for the service to answer the user. This time is at minimum the time for the request to travel through the network infrastructure. The closer (physically) the user is to the cloud providers, the lower the communication delays are. The increase of the network infrastructures and VLAN between the user and its VMs or the VMs running the service leads to higher latency and therefore a decrease in the Quality of Service (QoS).
- Scalability: program running on desktop or even cluster has to be adapted to the cloud to benefit from the elasticity of the infrastructure.
- Data storage: computing performances are dependent on the data location: the closer to the computing unit the smallest the retrieving time is. This means that the location of data influences the computing unit.

2.4.2 Reliability

Reliability is a key issue in CC. When the users depend on the availability of their computing resources to run their business and the service provider is committed to his SLA, minimising the downtime is crucial.

The network connectivity is essential for a CC service to be run. Obviously interruption of network services leads to interruption of the whole system itself. A lot of applications need a continuous communication and are sensitive to interruptions *e.g.* an HPC computation might crash in case of network errors. CC services are created and administrated by man in an unpredictable world, therefore utility outage is inevitable whether from malicious attack, administration errors, hardware failure or natural causes. Depending on the reliability level needed, solutions such as redundancy of hardware might be necessary. In case of critical infrastructure, disaster recovery policies *e.g.* mirroring the whole service on a different physical location, could be considered.

2.4.3 Interoperability

Efforts have been done by the open source community to develop a common standard or to be compliant with the Amazon API, but the migration of an application from one CC provider to another is still a non-trivial problem. Interoperability between the different CC platforms implies standardisation at many levels of the different basic blocks composing the platform. For example, transferring a VMI running on Microsoft Azure to a Nimbus based CC platform requires the conversion of the VMI to be readable by the hypervisor supported by Nimbus and network reconfiguration might be needed. For SaaS services it is even more application dependent *e.g.* changing from GMAIL web-mail service to YAHOO and transferring the content of the mailbox is impossible.

2.4.4 Information security

Information security is one of the key issues of the Cloud Computing (CC) environment leading to the fact that many businesses are reluctant to externalise their IT services to the Cloud due to security. The polls in Figure 2.4 ([59]) present the three main reasons for companies not to migrate to the CC.

In certain cases they have legal obligation that obliges them to keep the data or programs within their control. All categories of CC are affected by security, specifically the IaaS category, where the logical compartmentalisation of OS creates a false sense of security. Indeed compartmentalisation

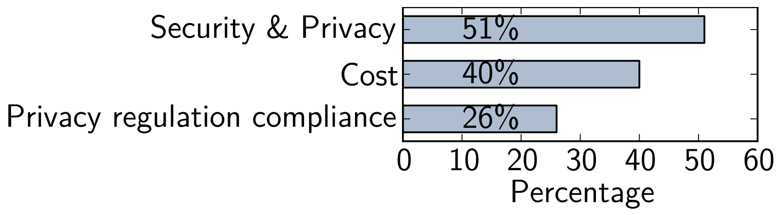


Figure 2.4: Top three reasons not to migrate to the cloud.

increase security but is not unbreakable and breaches have been found multiple times like in [134] and [128].

Data protection

Data in a public cloud has a different security exposure compared to an on-site environment. First of all a cloud infrastructure containing data from multiple sources is more attractive for a malicious attacker than a single site. Their security level or procedures are as well unknown to the user of the platform and certain requirements from the user have to be specifically handled by the service provider, *e.g.* the encrypted storage of the email of an email hosting company might not be feasible according to the service provider's architecture.

System Integrity

Cloud Computing platforms have to ensure the integrity of their systems to the clients, from the host OS to the applications depending on the kind of services they are providing. The partition of rights either in the file system itself or within the application using, for example, distinct groups has to be well defined and applied correctly throughout the system.

In the case of SaaS, the verification of the system integrity by the user is hardly feasible, leading to a blind trust to the cloud provider to ensure system integrity.

This dissertation will focus on the **Data protection** and the **Integrity** of the user's VM.

Chapter 3
Cryptographic Primitives to Enhance Computer Systems

Contents

One of the critical issues remaining in Cloud Computing (CC) platforms is the security of the resources composing the Cloud Computing systems. This topic is strongly linked to computer system security, indeed Cloud Computing security is based on notions coming from the computer security community. This Chapter is describing some of these concepts that will be used in the thesis. First we will define what computer security means and then we will describe some of the means used to secure computer systems.

3.1 Introduction

A secure computer system is the key to usability. Attacks to steal private information or to affect the correct behaviour of the system are common, therefore security measures need to be taken to secure data and infrastructure.

The terms defined bellow will be used in this dissertation.

Definition 2 (Attack) *An attack is an intentional action attempting to cause failure in a system or to gain information which the security policies do not allow you to have.*

Definition 3 (Vulnerability) *A vulnerability is a point in a system where an attack can be performed.*

The CIA model presented in Figure 3.1 for ICT security involves the following concepts:

- Confidentiality: private information should be protected to prevent its disclosure to any malicious attacker.

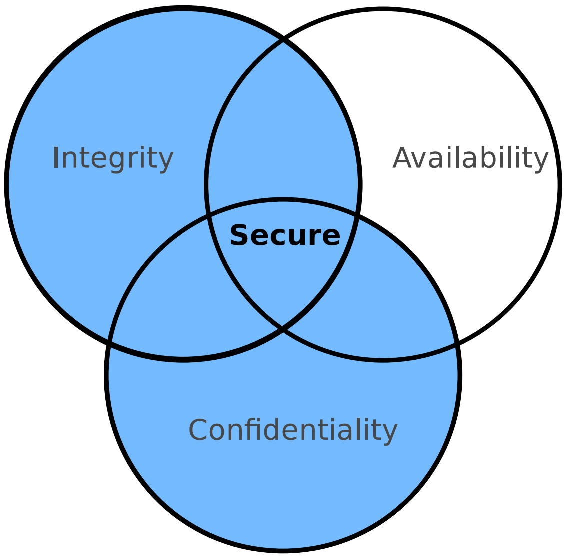


Figure 3.1: CIA security model (in blue domains holding contributions from this thesis).

- Integrity: a system should not be altered by a malicious user without being detected. In the case of data transmission, if a malicious attacker tampers the data during a communication then it should lead to a detection from the legitimate parties.

- Availability: the service should be available for legitimate users. A highly available system should always be active, even in case of attack, hardware failure, software error, or power outage.

Another concept which is often added to the CIA model is the non-repudiation, which is the impossibility for an actor to repudiate the fact that a message or communication has been performed by himself.

3.2 Encryption

The war between cryptographer and cryptanalysts exists since there is a need for secure communications. In the old times communication channels between parties were the messengers. Techniques to hide or crypt the messages appears to prevent a compromised messenger, *i.e.* caught by the enemy or with malicious intentions, from revealing confidential information. These techniques are called cryptographic techniques where cryptography literally means “secret writing”. The history of cryptography [100] can be divided into categories depending on the tools they used:

- Pre 1920: Paper-and-pencil with substitution, transposition, Caesar cipher, Vigenere cipher, Vernam cipher.

- 1920 - 1970: Machine ciphers with Enigma, Purple, Hagelin.

- 1970 - now: Computer Based with the algorithm: RSA, DSA, EC, DES, 3DES, AES .

- Future: Quantum cryptography?

Machines and after computers increased the computing power and helped alternatively the cryptanalysts and the cryptographers, leading to stronger cryptographic methods and ciphers to replace old and broken algorithms.

When dealing with cryptography theory the following definitions are used to depict the context. The following conventions and definitions are often used when describing a scenario: Alice and Bob are legitimate users and Eve is a malicious one as in Figure 3.2.

Definition 4 (Plain text) *the term "plain text" represents a non-encrypted text which can be read by anyone. This text is usually represented by the letter M.*

Definition 5 (Cipher text) *a "cipher text" is an encrypted version of a plain text. This text is usually represented by the letter C.*

Definition 6 (Encryption) *to encrypt a text means to make it unreadable. This encryption process is usually done by a function E and uses a key K_e.*

Definition 7 (Decryption) *To decrypt a text means to make it readable. This decryption process is usually done by an function D and use a key K_d*

Definition 8 (Insecure Channel) *an insecure channel is a mean of communication between two parties which is unsecured* i.e. *that a malicious party can read or alter.*

An example of an insecure channel is typically the Internet as the routers between Alice and Bob are not under their control.

Definition 9 (Cryptosystem) *a cryptosystem is set of algorithms needed to implement an encryption decryption system.*

A cryptosystem is often composed of three algorithms: one for encryption, one for decryption and one for key generation.

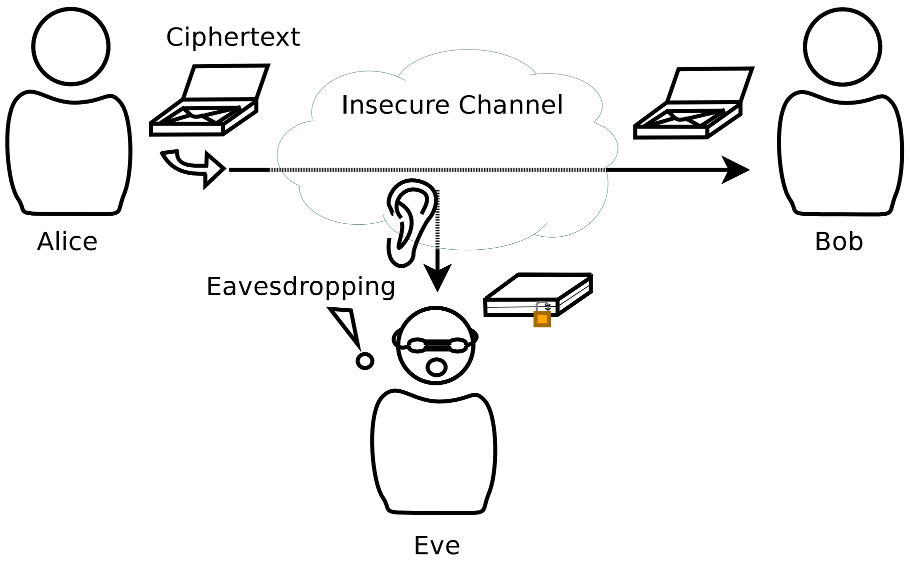


Figure 3.2: Principle of encryption.

One important principle in cryptography is how the security is measured, the following principle summarises good practise in conception of cryptographic systems:

Definition 10 (Kerckhoffs' principle [103]) *The security of a Cryptosystem should not be contained in the algorithm with which the cipher text is generated but rather in the secret key used to encrypt the message.*

This principle basically means that even if adversaries do whatever is possible (stealing machines, reverse engineering, bribing, kidnapping, etc.) to obtain useful information, they should not be able to use any encrypted data unless they have the secret key. The basic assumption is that Eve has access to everything, *e.g.* algorithms, [Pseudo]-Random Number Generators (PRNGs), hardware equipment, etc, except the key K.

To protect messages from disclosure, two types of encryption have been developed, historically the first one is the symmetric encryption (with methods as old as the Caesar shifting cipher), then the asymmetric encryption which is more recent (in the late 1970s).

3.2.1 Symmetric Encryption

One type of encryption used to protect the data is symmetric encryption. Symmetric encryption means that there is only one key to cipher and decipher the data which means: $E_K(M) = C$ and $D_K(C) = M$ are using the same key K to perform both operations. Therefore the key K is the secret that an adversary will want to have if he wants to understand the communication.

Caesar, Vigenère and Vernam ciphers

These three ciphers are categorised in the pre 1920s section. They use shifting which means that a letter is shifted to another letter *e.g.* the Caesar cipher is using a shifting of 3 which means that the 'a' letter is replaced by 'd', the 'b' by 'e', the 'c' by 'f' ... The decryption is quite simple and needs to shift the same number of letters but in the other direction.

The Vigenère cipher has the same principle but the number of shifting is determined by a password and the position in the text, in the following encryption $C[i]$ refers to the i^{th} letter of the cipher text, $M[i]$ refers to the i^{th} letter of the plain text and $K[i]$ refers to the i^{th} letter of the key (the addition has to be replace by subtraction in the case of decryption):

$$C[i] = M[i] + K[i \ mod \ lenght(K)]$$

The Vernan cipher is using the same principle as the Vigenère cipher with a key as long as the message itself. This cipher is unbreakable but very difficult to use in practise do to the key distribution problem.

The shifting in these ciphers uses a particular case of the substitution methods used in today's algorithm.

DES

The Data Encryption Standard (DES) [65] was invented by IBM and reviewed by the National Security Agency (NSA). It is a block cipher of 64 bits which means that it encodes data by dividing the input into blocks and it uses keys of size 56 bits. It is composed of permutations and substitutions operations. However due to the size of keys in the DES algorithm, brute force attacks are possible since the early 2000s, making it unsuitable for acceptable confidentiality. It was used as the standard for encrypting communication from 1976 to 2001 when it has been replace by AES.

Advance Encryption Standard

In 1997 the need for a more secure and fast symmetric encryption algorithm lead to an international competition for a new symmetric encryption which brought to light a new algorithm: the Advance Encryption Standard (AES) [67]. This algorithm is as well composed of multiple rounds of permutations and substitutions operations but the size of the keys varies from 128 bits to 256 bits.

3.2.2 Asymmetric Encryption

The asymmetric encryption is quite new compared to symmetric encryption. The first publicly available encryption scheme using asymmetric key is RSA developed in 1977 (found earlier by Clifford Cocks, a British mathematician and cryptographer at the Government Communications Headquarters (GCHQ)), but Diffie and Hellman in 1976 used a key exchange based on asymmetric cryptography. Asymmetric encryption means that the key to encrypt the cipher text is different from the key to decipher it. It results that $E_{K_e}(M) = C$ and $D_{K_d}(C) = M$ are using different keys K_e and K_d the encryption and decryption operations. This new cryptographic paradigm allowed secure communications between two entities without sharing a common secret, leading to easier key distribution.

Asymmetric encryption schemes are based on mathematical problems which are hard to solve. The computations needed to encrypt and decrypt are usually more time consuming than for symmetric cryptography, this is why they are used to encrypt small messages or even keys for symmetric cryptographic algorithms.

Diffie-Hellman Key Exchange

The Diffie-Hellman [73] key exchange is based on the difficulty of the discrete logarithms problem. The typical scenario for an exchange is the following:

- Alice or Bob selects a large prime number p and a primitive root α. p and α are public.
- Alice and Bob choose their private keys, respectively x and y.
- Alice and Bob exchange their public keys which are $\alpha^x \ mod\ p$ and $\alpha^y \ mod\ p$.
- Alice and Bob can compute the key $K = (\alpha^x)^y \ mod\ p = (\alpha^y)^x \ mod\ p$.

In this scenario it is very hard for an attacker to retrieve x and y from $\alpha^x \ mod\ p$. Retrieving this information is called the discrete logarithms problem.

Based on the same mathematical problem, the ElGamal [76] encryption scheme allows not only to exchange a common secret such as session key but also to send encrypted messages without needing an bi-directional communication between the two entities.

Rivest Shamir Adleman (RSA)

The RSA cryptosystem [138] is based on the factorisation problem. The following steps briefly describe the algorithm:

- Bob chooses two secret primes p and q and computes $n = pq$
- Bob chooses e with $gcd(e, (p-1)(q-1)) = 1$
- Bob computes d with $de = 1 mod((p-1)(q-1))$
- Bob's public key becomes n and e and his private key is $p\ q\ d$
- Alice encrypts m using the function $c = E_{n,e}(m) = m^e mod(n)$
- Finally Bob can decrypt c using the function $m = D_{p,q,d}(c) = c^d mod(n)$

Elliptic Curve (EC)

EC cryptosystems [162] are more recent and are based on the Elliptic Curve Discrete Logarithm Problem. It has the advantages of having smaller key sizes and keys faster to generate than RSA decreasing the computing power needed and making EC more convenient.

The asymmetric cryptography allows Alice to send a message to Bob without having to share a common secret with him. Alice just needs the public key of Bob to send him a message that only he can decipher. This allowed more possibilities than the symmetric cryptography, where the common secret has to be shared over a secure channel or prior to the communication. However, there is still a key distribution problem which is how Alice can be sure that the key she has is indeed Bob's public key.

3.3 Hash Functions

Hash functions are very useful in computer science and telecommunication, they are used to verify that a communication or a message has not been modified.

3.3.1 Checksums

Check-sums are used in error detection in telecommunication to ensure that the message sent has not been modified by a bad communication channel *e.g.* radio communication interferences. A well known check sums function is the Cyclic Redundancy Check (CRC) which has the advantage of being fast and easily implemented in hardware.

3.3.2 Cryptographic Hash

Check-sums function such as CRC are not suitable for cryptographic purposes as collisions are easy to find. The properties of a cryptographic hash function h – also called "message digest" as they are a "summary" of the message – are the following:

- Given the message m, the function $h(m)$ is fast to compute.
- It is a one way function, meaning that given d which is the result of the function $h(m) = d$ it is hard to find a m' such as $h(m') = d$.
- It is collision free, meaning that it is computationally infeasible to find two m_1 and m_2 such as $h(m_1) = h(m_2)$.

The Table 3.1 is referencing some exiting cryptographic hash functions as well as their complexity and if they are resistant or not.

Function	Size	Brute force attack	Collision resistance	Attack complexity
MD5 [137]	128 bits	$\mathcal{O}(2^{64})$	broken [142]	$\mathcal{O}(2^{30})$
SHA-1 [75]	160 bits	$\mathcal{O}(2^{80})$	broken [161]	$\mathcal{O}(2^{63})$
SHA-256 [24]	256 bits	$\mathcal{O}(2^{128})$	▨	▨
Whirlpool [136]	512 bits	$\mathcal{O}(2^{256})$	▨	▨
SHA-3 [53]	224, 256, 384 and 512 bits	$\mathcal{O}(2^{112})$ - $\mathcal{O}(2^{256})$	▨	▨

Table 3.1: Existing Hash functions.

Secure Hash Algorithm (SHA)

Developed by the NSA, the SHA-1 used to be the hash algorithm recommended by the National Institute of Standards and Technology (NIST) until 2012. It is composed of multiple rounds of bit shifting, word rotation, and bit operations such as `or`, `xor` and `and`. The NIST now recommends SHA-2 functions which have a similar algorithm than SHA-1 but no attack have been found on the full SHA-2. How ever attacks and weaknesses have been discovered on parts of SHA-2, *i.e.*, the reduction of the number of rounds in the algorithm allows an attacker to find collisions (two inputs resulting in the same hash value).

Therefore the NIST launched a competition [127] for a new SHA-3 which finished in 2013 making the new next standard based on Keccak [86] familly functions.

3.3.3 HMAC

HMAC is a special type of hash function used to for authentication of the message using a common secret. Indeed due to its construction and the one-wayness of hash functions, it is impossible to construct a correct HMAC value of a message M without knowing the secret K. It therefore can be used to authenticate a remote user with a challenge M to verify that the user is in possession of the common secret. In this case no encryption is needed to authenticate the remote user. The HMAC function is based on a cryptographic hash function h and is usually computed as follow:

$$HMAC_K(M) = h((K \oplus ipad)||h((K \oplus opad)||M))$$

With $ipad$ and $opad$, two constants formed using the values $0x36$ and $0x5c$ times the size of a hash block. Knowing the message and the common secret Alice will be able to verify the HMAC sent by Bob by computing the function. Alice will be sure that the message has been hashed by Bob.

3.4 Public-Key Infrastructure and Digital Signatures

The asymmetric cryptography have public and private keys allowing asymmetric encryption and digital signature, but the public key distribution problem arises.

3.4.1 Digital Signature

The principle for Digital Signature is as well based on public and private keys. Instead of encrypting with public keys and decrypting with private keys, the parties sign messages with private keys and verify signatures using the public key. Most of the asymmetric cryptosystems can be modified such as they can be used to perform signing and verifying operation. For instance, RSA cryptosystem has the following property: encryption and decryption functions are commutative, *i.e.*:

$$D_{K_{priv}}(E_{K_{pub}}(M)) = E_{K_{pub}}(D_{K_{priv}}(M))$$

It is therefore possible to use the $D_{K_{priv}}$ as a signing function $S_{K_{priv}}$, and the $E_{K_{pub}}$ as a verification function $V_{K_{pub}}$. This property is inherent to the mathematical problem it results from, *e.g.* for the Diffie-Hellman encryption $(\alpha^x)^y = (\alpha^y)^x$. It allows to "decrypt" before the encryption. When the decryption process happens on the message M rather than on the cipher text C it is called signature. If a message M is signed by Alice using her private key it means that anyone having Alice's public key can verify that the message was indeed signed by Alice.

In practise it is not the message itself which is signed but rather its "summary" *i.e.* the hash of the message. There are two reason for signing only the hash of the message: the first reason is that asymmetric cryptosystems are slow, therefore it is faster to sign only a hash of the message and the second reason is that if the message is big, it has to be divided into blocks to be signed. As each one of these blocks is signed, a malicious user can rearrange or remove some of them in order to change the meaning of the message *e.g.* if the block size is the size of an unicode character, the attacker can generate a completely different document but considered as signed.

3.4.2 Key Distribution

As seen previously with asymmetric cryptography there is no need to share a common secret prior to the communication between the two parties. However to send a message to Bob, Alice needs to know his public key. The key distribution over an insecure channel is subject to Man In The Middle (MITM) attacks, which means that a man in the middle (Eve) can impersonate Bob in the eyes of Alice, and Alice in the eyes of Bob, then relaying all the communications transparently. To solve this issue there are two main ways which are the use of a Certificate Authority (CA) or a web of trust.

Certificate Authority (CA)

A CA is a trusted third party that signs the public keys of different actors. The public key of the CA is conventionally possessed by all the actors.

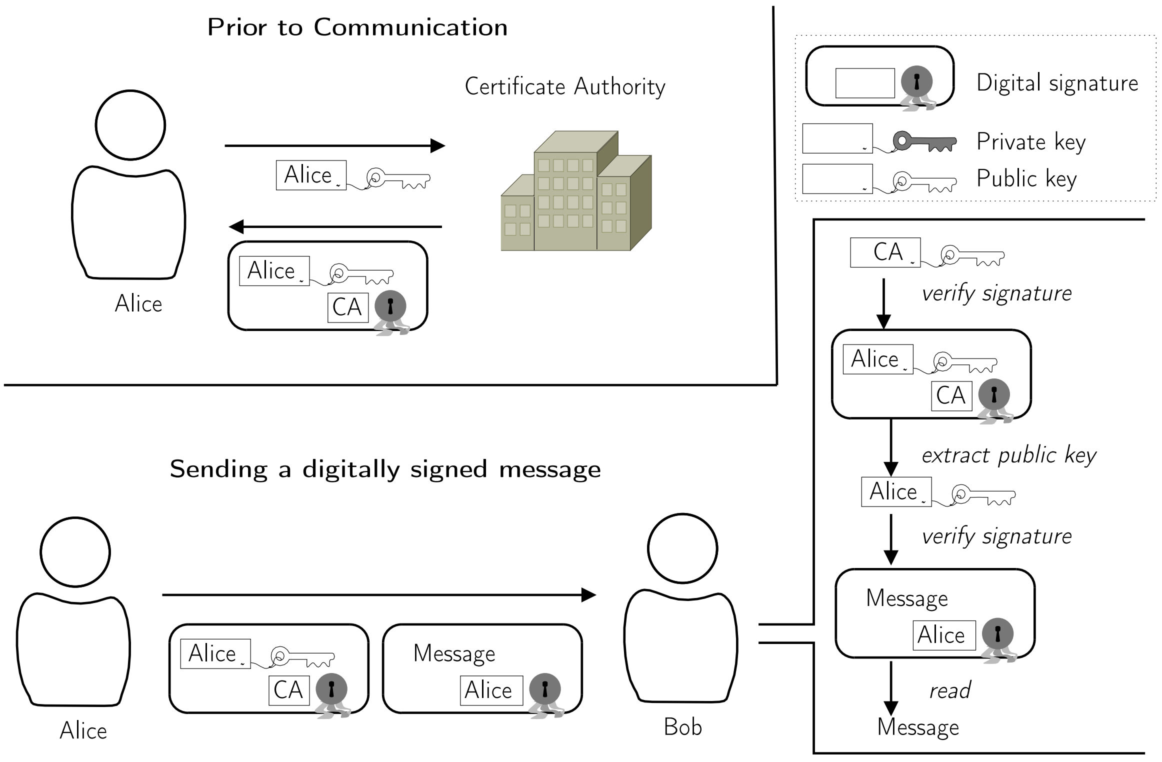


Figure 3.3: Scenario of Alice sending a signed message to Bob with a key signed by the CA.

The configuration seen in Figure 3.3 is the typical scenario happening on secured internet websites *e.g.* internet banking or webmail, where the websites have their public key signed by a CA. This signature is stored in a certificate which can be distributed to the users of their services. The users have the public keys of the main CAs within their web browsers which verify the certificate of the service providers.

This is a centralised approach which requires the trust of all the actors in the CA.

Web-of-Trust

The web-of-trust is the decentralised approach used in Pretty Good Privacy (PGP) [82] to solve the key distribution problem; a user does not trust a CA to sign the different keys but rather his network as well as the networks of his network, it is therefore based on the notion of "transitive trust". For example, if Alice trusts Mark and signed his key, and Mark trusts John and signed his key, and John trusts Bob and signed his key, then Bob can send a signed message to Alice and she will be able to verify his signature because she will know that the private key used to sign the message is indeed Bob's key by deriving trust through Mark and John.

3.5 Open issues in Computer Security

Computer security mostly relies on cryptography. But new attacks are discovered every day, computers have more and more computing power allowing stronger brute-force attacks.

However errors in the implementation of the algorithms make attacks possible without calling into question the difficulty of the mathematical problem.

Even if a cryptosystem is considered as unbreakable within reasonable time it does not make a computer system secure. Indeed the cryptographic algorithms are only one basic block, a tool, it does not make a system secure. To be secure, a system has to use hardware and software protection against the attacks.

Chapter 4
Hardware Security: Overview and Open Challenges

Contents

This Chapter presents a classification of hardware chip's attackers and a description of the Trusted Platform Module (TPM) which has been used in CertiCloud (see Chapter 7). This cheap hardware chip is already embedded in many motherboards, trying to provide basic cryptographic capabilities and platform security assessment.

4.1 Taxonomy of Hardware Attacks

This section will present the different hardware attacks by classifying the types of attackers, the types of attacks and the different methods for attacking hardware components.

4.1.1 Class of Attackers

The following classification of attackers has been given by [33]:

- Class I adversaries are considered as clever outsiders. They have neither knowledge about the system nor very sophisticated equipment. They usually try to take advantage of existing vulnerabilities.
- Class II adversaries are Class I adversaries with detailed understanding of parts of the system. They have access to highly sophisticated tools.
- Class III adversaries are funded organisations composed of Class II adversaries. They can design sophisticated tools specifically to attack the system.

4.1.2 Type of Attacks

A common classification of the types of attacks is based on the degree of invasiveness as in [133] or [144].

Non-invasive

The non-invasive attacks do not violate the physical integrity of the hardware component which means that the attack is done only by analysing the response of the device. A typical non-invasive attack is the side channel attack. A side channel attack is the analysis of a side channel to deduce what happens in the device.

For example, the side channel can be the power consumption of the device and its analysis can reveal what kind of operation the device is performing [106]. Time can be used as a side channel, indeed the analysis of the response time of a device to different requests might result in a gain of information [107]. A mid range oscilloscope is enough for these two attacks to measure with precision the variation in voltage and time response. Another side channel is the electromagnetic emissions produced by the device itself while running [135]. To analyse this channel both electromagnetic sensors and electromagnetic isolation are required.

The side channels are interesting to analyse during a normal scenario of the utilisation of the device but they often leak more information when used in unexpected conditions. These unexpected conditions are badly formed requests, incoherent data injection, a clock given to the device which is not in the range of the frequencies given by the device's specification, a lower voltage leading to errors, etc.

Other non invasive attacks are the attacks targeting software and protocol errors or using debug interfaces in an unintended way.

Semi-invasive

A semi invasive attack [144] is an attack which requires the exposition of the die of the device *i.e.* the plastic cover or part of it has to be removed. The exposition of the chip allows the attackers to inject more easily errors using laser pointers, photographic flash guns or UV light. These error injections use light, it is therefore possible to target specific zone of the chip *e.g.* cache memory or Arithmetic logic unit (ALU). With the exposition of the die the non-contact probing it is as well more reliable.

Invasive

Invasive attacks remove both the device package and the passivation layer *i.e.* the die of the chip is directly accessible. It is therefore possible for the attacker to perform modifications to the circuit by adding or removing parts. These kind of attacks is destructive, the process of removing the passivation layer alters the die integrity, therefore it requires high specialisation of equipment as well as skills and most attackers do not have such laboratory equipment to perform.

The available options to an attacker can be classified into three methods:

- Probing: mechanical probing implies physical connections using a probe to monitor the circuit. It requires the removal of the passivation layer used to protect the die from oxidation damages.

- Circuit editing: the modification of the circuit implies the creation and removal of some of the circuit connections. This often implies deterioration leading to possible destruction of the circuit.

- Reverse engineering: complete circuit netlist, *i.e.* connections between element of an electronic design, is extractable but destructive and requires many chips to extract the complete transistor or gate-level netlist information [154].

From a general point of view, semi-invasive and invasive attacks are only available to Class III attackers.

Tool based classification

Another classification of attacks' types is proposed in [151]. It is not based like the previous one on how the attack is performed and how invasive it is but rather depends on the tools necessary to achieve an attack. It defines three types:

- Hack attack: the attacker is only able to execute software attacks.
- Shack attacks: the attacker has access to equipment from a store such as Radio Shack. He has physical access to the device but lacks equipment or expertise to attack the device from within.
- Lab attacks: the attacker has access to high precision lab equipment such as microscopic logic probes to influence the device.

Summary of classifications

The attacks' classifications are summarised in Figure 4.1 which regroups the different types of attackers, their equipments and the types of attacks they are able to perform. Obviously the strongest type of attackers is the Class III, they can perform all the types of attacks defined previously.

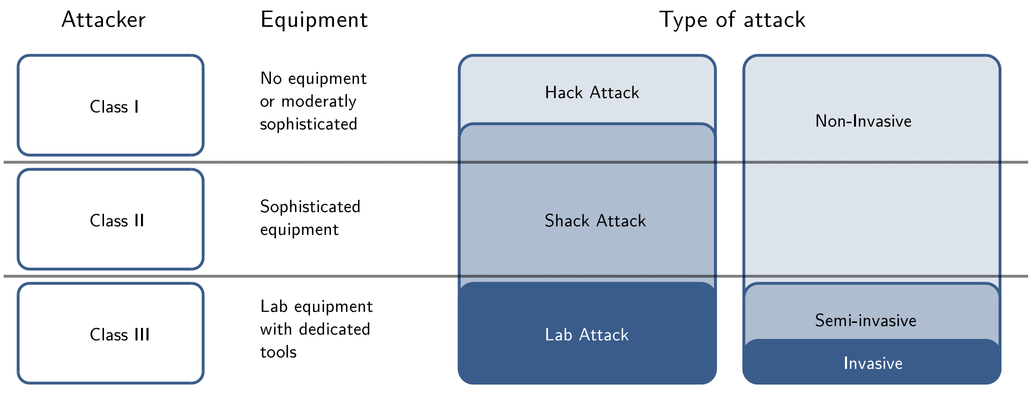


Figure 4.1: Summary of the attackers and types of attacks with the relation between them.

4.2 Trusted Platform Modules

A TPM [99] is a small tamper-proof hardware chip embedded in most recent motherboards. In order to be protected from software attacks affecting the OS, it has been designed as a stand alone hardware module. It is designed to resist to tampering attacks which means that it should be very hard for Class III (see Section 4.1) attackers and impossible for Class II and Class I attackers to perform some alterations to the device. Its specification [149] originated from the Trusted Computing Group (TCG) [19], a worldwide consortium involving the main actors of modern computing (HP, IBM, Intel, Microsoft, AMD etc.). It follows that TPMs are assumed to become a *de facto* standard component in future computers. A core concept elaborated by the consortium on top of the TPM is the notion of Trusted Computing (TC) which is now recalled.

4.2.1 Trusted Computing (TC) and the Chain of Trust

The basic idea is the creation of a chain of trust between all software elements in the computing system, starting from the most basic one, *i.e.*, the BIOS. In a TC scenario, a trusted application runs exclusively on top of trusted and pre-approved software and hardware. The trusted hardware parts that form the Core Root of Trust for Measurement (CRTM) are composed of the following Trusted Building Blocks (TBB): the CPU, the memory, the motherboard controller, the keyboard, the first software that is executed on the computer within the BIOS and finally the link between these components. The TPM does not require trust because it contains hardware protection, making it difficult to attack.

The chain of trust is derived using induction. As seen previously, the first software running on the machine is considered as trusted. This first software will read the next software to be executed on the machine and will generate a cryptographic hash (see Section 3.3) using the TPM representing the next software. It then will be able to verify that the hash corresponds to the fingerprint of the expected software. This assumes of course that the first software has in his data section an hash value representing the expected next software. By induction *i.e.* each software executing on the machine checking the next software, the chain of trust is therefore built.

Such a chain is represented in Figure 4.2 and is called the Root of Trust for Measurement (RTM) in the TC terminology.

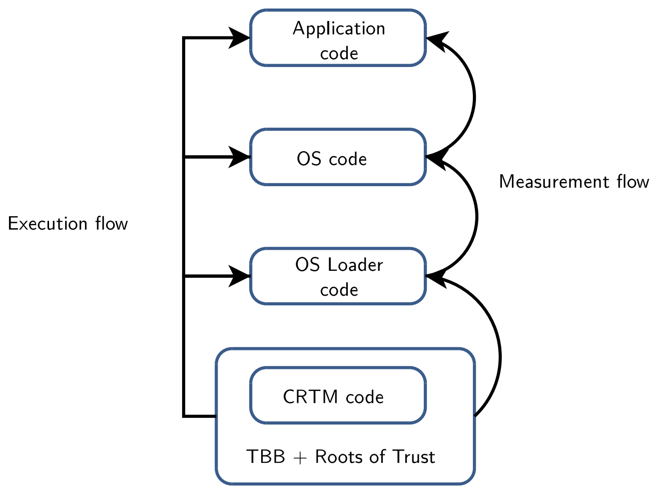


Figure 4.2: Root of trust for measurement.

4.2.2 TPM Components

In practice, the heart of the TPM chip consists of a cryptographic co-processor and two kinds of memory, one permanent (*i.e.* non-volatile) and physically shielded (*i.e.* resilient to interferences and prying), the other volatile. The *coprocessor* is a common architecture that comprises a secured random number generator and is capable of performing various cryptographic operations such as encryption, decryption, signature checking, 2048 bits RSA-key generation and SHA1 hashing. More precisely and according to the TPM specification [150], the cryptographic co-processor must be able to perform the following operations:

- Asymmetric key generation/encryption/decryption/signature (RSA) (see Section 3.2.2) with support for key length 512 bits, 1024 bits and 2048 bits.
- Hashing using the SHA-1 function (see Section 3.3.2).
- Symmetric encryption: for authentication and transport sessions, the mandatory mechanism is a Vernam one-time-pad with XOR (see Section 3.2.1).
- HMAC $H(Key\ XOR\ opad, H(Key\ XOR\ ipad, text))$ (RFC 2104) (see Section 3.3.3).
- A true random-bit generator used for key generations and nonce creation.

- The TPM may implement other algorithms as well, but this will be platform specific.

Memory

The TPM has two kinds of memory: one *volatile*, the other *non-volatile*. The TPM can also extend its memory using the memory of the computer, in which case it has to crypt this data using keys stored within the TPM or keys stored externally but encrypted with a key stored in the TPM.

The **volatile memory** is used to store Platform Configuration Register (PCR): a PCR value is 160 bits long and is able to hold an unlimited number of measurements by chaining the value to hash with the precedent hash value:

$$new\_PCR_i = Hash(old\_PCR_i||value)$$

These PCR values have to be reset at system start. The PCR registers 0-7 are reserved for TPM use, and the 8 to 15 are available for the operating system and application use.

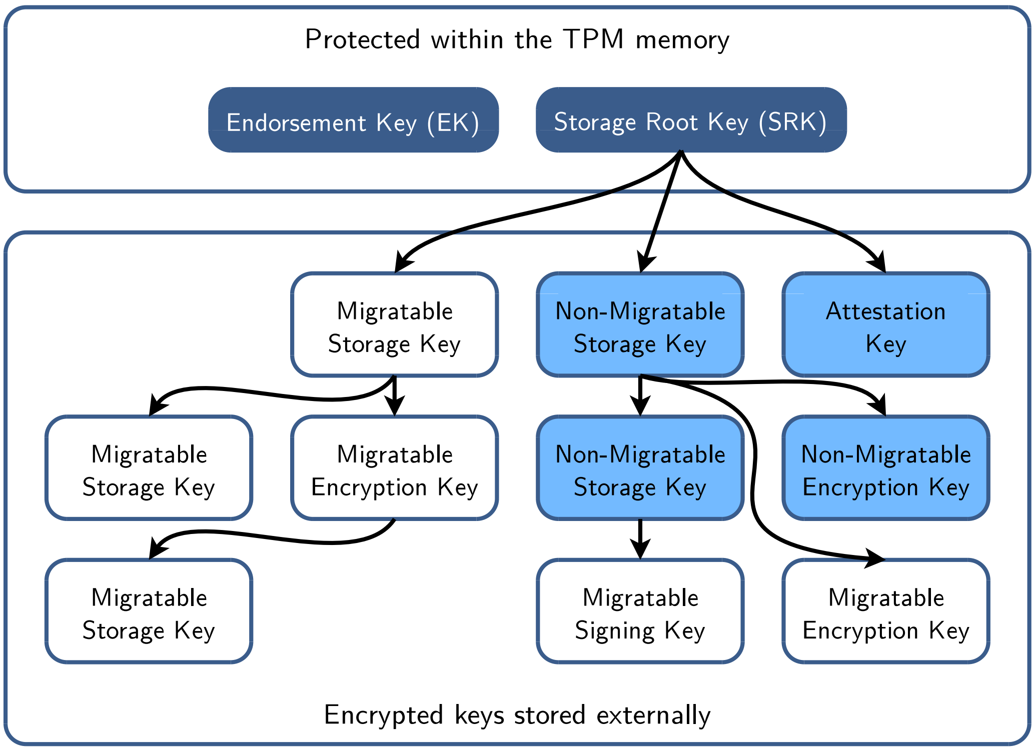

Protected within the TPM memory
Endorsement Key (EK)
Storage Root Key (SRK)
Migratable Storage Key
Non-Migratable Storage Key
Attestation Key
Migratable Storage Key
Migratable Encryption Key
Non-Migratable Storage Key
Non-Migratable Encryption Key
Migratable Storage Key
Migratable Signing Key
Migratable Encryption Key
Encrypted keys stored externally

Figure 4.3: key hierarchy in the TPM.

The **non-volatile memory** is shielded (resistant to interferences and prying) in order to protect data. This memory stores three kinds of keys (see figure 4.3):

- **Endorsement Key (EK)**: the TPM entity, typically the TPM manufacturer, generates the endorsement key (EK) and signs the credential. This must be a 2048 bits RSA Key. This key cannot be used for signing – it is only used when somebody wants to take the ownership of the chip.

- **Shared Secret**: a secret which is shared between the TPM owner and the TPM, this secret is defined when somebody with a physical presence (person present near the machine) takes the ownership of the platform.

- **Storage Root Key**: as the memory in a TPM is limited, the Storage Root Key is used to manage keys stored elsewhere in the computer (but encrypted with this key). This key is generated at the command `tpm_takeonwership`, and is associated with the owner of the TPM.
- **Persistent flag**: some values like the state register of the random number generator, flags for the Opt-In, etc ...

Other kind of keys exist such as the Attestation identity keys (AIKs) which is used to sign data but never for encryption. These are 2048 bit RSA keys, which can be stored inside or outside (encrypted) the TPM and cannot be moved to another TPM.

The TPM doesn't perform any monitoring nor control, measurements are done by the computer and then sent to the TPM. The TPM cannot control the execution of the computer nor its state. The owner can deactivate the TPM whenever he wants, he controls the use of keys as well.

Among all the built-in functions available in the TPM, the protocols defined in this thesis make use of the following procedures:

- `TPM_UnBind(msg,'K')`: decrypts message `msg` encrypted with a public part of a binding key `K`;
- `TPM_Extend(value,i)`: adds a new integrity measurement (`value`) to a PCR identified by its index number `i`. This operation is performed by a cryptographic hash function as follows:

$$PCR_i^{new} = H(PCR_i^{old} \| value) \text{ and } PCR_i^{init} = 0$$

- `TPM_Quote('AIK_ID',nonce,`i_1`,...,`i_k`)`: provides cryptographic reporting of PCR values (for the register indexes $i_1, \ldots, i_k$) signed with an AIK, *i.e.*, $SIGN_{AIK}(H(PCR_{i_1}, \ldots, PCR_{i_k}), nonce)$. It therefore permits the measurement (via a cryptographic hash) of the configuration of a remote platform identified by its AIK which is of prime importance for this work operating in the CC context.

4.2.3 Open-source Community and the TPM

The development of open-source tools for the TPM at the different levels of the Root of Trust for Measurement (RTM) allows open-source Operating Systems (OSs) to take advantage of the existing hardware present on the motherboard. The Figure 4.4 presents some of the exiting open-source software used to communicate with the TPM, it exposes as well where in the software stack these programs are used.

Open-source BIOS

`coreboot` is an open-source BIOS which includes TPM support [16]. However, when present on a mother board, the TPM is usually handled by the proprietary BIOS, without forbidding the use of open-source OS.

Trusted Boot

A boot loader is the software program which runs after the BIOS. It is responsible for loading and transferring control to the OS kernel software. GRand Unified Bootloader (GRUB) [17] is an open-source multiboot boot loader. The trusted boot version of GRUB [18], called `grub-tcg`, is a patched version of the GRUB boot loader. It adds the TCG measurement capability to the GRUB boot loader. The stage 1 of grub (Master Boot Record (MBR)) is measured by the BIOS, but the stage 1.5 (or stage 2) is measured by the trusted boot and sent to the TPM. The PCRs values are updated with the values corresponding to the GRUB software. It allows as well the integrity measurement for some key files within the File System (FS) such as `/bin/sh`, `/bin/ls`, `/bin/qemu-kvm`, `/usr/bin/python` etc.

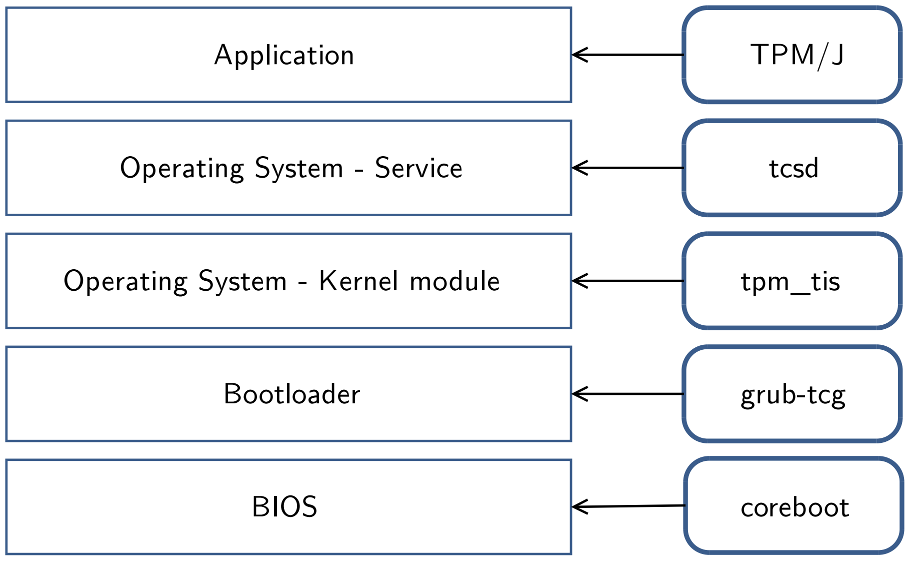


Figure 4.4: Open-source software solutions relying on TPMs – and their level in the boot sequence

TPM kernel module

The drivers for different TPM chips are already in the Linux kernel (starting version 2.6.12 for some manufacturers) and are thus available for all the open-source OS that rely on the Linux Kernel.

TrouSerS and `tpm-tools`

Trouser is the open-source TCG Software Stack. It includes the services to communicate with the TPM drivers using a service called `tcsd` to allow the `tpm-tools` to perform actions using the TPM such as taking ownership or retrieving the public key of the EK key.

The `tpm-tools` software contains different command line programs to set up the TPM and to seal data. These tools are limited and do not allow the user to perform all the possible actions proposed by the TPM.

TPM/J

The TPM/J is a `Java` library that provides communication with the TPM for `Java` programs. It allows the user to send all possible commands to the TPM, for example to read or modify the PCRs values or to use the keys stored in the TPM to encrypt, decrypt, verify or sign data.

TPM emulator

The TPM emulator is a program that can simulate one or more TPMs on the machine. This emulator is based on the specifications of the TCG and contains all the operations that a normal TPM would do. The TPM creates a virtual device on the computer which is accessible using TPM/J, or TrouSerS. It can be used as well to allow a VM to use a TPM, *e.g.*, Xen hypervisor has such option.

With the open-source software, it is possible to interface most of the functions provided by the TPM devices, they allow communication with the device as well as the set up of the chain of trust certifying that the OS has not been modified.

4.3 Open issues in Hardware Security

4.3.1 No Unbreakable Hardware

Every chip, even the most heavily protected one, is not impenetrable by Class III attackers using highly sophisticated lab equipment and a lot of resources. However different protections such as shield make it very costly for an attacker to penetrate. The cost of an attack on a protected hardware should be higher than the gain that the attacker would obtain in case of successful extraction of data.

For example Christopher Tarnovsky [148] managed to reverse engineer a TPM chip from STMicroelectronics by removing the protection layers and to extract the private key stored inside the device. But the price of a TPM chip is very low compared to the cost of extracting information from it, unless the secrets stored in the TPM are very valuable. It makes the hardware protection given by the TPM better than an unprotected machine. One can even imagine a TPM chip with very high protection using for example an autodestruction mechanism in case any tamper is detected. Such techniques exist using power a battery within the chip to keep the detection system activated even when the chip is powered off.

The verification of the integrity of a machine based on the chain of trust and using a TPM is vulnerable to Time Of Check Time Of Use (TOCTOU) attacks. Indeed, between the computation of the PCR values and the actual check, the system might have been compromised without performing any PCR modification, *i.e.* letting the PCR values correct on a compromised system.

4.3.2 User's Privacy Concerns

The TPM allows tied selling, forcing the user to use only one kind of OS. Therefore open source community has not been very enthusiastic about the TPMs due to the fear of loosing control of one's computer. Recently the German Federal office for Information Security (BSI) [1] has warned that combination of TPMs and Windows 8, in case of a misuse of hardware or OS, might result in a permanently unusable system. This is however not an issue with the TPM itself because even in version 2.0, according to the specifications, the TPM chip is only a passive piece of hardware, *i.e.* it does not have any control on the machine and acts as a standalone hardware. However based on the TPMs' measurement the OS, the boot loader or the BIOS can choose not to launch the next level, *e.g.* a BIOS, detecting that the legitimate Microsoft boot loader is not present on the disk, might choose not to launch it and to crash, preventing the installation of any other OS.

In another register, Intel developed a hardware protection mechanism against thievery embedded in the professional versions of their processors [95]. This technology embeds a 3G communication device within the Intel processor with its own operating system allowing the control of the laptop/computer if in range of a 3G antenna even if powered-off. Used by an attacker, this technology might lead to the disclosure of information contained in the user's computer that even an encrypted hard-drive using a TPM cannot protect. Indeed, during execution of the machine the disk is unencrypted on-demand, therefore, controlling the CPU allows the attacker to decrypt any part of the disk he wants.

The need for hardware protection is indisputable for better security. However it needs the trust of all the actors involved, which includes the users. However, these specifications are not enough to ensure such guarantee, only open hardware *i.e.* the schematic plan of the netlist and gates contained in the die of the chip.

4.3.3 TPM protections for CC platforms

The open specifications allow already to verify that the enumerated functions do not provide, in theory, possibilities for the manufacturer to gain control of the machine. Being a cheap and widely used hardware chip, TPM is good candidate for increasing the security of a machine at a low cost. Already

[1] `https://www.bsi.bund.de/DE/Presse/Pressemitteilungen/Presse2013/Windows_TPM_Pl_21082013.html`

deployed even on servers, this TPM chip can be used by the nodes of the Cloud Computing (CC) providers to increase the security level of the service. The Chapter 7 will present how CertiCloud uses TPMs to increase user's security.

Chapter 5
Software Protections: from BIOS to Applications

Contents

Software protection can be applied at different levels in Cloud Computing (CC): at the Operating System (OS) level and at the application level. This allows to secure the different types of Clouds, IaaS, PaaS and SaaS.

5.1 Software Security: Operating System Level

5.1.1 Anti-virus and Malware

Malware is a specific category of software which has a malicious behaviour, it is a general term including viruses, worms, Trojans and other forms of programs which act in an autonomous or semi-autonomous way to attack computer systems in a large scale.

Viruses

A virus is a piece of code that infects another file *i.e.* attaches itself to another program and propagates when the file is sent to another computer typically using the network or the intervention of the user *e.g.* a infected file might be copied on a USB key and transferred to another computer.

Worms

Unlike a virus, a worm does not infect other files which means that it is a standalone program. It sends itself to other computer systems typically using a breach in a remote system.

The purpose of these types of malware is to reproduce and to install malicious payloads. These payloads are codes that execute on the machine and that can be harmful to the machine itself or can only open a backdoor on the system for future intrusion or commands.

Botnets are composed of a large number of machines that are controlled by malicious entities to perform attacks on other systems. They are usually created using worms or viruses.

Anti-virus

To detect the viruses and prevent spreading, anti-viruses can be used. An Anti-virus is a program running directly on the machine to protect and scan principally the files of the OS and the programs which are executed in order to detect viruses. It uses a database containing the fingerprints of known viruses and malware.

The main issue of anti-viruses is that they rely on a database to find the malware, which means that in case of the attack of an unknown virus, it cannot detect it without updating its database. This requires permanent surveillance of the network attack by the antivirus software companies to react fast enough to contain the attack. And during the phase of fingerprinting, the new malware has the time to spread and potentially harm systems.

5.1.2 Firewall

Firewalls are network protection against intrusion. They can be located on the machine which needs to be secured or directly on the router to protect the whole network. Network connections are made using network addresses on a port. A server is waiting for connection on a specific port number by asking the operating system to open it. The firewall is protecting the network by filtering network communications through inspection of the source address and the destination address as well as the source port and the destination port. Predefined rules allow to block the traffic from, for example, a web service which should be only available to the local network.

5.1.3 Intrusion Detection Systems

Intrusion Detection Systems (IDSs) prevent attackers from accessing a computer system. There are two kinds of IDS [129], network based or host based. The network based IDS are detecting abnormal behaviours on the network, they are reporting any unexpected action *e.g.* high traffic at 2 *a.m.* when no one is at the office. They have as well in their databases known scenario of different attacks. The second kind of IDS is host based: it tries to detect any suspicious behaviour using the monitoring of different components of the OS such as log files, file modification, network access, etc.

An interesting tool-suite in this last context is Tripwire [104] which periodically computes checksums of the main system files and folders. These checksums are then compared with reference values stored in a remote database hosted on the trusted computer.

The IDSs are usually only used for reporting intrusion and do not stop the attacker directly. The main issues with these systems are the false-positive *i.e.* the IDS reporting a legitimate user as an intruder due to an unconventional comportment.

5.2 Software Security: Application Level

5.2.1 Anti-debugger

An anti-debugger is a piece of software that can detect if it is being analysed by a debugger. This detection is performed by studying the side effects provoked by the debugging techniques. It is usually integrated within the software to protect and has different ways of handling the analyser. It can exit, crash, or respond in a more subtle manner like executing a dedicated part of the program which does not correspond to a normal execution, *e.g.*, an infinite loop performing actions that look legitimate.

Anti-debugging methods can detect the use of `ptrace` (used by debuggers in UNIX environment) like in [143], or like in [72]. They detect differences in execution time as the debugging process adds delays that can be detected by recording the time of execution of a program. The `Python` program in Listing 5.1 demonstrates the detection of a debugger. It presents two possible kinds of detection, the first detects if the `trace` function has been set and the second uses the fact that the debugger provokes a delay in the computation that might be detected.

Listing 5.1: Anti-debugger concept in `Python`.

```
import time
import sys
MAGIC_NUMBER = 5
def debugger_detector_sys():
    if sys.gettrace() :
        print "debbuger"
def debugger_detector_time():
        t1 = time.time()
n       for i in range(10):
                j = i**2 / 3.14
        t2 = time.time()
        for i in range(100):
                j = i**2 / 3.14
        t3 = time.time()
        if (t3 - t1) / (t2 - t1) > MAGIC_NUMBER:
                print "debugger␣detected"
debugger_detector_sys()
debugger_detector_time()
```

5.2.2 Anti-disassembly

In low level machine code, the size of the instructions are not always the same allowing methods to protect the code against automatic disassemblers. The Figure 5.1 is showing how the anti-disassembly is working [110],[143]. Indeed machine code can be interpreted differently depending on the starting address of the code, and combined with jump instructions to set up the entry point of procedures, a static disassembler might be unable to retrieve the initial assembly instructions.

To fool disassembler using the information given by the jump address, the anti-disassembly can either store the jump address in a variable, or insert false jump with dummy addresses within the code.

These techniques are efficient against static disassembler, however they are not working with resilient dynamic disassemblers like `gdb` [5].

5.2.3 Anti-modification Techniques

Anti-modification techniques are using checksums function to verify that the code has not been modified. This verification is performed at run time and leads in case of the detection to an exit. This is as well effective against debuggers as they often insert code to debug the program.

For instance in [32] the binary code is designed to ensure that the Control Flow of executed programs is correct. This goal is achieved by adding stack protection mechanisms or encrypting the returned function pointers. Yet it was also proved that this technique does not scale and is not resilient against attackers having privileged (*i.e.*, `root`) access to the system.

5.2.4 Self-modification for Code Obfuscation

At the assembly level, obfuscation techniques are used to protect the code from being reverse engineered. Obfuscation (see Chapter 8) modifies a program to produce a “virtual black box”, *i.e.* looking at the

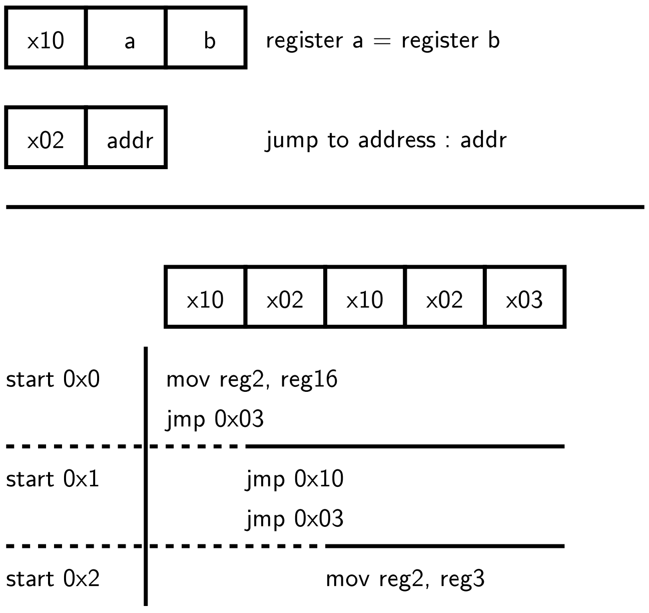


Figure 5.1: Illustration of the protection.

source of the program should not give more information than its execution.

There are multiple techniques to obfuscate code at the assembly level [114] such as:

- Kanzaki's method [101]: replacing some parts of the code with dummy instructions, before execution restoring them to the original code, and replacing it with dummy code again at the end.
- Madou's method [113]: consisting of using a PRNG and the code to feed the function XOR. The obfuscated code is xored again at run time using the same PRNG with the same seed.
- Burneye's method [152]: is quite similar to Madou's method but instead of using a PRNG it uses an encryption cipher such as RC4, with the possibility of specifying a secret that will be asked at runtime to decrypt the binary.
- Shiva [119]: is a GNU/Linux executable encryptor which encrypts the binary using AES algorithm. Shiva is dividing the program into blocks and encrypts all the blocks independently, the binary is then never completely decrypted. The keys are different for every binary and the key reconstruction functions obfuscated using different obfuscate methods.
- Encryption of using MIPS emulator [160]: are using encryption techniques on blocks of code to avoid the reverse engineers to understand the algorithms.

All of these programs and techniques are used to protect software and OS from intrusion and reverse engineering. Software protection techniques are used as well by malware programmers to hide the behaviour of the program and to prevent detection by antiviruses.

5.3 Open Issues in Software Protection

Software Protection at the OS level is well developed and quite effective, indeed without firewalls and anti-viruses there would be more attacks on the different computer systems. However as long as the computer systems become complex, the number of breaches in software will increase, therefore leading to a greater need for such tools to protect the OS from intrusions. And the main problems with these protections are their lack of reactivity compared to the fast contamination of network in case of the attack of a worm as well as the false-positive and false-negative errors coming from the detection system.

Software Protection at the application level allows to protect applications from reverse engineering but no perfect method exists to defeat attackers with knowledge and funds and to prevent them from extracting the secrets within the code. However, it can be considered safe for a short period of time, the time for attackers to find keys embedded in the code used for self-encryption.

The two levels of software protection often compete against each other as obfuscation techniques are often applied to malware creation in order to avoid detection from OS protection software. For example, the Flame malware [41] was detected approximately two years after its release. These malware however require huge development and cannot be programmed by single individual but rather by secret services.

Part II

Platform Protection in a Cloud Environment (CertiCloud)

Chapter 6
Security of Protocols: Methods and Tools

Contents

This chapter describes briefly the methods used by automatic verification tools to prove the security of a protocol for a given model of communication. It presents as well the two tools used to verify the protocol designed during the thesis that will be described in the next Chapter 7.

6.1 Automatic Analysis of Security Protocol

Despite the relatively small number of communication messages used in network protocols, *e.g.* Kerberos is using 6 messages for the authentication of a user, it is very difficult to design them correctly, and their analysis is complex. Moreover, assessing the security of protocols requires more than showing their robustness against a few use cases. The canonical example in this context is the famous Needham-Schroeder protocol [125], that was *proven* secure by using a formalism called BAN logic. Yet, seventeen years later, a flaw was found [111] using an automatic verification tool (Casper/ FDR [112]). It was not detected by the original proof because of different assumptions regarding the intruder model. This example illustrates the fact that automatic analysis is now considered **mandatory** for the security validation of cryptographic protocols.

6.2 Verification of Models

Model verification is based on logic which is a wide subject. A small set of existing logics, summarized in Figure 6.1, are:

- Propositional logic is based only on propositions. A proposition can be either `true` or `false` and can be combined to build more complicated propositions. The operators used to combine the propositions are the following: implication $\rightarrow$, or $\vee$, and $\wedge$, negation $\neg$. Propositional logic is not expressive enough therefore other logics based on propositional logic has been developed.
- The predicate logic (as well called first order logic) adds to the propositional logic the variables and the two following concepts: *for all* $\forall$ and *there exists* $\exists$. These new operators are applied on the variables which are feeding the propositions *e.g.* $\forall x \exists y P(x) \rightarrow Q(x)$.
- Temporal Logic includes the notion of time in the formulas to be more expressive in the model one would like to verify. In this context, two subclasses of temporal logic have been defined:

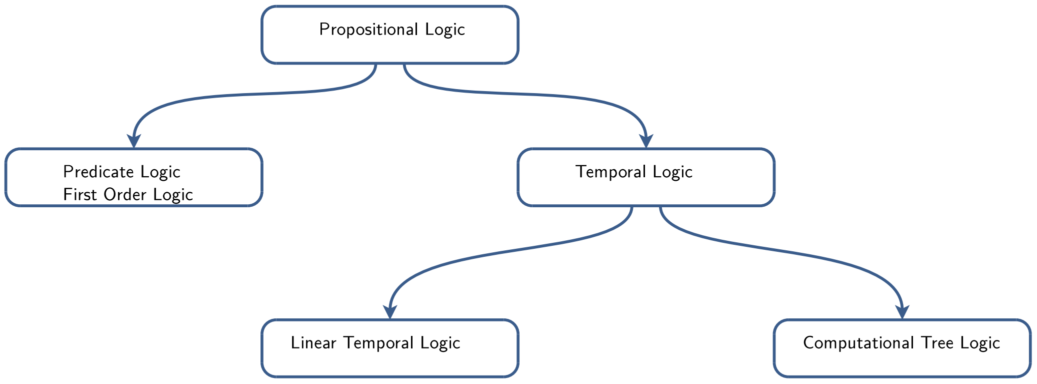


Figure 6.1: Hierarchy of Logics.

- The Linear Temporal Logic: it models time as a sequence of states which can then be represented as a graph. This sequence of states is called a computation path.
- The Computation tree logic is a branching-time logic and can be represented as a tree structure in which the future is not determined, *i.e.* any path in the tree is a possible future.

6.2.1 Proof-based vs Model-based

Verification of logical propositions can be either proof based or model based. In the case of proof based verification, the system is described as a set of formulas Γ and the proposition to be proved is another formula Φ, and the verification process consists of finding a proof that shows $\Gamma \vdash \Phi$. An example of a proof in propositional logic is presented in Figure 6.2.

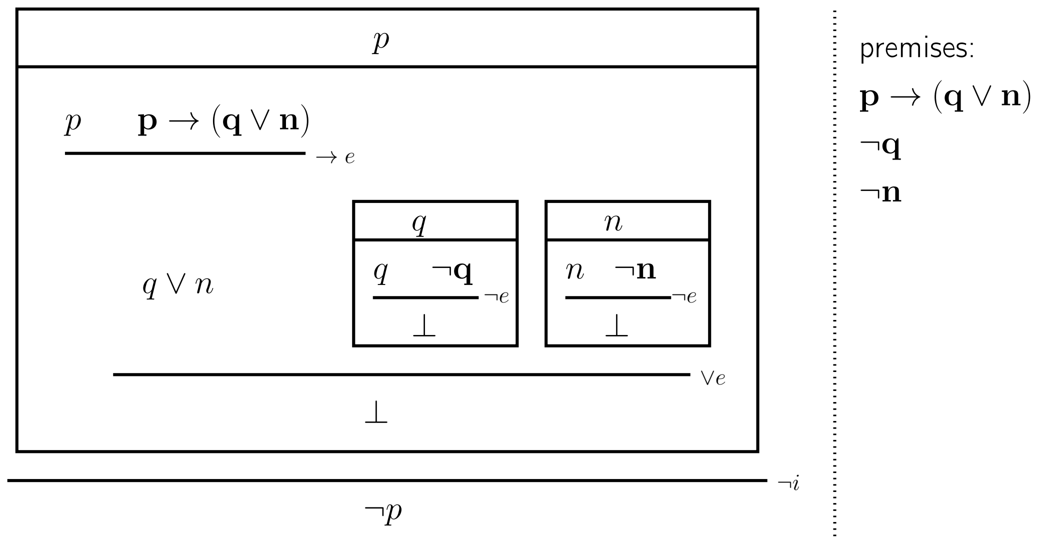


Figure 6.2: Proof that the weather is bad, assuming that I do not swim in the sea and that I do not walk outside

This is the proof that $p \rightarrow (q \vee n), \neg q, \neg n \vdash \neg p$ if p stands for the weather is nice, q stands for I walk outside and n I swim in the sea, then $p \rightarrow (q \vee n)$. If the three premises following are true: $p \rightarrow (q \vee n)$ which means a nice weather implies that I am swimming in the sea or I am walking outside, $\neg q, \neg n$,*i.e.* I do not swim in the sea and I do not walk outside, the proof in Figure 6.2 demonstrates that the weather is bad, *i.e.* $\neg p$. The proof has to be read from top to bottom and is constructed using rules of natural deduction which can be found in Appendix E.1.

Verification by model checking consists of verifying that a model $\mathscr{M}$ satisfies the formula Φ *i.e.* $\mathscr{M} \models \Phi$. The difference between proving and satisfying is that **proving** requires a proof like in Figure 6.2 and **satisfying** tries all the possible values respecting the premises to see if Φ is valid in any case. Model checking tools are usually based on temporal logic introducing the idea that a proposition is not statically true or false in a model, instead there might be multiple states where the formula is true and other states in which the formula is false.

6.2.2 Model checker for Protocol Verification

In the context of protocol verification, the theorem proving is time consuming and requires considerable expertise. Moreover proof based verification provides less support for error detection in the case of flawed protocol. This is why model checkers are often preferred over theorem prover for protocol verification.

In the model checking family there exist three classes of tools:

- The tools trying to prove the correctness of a protocol (NRL [118] and ProVerif [13]) and do not have termination guarantee.
- The tools trying to find attacks against the protocol but with a bounded number or runs of the protocol.
- Hybrids trying to find both proofs and counterexamples, which are sometimes able to establish the correctness of a protocol even for an unbounded number of runs, gives a counterexample and cannot guarantee the termination (Scyther [66]).

The tools used in this work will be presented in the next sections, they both used the Dolev-Yao [74] model for the communication channels. In this model the communication channels are supposed insecure, which means that the attacker has access to all the messages, and can modify them or extract parts of them to get information. He is as well allowed to start new instances of the protocol to acquire more information by having new messages generated from legitimate entities.

6.3 Scyther

The Scyther [66] model checker implements an unbounded model to perform protocol verification. Its syntax is simple and easy to use. Many roles can be defined and the message sent and received has to be specified for each player. Variables (which can be `fresh`, *i.e.* chosen by one of the player) such as nonces and timestamps can be used within the protocol description and, of course, symmetric and asymmetric encryption.

Scyther has a customizable bound, but can detect whether or not a lower bound is reached. The specification of a bound allows to guarantee termination of the browsing of the search space. There are three possible outcomes for this search: either the pattern of an attack is found and presented to the user, or no attack has been found and the bound has not been reached or finally no attack has been found but the bound has been reached, an attack might therefore exist and it might be detected with an higher bound.

6.4 AVISPA

AVISPA [1] is a model checker using different backends to perform model checking. The model to verify has to be written in High Level Protocols Specification Language (HLPSL) which is the language used by AVISPA to specify the protocols and properties. The HLPSL specifications are based on

Lamport's Temporal Logic of actions [108]. AVISPA then converts the HLPSL specifications into an Intermediate Format (IF) using `hlpsl2if` which can be read by the different model checkers that have been developed within the AVISPA project. The available model checkers are:

- Constraint-Logic-based Attack Searcher (CL-Atse): translates the transition relation contained in the IF file into a set of constraints used to find attacks and produces nice human-readable attack descriptions.
- On-the-fly Model-Checker (OFMC): builds on the fly an infinite tree defined by the protocol analysis problem. OFMC can be used for efficient detection of attacks in a protocol but as well for verification for a bounded number of sessions.
- SAT-based Model-Checker (SAT-MC): constructs a propositional formula using a bounded unrolling of the transition relation specified by the IF. The propositional formula is then given to a SAT solver to verify the satisfiability.
- Tree Automata based Automatic Approximations for the Analysis of Security Protocols (TA4SP): constructs a tree to perform the computation of the intruder knowledge (using some approximations).

The validation with AVISPA of the protocols developed was performed using all the different backends presented.

The syntax of the HPSPL file consists of different roles regrouping the actions of the users, the sessions and the environment. And finally the goals are described at the end with option such as `secrecy_of` or `authentication_on` to verify if a secret cannot be leaked or if the authentication of a user cannot result in the authentication of the attacker.

6.5 Conclusion

The Dolev-Yao attack model [74] used by the model checkers is the worst scenario *i.e.*, that the attacker has full control over the network and can start many different communications to gather parts of messages. This model however respects the "black box" property of the cryptosystems used. Trying all possibilities according to the model (often within bounds) is a pledge to the security of the protocols, specially when multiple protocols are involved. Indeed, verifying that all the communication coming from a server to find a flaw is nearly impossible. It would require to verify all possible messages sent by different parties increasing the number of possible states, specially if keys are reused multiple times. However, when dealing with security protocols it is mandatory to verify their validity using automatic tools. They try to break the protocol using all kinds of possible forged messages using old message parts.

In this dissertation the proposed protocols have been validated against AVISPA and Scyther in order to cover not only the common attacks analysed by these frameworks but also the specific ones determined by their underlying analysing models.

Chapter 7
The CertiCloud framework: a Novel TPM-based Approach to Ensure Cloud IaaS Security

Contents

The CertiCloud framework focuses on the security aspects of the third category of cloud services, i.e., IaaS platforms (Section 2.2.3) and more precisely on confidentiality and integrity issues. In Section 5.1, two levels of software security are presented, CertiCloud can be categorised as an **OS level** protection software, it is designed to perform verification of OS of both the resources of the CC provider as well as the VMs' of a user.

7.1 Introduction

Whereas security problems are considered in a vast majority of the literature relative to CC, only few articles propose to tackle these questions from a user's point of view, assuming I am using an IaaS Cloud platform to deploy my system:

- Can I be sure that my environment and its associated data remain confidential on such a shared platform?
- Is there a way to ensure that the cloud resources are not corrupted?
- Once deployed, is there a way to assert on demand and trustfully the integrity of my environment to detect undesired system tampering (typically by means of rootkit or malware)?

One typical scenario where CertiCloud can be used is to detect if a resource has been modified by an attacker to spy on other VMs like in Figure 7.1. Indeed the attacker might use vulnerabilities in the virtualisation software to gain privileges.

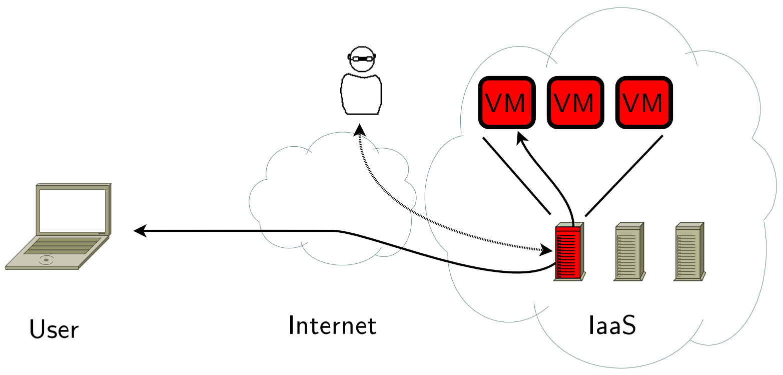


Figure 7.1: Attacker having the control of a Cloud resource

Whereas perfectly legitimate, these questions are not answered by the current cloud providers. On the contrary, the contributions of this thesis permit to tackle these questions via the proposal of CertiCloud framework, a novel approach to protect IaaS platforms that rely on the concepts developed in the Trusted Computing Group (TCG) together with hardware elements, *i.e.* Trusted Platform Module (TPM) to offer a secure and reassuring environment (see Section 4.2).

The heart of CertiCloud consists in two security protocols relying on TPMs– TCRR (Section 7.5) and `VerifyMyVM` (Section 7.7):

- TPM-based Certification of a Remote Resource (TCRR) asserts the integrity of a remote resource and permits to exchange a user-defined symmetric key which can be used for various cryptographic purposes, from the protection of network communications to data encryption. In the IaaS context, it ensures that only the remote resource with which the user is communicating using the TCRR protocol can interact with the ciphered data.
- The second protocol, `VerifyMyVM`, authorizes the user to detect trustfully and on demand any tampering considered as abnormal on its running VM.

On top of these protocols, CertiCloud relies on the installation of a virtualization framework on the Cloud resources that will be discussed in Section 7.8.

7.2 Related Work

Until the recent generalization of TPMs, checking the integrity of a system was limited to software approaches. In practice, one generally verifies that the running system has not been modified or altered (by malicious acts or not). At this level, malware detection techniques (Section 5.1.1) or checksum-based approaches can be used.

	Terra	SecMilia	TCCP	Bastion	CertiCloud
VM	✓	-	✓	✓	✓
TPM	×	✓	✓	✓ ✓	✓
Automatic Verification	n.a.	✓	×	n.a.	✓
CC	×	×	✓	×	✓

Table 7.1: Summary of some existing techniques to secure resources.

We will see that CertiCloud reuses the concept of Tripwire [104]) (see Section 5.1.1), yet adapted to the functionality offered by the presence of TPMs. More generally, pure software-based approaches do not allow to fully trust a remote platform. It is therefore mandatory to rely on hardware components with computing and cryptographic capacities, such as the TPMs. This low-cost, tamper proof hardware is now available on most modern motherboards. Its usage spreads worldwide. For example, Microsoft Bitlocker [120] uses it to perform a full disk encryption. It is also emulated in the Xen Hypervisor OS [22] to enable VMs to use it. In all cases, TPMs offer novel perspectives for checking the security of a system, whether local or remote. It follows that the idea to use TPM to secure cloud services is not new. It has been evoked by the TCG in [19], yet without concrete details. Moreover, in [69], the TPM and its emulator counterparts are used by the Xen Hypervisor to secure the different running VMs. Different environments are then provided to the user, which are protected from each other. Yet no concrete validation or implementation was proposed. Recently, an infrastructure called Trusted Cloud Computing Platform (TCCP) have been proposed in [140]. Similarly to what is done in Terra [83], TCCP enables IaaS providers such as Amazon EC2 to provide a closed box execution environment that guarantees confidential execution of a users' VMs. Whereas this paper does not consider the same perspective addressed in CertiCloud, it perfectly illustrates the recent attempts to propose protocols that exploit the remote attestation capabilities of the TPMs. Yet like most of the proposals in this domain, the described protocols are not validated nor analysed using protocol checker therefore the infrastructures built on top of them might be insecure.

On the contrary, the approach developed in this dissertation relies on automatic protocol verification to ensure security from a communication point of view and focuses on providing security for a user of the cloud services.

Our design methodology is close to the one used by Munoz al. in [123] where the SecMilia framework, which relies on a TPM-based protocol to achieve the migration of an agent (Java code) `A` from an agency "AgA" (using a TPM to ensure the security of the agency) to an agency "AgB". More precisely, the agencies verify the states of each other using their TPMs. However SecMilia does not provide to a cloud user a way to answer the questions mentioned previously (Section 7.1).

Other systems develop security directly at the processor level, like in [60] where the full processor is designed to be secure. It is of course more reliable in terms of security but requires higher investment than a passive chip such as the TPM.

The Table 7.1 presents the different attributes of the existing techniques to secure resources presented before.

7.3 Context and Assumptions

In this work, we assume an IaaS model with following entities:

- The **Cloud provider**, which is the organization that offers the IaaS service. It owns and manages the **IaaS platform** composed of a set of computing resources running a virtualisation framework able to deploy Virtual Machines (VMs), *i.e.* a *hypervisor*. The platform is interfaced by an access front-end, eventually distributed.

- A **trusted party** which provides the user with the correct PCRs values and **regulates the physical access** to the Cloud provider resources. This role can be played directly by the Cloud provider. It has as well the role of a **Certificate Authority (CA)** trusted by both the user and the cloud provider and attests the validity of certificates and keys used on the platform. In particular, each AIK and EK signed by the CA attests that they are tied to valid Endorsement, Platform and Conformance credentials, *i.e.*, the corresponding TPM and system. In parallel, each user has a certificate signed by the CA that attests its identity.
- The **User** owns one or more VMs to be run on the resources "shared" by the cloud provider.

In the sequel, the following hypotheses are assumed:

(H1) Each computing resource hosts a TPM *owned* (as described in Section 4.2) by the cloud provider.

(H2) Physical manipulations of the TPM might not be detected by the CertiCloud framework. It is therefore assumed that the entity which has physical access to the TPM and to the machine is trusted.

Finally, it should be noted that no specific measures have to be taken to ensure the security of the front-end as the protocols designed in CertiCloud assume that the communication medium is not secured and potentially subjected to *man-in-the-middle* attacks, indeed the front-end does not play any role in the protocol itself. It is however preferable for the user that the front-end is available, as an attacker controlling this entity would be able to prevent the user from reserving computing nodes on the CC platform.

7.4 Preparation of the VMI and the VM request

Prior to the deployment of the VM, different steps are required to prepare and transmit the VMI. First the user has to create its VMI using the desired environment and configuration, then he has to compute the hash values of the important files. He will be able to verify their integrity after the VM will be deployed. And finally he has to encrypt the VMI using a symmetric key.

These steps are presented in Figure 7.2.

The two next steps (in Figure 7.3) which are the VMI transfer and the VM request do not change from a user point of view.

This preparation allows the user to create a personalised system. Therefore, he is aware of all the software that will be running on his VM. It is common for Cloud provider to allow users to build their own environment, for example Amazon provides tools to bundle and upload the VMI created by the user (`ec2-bundle-image` and `ec2-upload-image`). However if a user prefers not to generate a VMI by himself, the **trusted party** might provide VMIs as well as hash values corresponding to the files within the VMI for integrity verification using `VerifyMyVM` (see Section 7.7).

7.5 TCRR

One of the first issues we propose to tackle in the CertiCloud framework is the definition of a network protocol that permits to certify that a remote resource will consistently behave in expected way, *i.e.*, running the non-compromised OS.

As in the Trusted Computing (TC) context, the challenge is here to allow the user to verify that only authorized code (BIOS, MBR, kernel, OS etc) runs on a remote system. The protocol described in this section, entitled TCRR (TPM-based Certification of a Remote Resource), provides this functionality. TCRR also exchanges a symmetric key between the user and the proven secure node.

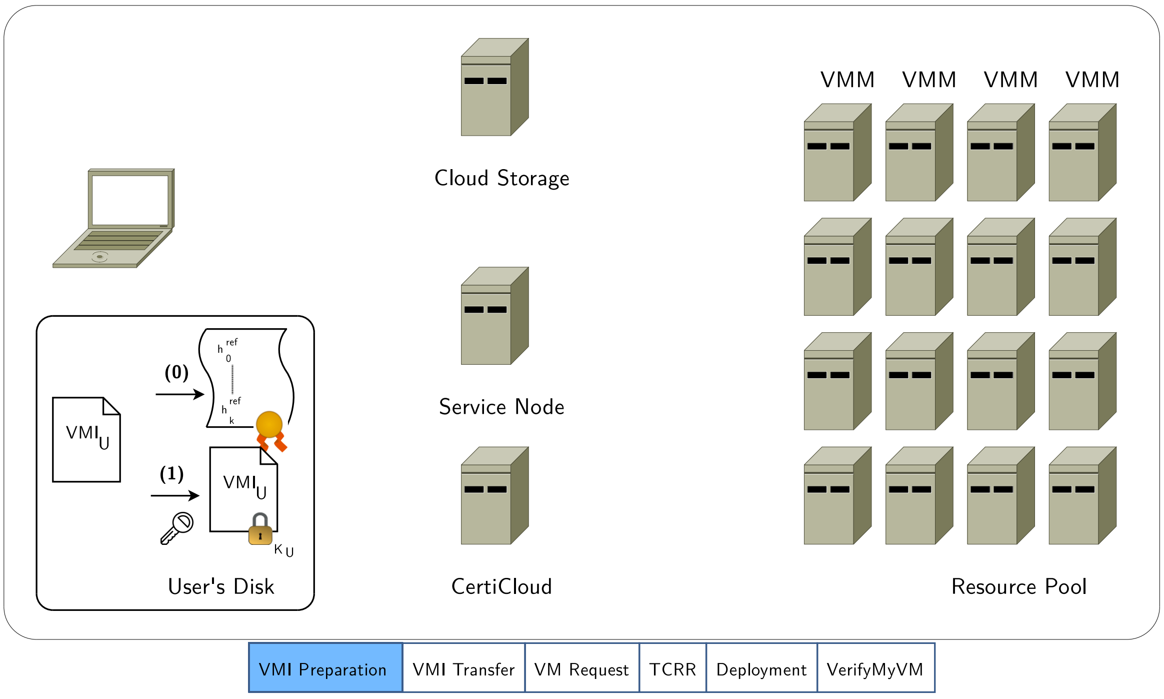


Figure 7.2: Overview of the CertiCloud framework (preparation of the VM).

The protocol involves three actors: the *user* U, the remote resource or *node* N, and its associated TPM. The position of the TCRR protocol in the CertiCloud framework is presented in Figure 7.4 and the successive messages exchanged in TCRR are described in Figure 7.5. Note that we assume (1) that each actor owns the digital certificates of other involved entities, signed by the trusted CA and (2) that prior to the protocol initialization the user has the PCR values he considers to reflect a secure system.

There are two phases in the TCRR protocol that are described in the following sections.

7.5.1 Node Integrity Check (msg 1-4)

U is sending to N a nonce n_1 and the identity of N signed using his key. The signature attests the identity of the user to N, which can then send to the TPM the received nonce $n1$, a newly generated one n_2 and its identity.

This will serve as a challenge for the further `TPM_Quote` function as the TPM replies by reporting to N the actual PCR values signed with its AIK (*i.e.*, the result of the 'quote' function – see Section 4.2). Then N forwards to U a signed version (with its key) of the received message together with n_2. At the end of this phase the following properties apply:

1. U checks the correctness of the successive signatures using the certificates provides by the CA. U checks if the values of the nonces have not been modified. This authenticates N and the associated TPM. It also guarantees the integrity of the transmitted messages.

2. The equality between the expected and the received PCRs values proves to U that the remote resource at the time of the verification is the non compromised one.

Due to (H2), after the successful verification of the fourth message, the user U can trust the remote node to run the configuration he considers as secure. Furthermore, from this point, **the node N and the TPM are considered as a single entity**.

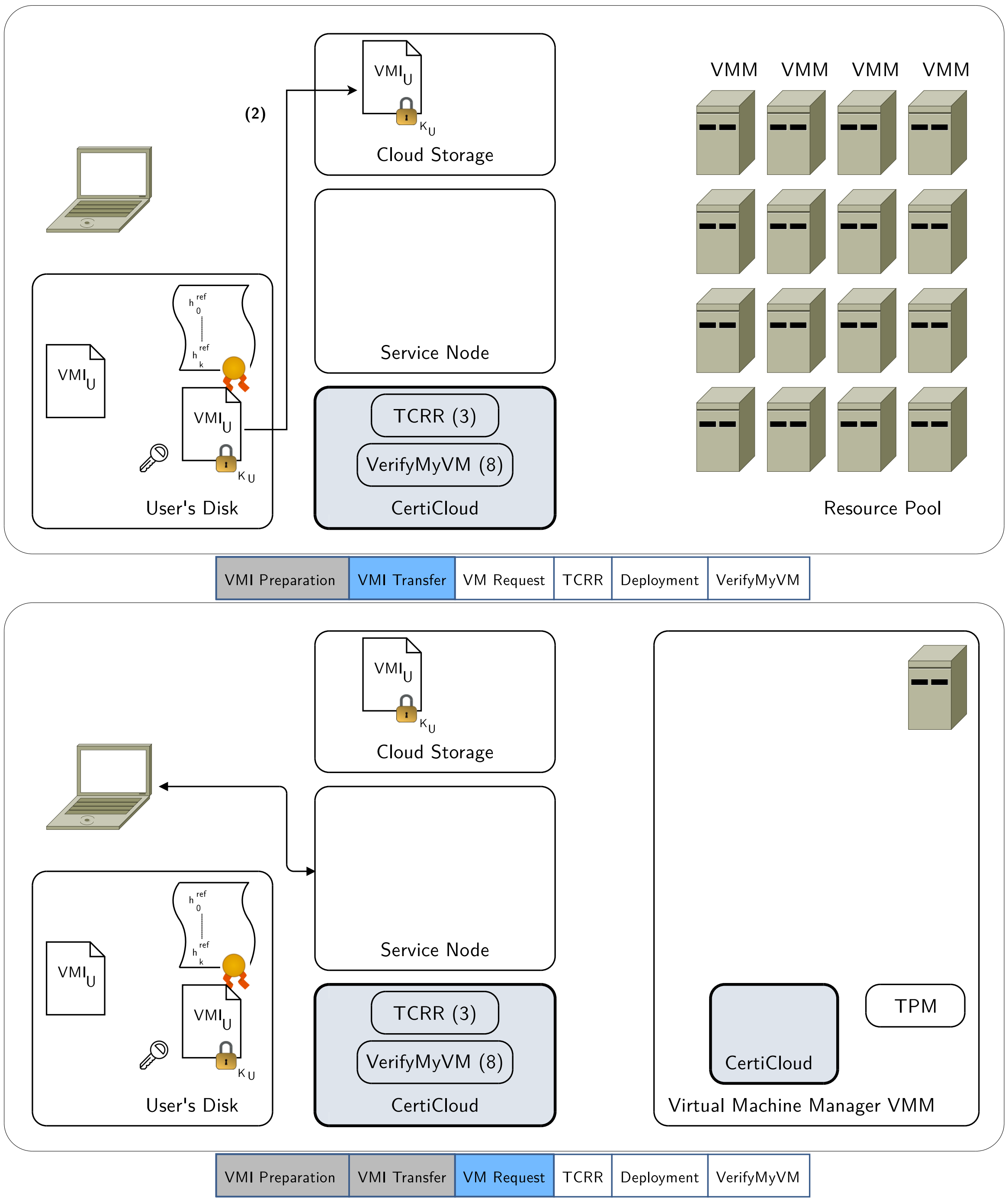


Figure 7.3: Overview of the CertiCloud framework (transfer of the VMI and request of a VM).

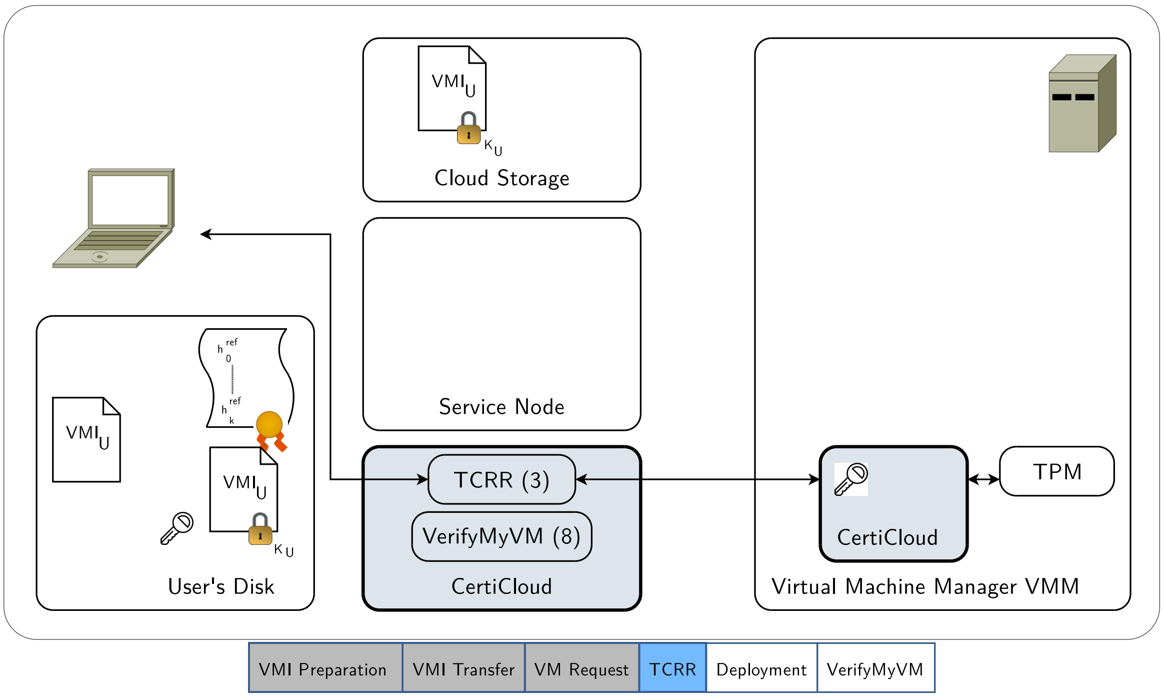


Figure 7.4: Overview of the CertiCloud framework (TCRR).

7.5.2 User Session-key Exchange (msg 5-8)

U uses the TPM encryption key `subEK` to cipher a message containing the two previous nonces n_1 and n_2 (generated during the first phase of the protocol), a freshly generated one n_3 and a private session key $K_{session}$. U sends a signed version of the encrypted message to N. N then checks the signature, asks the TPM to decrypt the message and returns to U the nonce n_3 encrypted with the session key.

7.5.3 TCRR Protocol Validation

The TCRR has been analysed and verified using both the AVISPA [1] and Scyther [66] (see Chapter 6). The Listing 7.1 and the Listing 7.2 present the results of these verifications which prove its correctness according to the model.

In these verifications the following constraints have been verified:

- The protocol ends at the same time for all the actors. This corresponds to the `authentication_on valid` property for AVISPA and the `Niagree` and `Nisynch` for Scyther.
- The session key K_{session} is only known by the legitimate users.

The certificates of the different parties and other informations used during the protocol can be requested to the CA. Therefore every actor needs to possess the certificate of the CA prior to any communication. The Listing F.5 presents such communication between the user and the CA to request the certificate of a node, its TPM as well as the expected PCRs values – if the user trusts the CA for correct register values.

7.5.4 The PCRs comparison problem

In order for the user to verify the integrity of the remote machine, he has to know the correct values of the PCRs. Indeed, the comparison of the two sets of values (the expected ones and the values send by the TPM) determines the integrity of the remote resource.

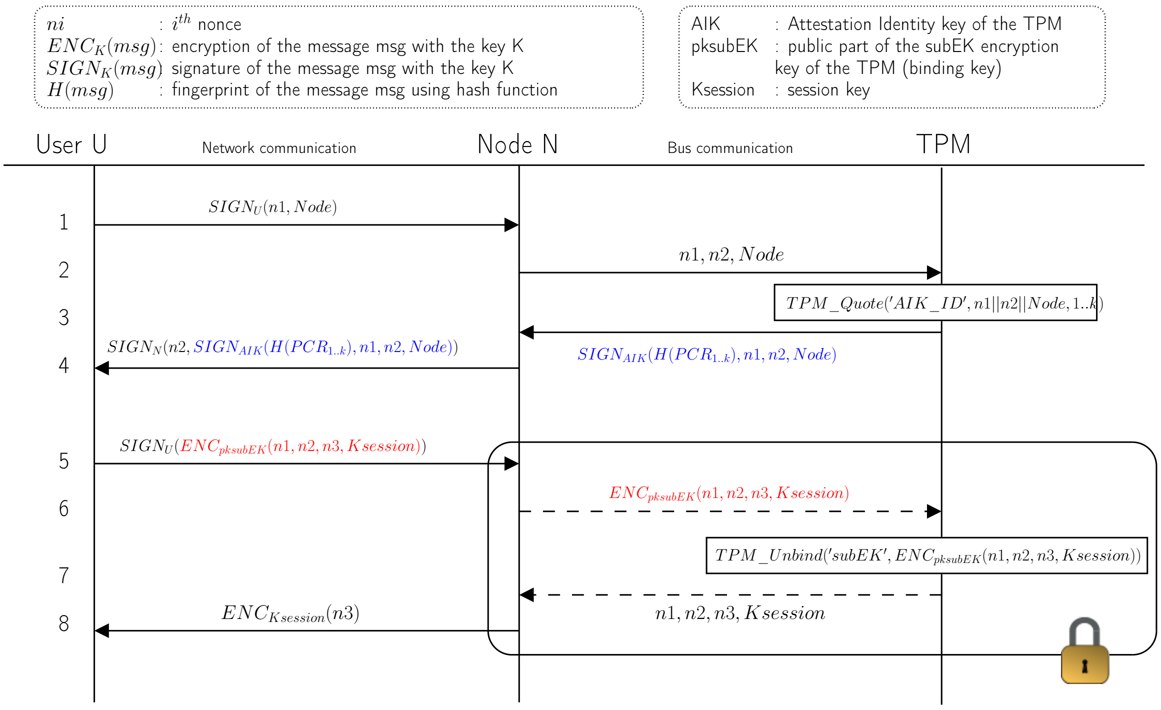

ni : i^{th} nonce
$ENC_K(msg)$: encryption of the message msg with the key K
$SIGN_K(msg)$ signature of the message msg with the key K
$H(msg)$: fingerprint of the message msg using hash function
AIK : Attestation Identity key of the TPM
pksubEK : public part of the subEK encryption key of the TPM (binding key)
Ksession : session key
User U
Network communication
Node N
Bus communication
TPM
1
2
3
4
5
6
7
8
$SIGN_U(n1, Node)$
$n1, n2, Node$
$TPM\_Quote('AIK\_ID', n1||n2||Node, 1..k)$
$SIGN_N(n2, SIGN_{AIK}(H(PCR_{1..k}), n1, n2, Node))$
$SIGN_{AIK}(H(PCR_{1..k}), n1, n2, Node)$
$SIGN_U(ENC_{pksubEK}(n1, n2, n3, Ksession))$
$ENC_{pksubEK}(n1, n2, n3, Ksession)$
$TPM\_Unbind('subEK', ENC_{pksubEK}(n1, n2, n3, Ksession))$
$ENC_{Ksession}(n3)$
$n1, n2, n3, Ksession$

Figure 7.5: Overview of the TPM-based Certification of a Remote Resource (TCRR) protocol.

The user therefore needs to know the correct values:

- The **trusted party** supplies him with the correct value for the BIOS, kernel, OS, hypervisor, middleware etc.

- If open-source software is used by the Cloud provider and the version numbers are available, the user can compute them on his side.

TCRR allows the user to share a secret key used for the VMI encryption but as well to verify the integrity of the distant machine. This verification checks the BIOS, the bootloader, the OS kernel but as well files within the remote machine typically the hypervisor used to execute the VMs or other important files of the host OS (executables and configuration files).

7.6 CertiCloud Framework

The TCRR protocol has been analysed using two of the reference tools AVISPA [1] and Scyther [66] (see Chapter 6). It is important to notice that the Dolev-Yao intruder model [74] has been considered. In this model, the attacker has full control over all the messages that are sent over the network such that he can analyse, intercept, modify or forge any message which is sent to any player. These tool-suites provide a special language for describing security protocols and specifying their intended security properties. The TCRR specifications and its successful validation by both tools is provided in appendix. In particular, the verification proves that the following constraints are met:

1. The protocol ends in finite time without any successful known attack (replay, man-in-the-middle etc.);

2. Only the legitimate actors know the private key $K_{session}$, which remains confidential.

As two reference tools confirm this analysis (using different underlying approaches), it authorizes us to claim that TCRR is secure considering attacks in the Dolev-Yao model.

```
$> ./avispa verification_avispa . hlpsl --ofmc
% OFMC
% Version of 2006/02/13
SUMMARY SAFE
DETAILS BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/benoit/git/tcrr/avispa/ results_verif . if
GOAL      as_specified
BACKEND OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 7.93s
  visitedNodes: 6696 nodes
  depth: 17 plies
```

(a)

```
$> ./avispa verification_vm_avispa . hlpsl --ofmc
% OFMC
% Version of 2006/02/13
SUMMARY SAFE
DETAILS   BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/benoit/git/verifymyVM/avispa/results_verif.if
GOAL         as_specified
BACKEND OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.07s
  visitedNodes: 158 nodes
  depth: 6 plies
```

(b)

Listing 7.1: Protocol Validation of TCRR (a) and `VerifyMyVM` (b) using AVISPA.

```
$> time ./scyther.py  --max-runs=20 --all-attacks
 verification_scyther .spdl
 Verification  results :
claim id [tpmp,u1], Niagree : No attacks.
claim id [tpmp,u2], Secret ksession  : No attacks.
claim id [tpmp,u3], Nisynch : No attacks.
claim id [tpmp,n3], Nisynch : No attacks.
claim id [tpmp,n1], Niagree : No attacks.
claim id [tpmp,n2], Secret ksession  : No attacks.
real          0m6.029s
user          0m5.950s
sys           0m0.060s
```

(a)

```
$ time ./scyther.py verification_vm .spdl
 Verification  results :
claim id [verifvm,u1], Niagree  : No attacks.
claim id [verifvm,u3], Nisynch  : No attacks.
claim id [verifvm,n3], Nisynch  : No attacks.
claim id [verifvm,n1], Niagree  : No attacks.
real     0m0.055s
user     0m0.034s
sys      0m0.019s
```

(b)

Listing 7.2: Protocol Validation of TCRR (a) and `VerifyMyVM` (b) using Scyther.

Based on the TCRR protocol described in Section 7.5, this section presents CertiCloud, a novel approach for the protection of IaaS platforms. As in most (if not all) IaaS cloud services, we assume that the clients would like to deploy their VM on the cloud provider's resources. In CertiCloud, this task is operated by the *Cloud VM Manager* that runs a lightweight Xen hypervisor on a machine equipped (and associated) with a TPM. The same *Cloud VM Manager* plays the part of the Remote Resource in the TCRR protocol. The general overview of the CertiCloud framework is depicted in Figure 7.2, Figure 7.3, Figure 7.4, Figure 7.6 and Figure 7.7. In the sequel, we assume that a user U owns a VM image VMI_U he wishes to deploy securely on the resources offered by the Cloud provider. It is also supposed that U computes locally (at step 0) the reference checksum values $\{h_0^{ref}, \ldots, h_k^{ref}\}$ associated with VMI_U that reflect what he considers as a secure VM configuration. The elements considered for hashing are summarized in Table 7.2; they represent what CertiCloud calls VMI_U certificate.

To deploy securely his systems, the following steps are proposed in the CertiCloud framework:

1. U generates a private key K_U he uses to encrypt VMI_U.

2. The encrypted image is sent onto the cloud storage area using any secure transmission protocol (`scp`, `sftp` etc.) supported by the IaaS platform.

3. Now U wants to deploy (or run) his image. He first checks the integrity of the Cloud VM Manager and exchanges with it the session key K_U. This is conducted using the TCRR protocol described in Section 7.5.

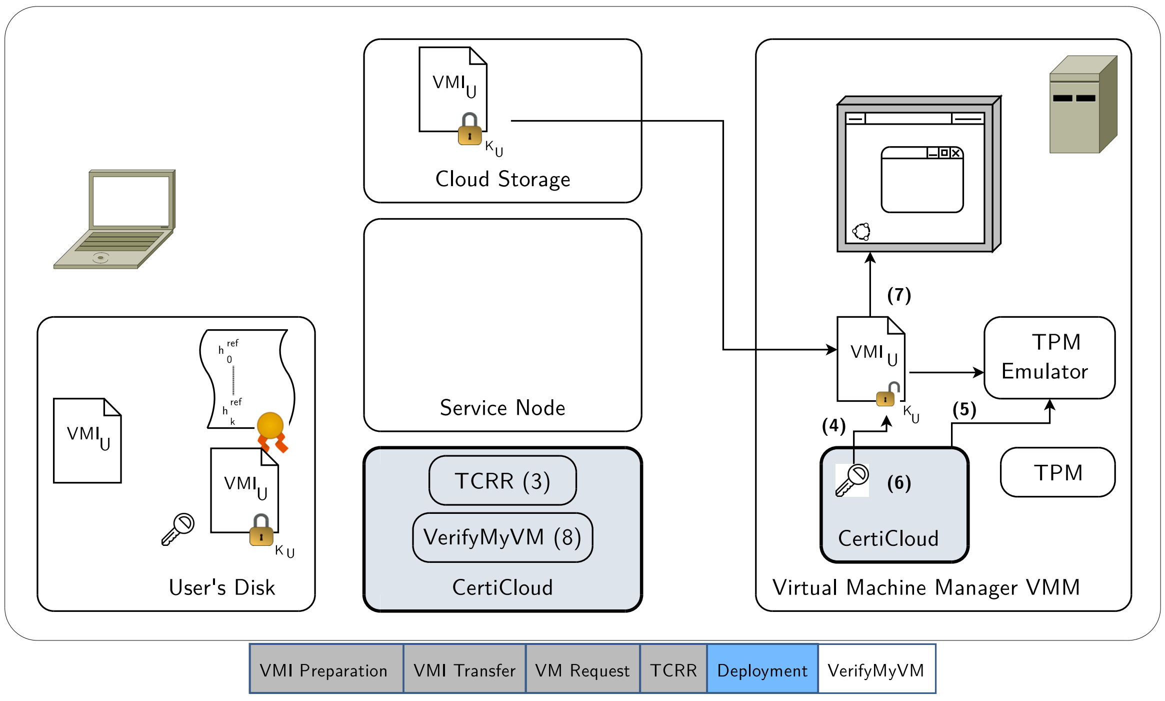


Figure 7.6: Overview of the CERTICLOUD framework (deployment of the VM).

4. Using K_U, the Cloud VM Manager is able to retrieve the encrypted VMI_U and decrypt it in a secured (local) memory area.

5. The Cloud VM Manager now runs an emulated TPM that will be associated upon deployment to VMI_U. Consequently, there will be as many emulated TPMs as VMs to be run.

6. The emulated TPM behaves as a TPM and its PCRs needs to be initialized with the values that reflect the original state (considered as secure) of VMI_U.

7. Now VMI_U can be run.

8. On demand, the user can check the integrity of his running VM using the `VerifyMyVM` protocol (see Section 7.7 below).

In CERTICLOUD, the IaaS provider controls a lightweight Xen or QEMU hypervisor to run users' VMs and each of these VMs is associated to, on deployment, a unique emulated Trusted Platform Module (TPM), referred to as $TPM_{emulator}(VM_U)$ in the sequel. We use a software-based approach instead of a hardware TPM for the following reasons:

1. The number of PCRs of a TPM is limited and they are mostly used for the certification of the Virtual Machine Manager (VMM), *i.e.*, the Remote Resource.

2. Many VMs should be able to run on the node simultaneously (and their state requires the usage of at least one PCR), using as many dedicated soft TPMs as VMs in order to ensure a scalable and rigorous approach.

In practice, CERTICLOUD makes use of the TPM emulator software [146] (see Section 4.2.3). Whereas we could have implemented our own software, designing a new cryptographic library without any flaws takes a lot of time in terms of developments and reviews. Reusing an existing framework is quicker and safer. In case of multiple physical TPMs present on the machine (which is not common

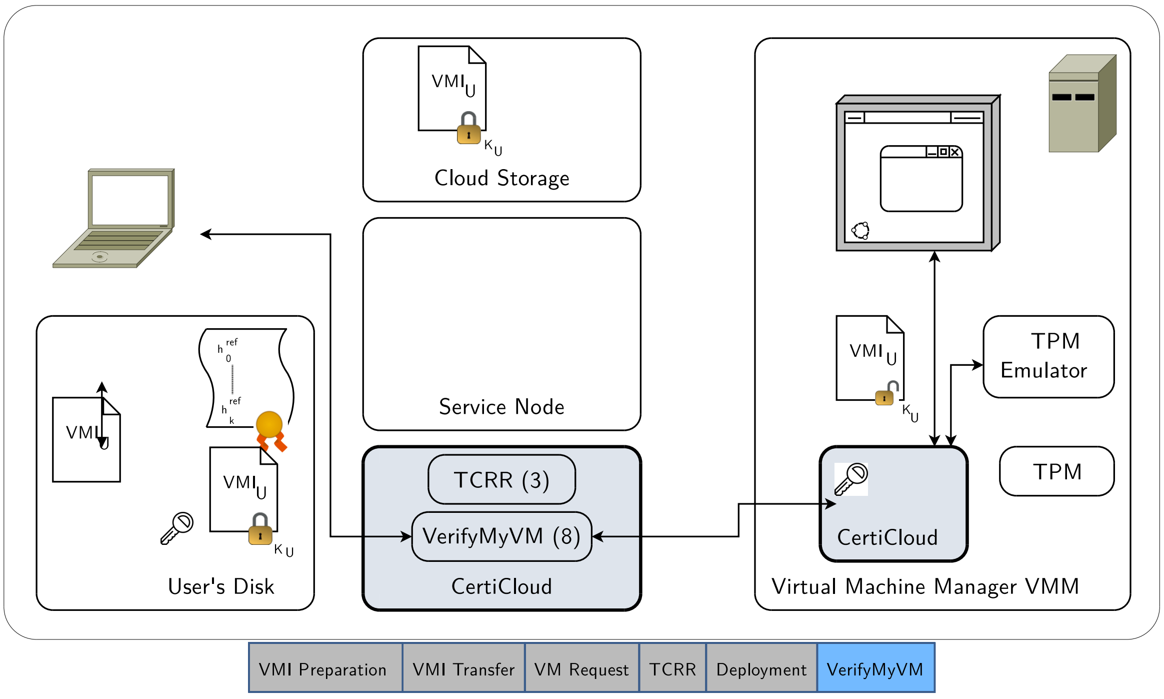


Figure 7.7: Overview of the CertiCloud framework (`VerifyMyVM`).

nowadays) and the possibility of resetting the PCR values of the extra TPMs, the usage of the physical ones instead of the simulated ones is straight forward.

It is also important to notice that the TPM emulator, just as the other elements running on the Cloud resource, is checked by the TCRR protocol. Being software-based, this approach has no physical security (for instance, a shield memory to protect the keys) like a hardware TPM. Yet at this level, not all the functionalities of the TPM are required. The crucial point is that the emulated TPM (or any software library that provides the functions used in CertiCloud to verify a running VM) operates *outside* the scope of the checked VM such that a corrupted VM cannot falsify the result of the integrity measures.

7.7 VerifyMyVM

Once the integrity of the Cloud VM Manager is established via the TCRR protocol and the VM deployed, we propose an novel mechanism – namely the `VerifyMyVM` protocol – that permits the user to check on-demand the integrity status of its running VM in order to detect any tampering attempt on the configuration of the running system. The verification mechanism `VerifyMyVM` used to ensure the integrity of the VMs is based on the same ideas developed in Tripwire [104], concretely saving and checking the hash values of most of the binaries and configuration files of the operating system to detect any modification, deletion, replacement or addition performed in the system. Of course, these checksums are not computed by the VM itself (otherwise they could be manipulated by an attacker having control of the VM) but rather by the Cloud VM Manager node that runs the virtualization OS and $TPM_{emulator}(VM_U)$. In practice, the hashing operations are performed by mounting the VM image disk in read-only mode (to avoid any unwanted writing inside the VM). This can be done during the execution of the VM because there is no need to stop or pause the VM to compute the checksums.

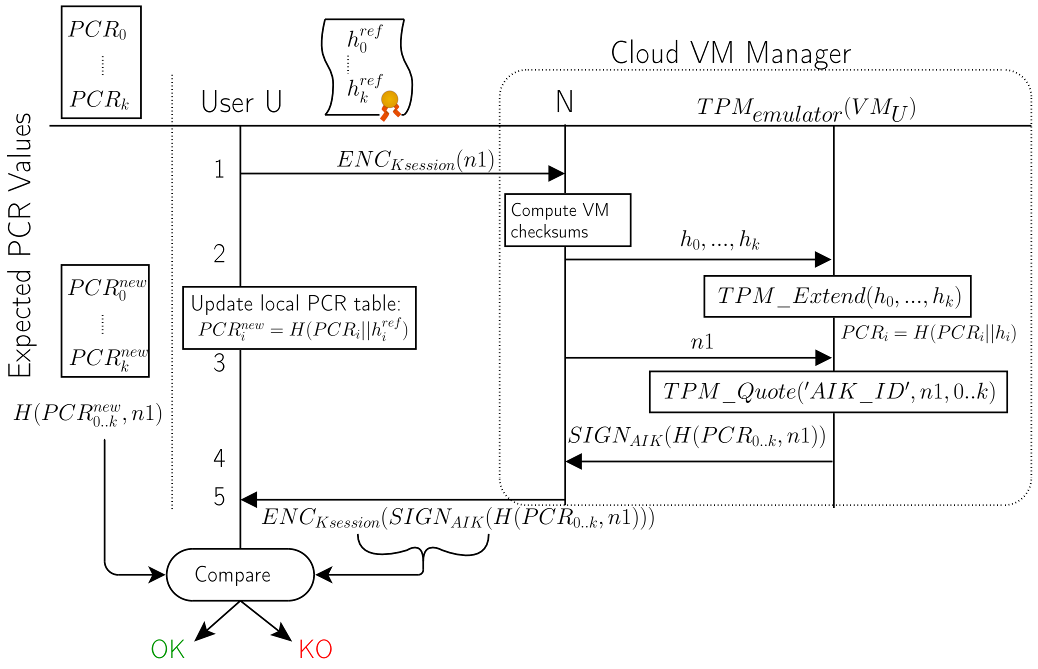


Figure 7.8: Overview of the `VerifyMyVM` protocol.

7.7.1 The `VerifyMyVM` Hypothesis

The `VerifyMyVM` protocol is detailed in Figure 7.8. In the CertiCloud proposal presented in Figure 7.7, it is assumed that this protocol is launched on demand after step seven that permits to assume the following hypothesis:

- the Cloud VM Manager is secure and the private session key K_U has been safely exchanged in step (3). In particular, the communication channel between the machine N running the virtualization OS and $TPM_{emulator}(VM_U)$ is assumed to be secure;
- VMI_U has been securely decrypted and deployed by the cloud VM manager (steps 4 to 7);
- the reference checksums $\{h_0^{ref}, \dots, h_k^{ref}\}$ associated with VMI_U have been computed (by the user at step 0 and by the cloud provider at step 6). For the moment, CertiCloud considers only the checksum of the elements listed in table 7.2 (in particular, $k = 8$). They are used to initialize the PCRs of the $TPM_{emulator}(VM_U)$ prior to the deployment via the built-in `TPM_Extend` function such that at the end of step 6, $PCR_i = H(0||h_i^{ref}) = H(h_i^{ref})$. On his side, U computes locally these values for further comparisons.

7.7.2 The `VerifyMyVM` Protocol

The `VerifyMyVM` protocol then operates as follows:

1. U requests the Cloud VM Manager to check his running VM_U: he sends a nonce n_1 encrypted with the session key $K_{session}$. Upon reception, N mounts VMI_U in read-only mode and the actual VM checksums $\{h_0, \dots, h_k\}$ are computed.
2. The checksums are used by $TPM_{emulator}(VM_U)$ to extend its PCRs values by running `TPM_Extend(`h_i`,i)`. In parallel, U updates his local table storing the expected PCR values using the VM_U certificate as follows:

$$PCR_i = H(PCR_i||h_i^{ref})$$

3. N submits to $TPM_{emulator}(VM_U)$ the nonce n_1.
4. $TPM_{emulator}(VM_U)$ issues the `TPM_quote` command in order to report to N over the challenge $n1$ the values of $PCR_0, \dots, PCR_k$, signed with its AIK.
5. N returns to U the cryptographic report of the PCRs values stored in $TPM_{emulator}(VM_U)$, yet encrypted with the session key $K_{session}$.

Name	Description	Hashed elements
h_0^{ref}	VM kernel	`vmlinuz-2.6.26-2-xen-686`
h_1^{ref}	Init. ramdisk	`initrd.img-2.6.26- 2-xen-686`
h_2^{ref}... h_8^{ref}	VM main system folders	`/bin /etc /sbin /lib /usr/bin /usr/lib /usr/sbin`

Table 7.2: VM Checksums considered in CertiCloud.

Using $K_{session}$, U can decrypt the last message, check AIK signature and the validity of the 'quote' command over the challenge $n1$. U can then compare the actual PCRs values reported by $TPM_{emulator}(VM_U)$ with the expected values he previously computed.

In case of the evolution of the VM image (for example, update of the system), the user can either ask the VMM for the new h_i^{ref}, or compute them himself.

7.7.3 `VerifyMyVM` Protocol Validation

As TCRR, the `VerifyMyVM` protocol has been extensively analysed by the two reference tools AVISPA and Scyther (see §6.1), to prove that the protocol ends in finite time without any successful known attack (mainly replay or man-in-the-middle). The listings used for the successful validation of `VerifyMyVM` on each of these tool-suites are provided in Appendix F and the Listing 7.1 and Listing 7.2 are showing the output of the validation tools. Again, the fact that neither attack nor flaws are revealed by this analysis does not mean that none exists against the `VerifyMyVM` protocol. This step authorizes to claim that the protocols are secure according to the model.

7.8 Implementation

The implementation of the CertiCloud framework has been designed with the interoperability in mind. Indeed, the dependence on the Nimbus [126] Cloud platform (see Section 2.3.4) is very small, and appears only in one single point through a *hook* which is platform dependent.

This approach gives a flexibility in the integration of CertiCloud into different CC platforms. In this sense a standard API called “CertiCloud API” has been defined for the communication within CertiCloud. This flexible integration lets unchanged the basic blocks composing the framework that will be presented in the following section.

7.8.1 Cloud-Middleware Independent Modules

- `server_tcrr`: a server for the TCRR protocol relying on the unique workspace number obtained from the Nimbus workspace used to link the VM to the user. This server only acts as a proxy allowing the user to communicate with the machine hosting the TPM that will be the host for his VM in case of successful communication, *i.e.* once the machine has been verified. This server is not critical for the security of the TCRR protocol as it is only acting as a proxy forwarding messages from one party to another. Indeed in the Dolev-Yao model the wire is already considered as insecure. A jeopardized proxy server would however be an issue for the availability of the service as a compromised wire.

- `server_verifymyvm`: a server for the protocol `VerifyMyVM`, working on the same principle than the TCRR server, the same machine can even be used for the two servers.
- `server_ca`: a server taking care of the certificates of the different entities playing a role in the CertiCloud framework. As this server generates and stores the certificates of the different entities it must not be compromised otherwise a malicious attacker could generate certificates for untrusted resources *e.g.* TPMs outside the control of the CC providers or even fake TPMs, allowing the attacker to decrypt the VMI of the user.
- `user_tcrr`: the command-line user interface of the TCRR protocol allowing the user to verify the node integrity remotely and in case of a successful verification to send the symmetric key used to encrypt the VM.
- `user_verifymyvm`: the command-line user interface of the `VerifyMyVM` protocol allowing the user to launch verification of the files contained within the VMI during the execution of his VM.
- `user_compute_hashes`: the command-line tool allowing the user to compute the reference values of his VMI.
- `node_tcrr`: the interface for the nodes of the Cloud to use the TCRR protocol.
- `node_verifymyvm`: the interface for the nodes of the Cloud to use the `VerifyMyVM` protocol and to manage the virtual TPMs of the node.
- `nimbus_decrypt`: script launched by Nimbus' hook, allowing the Nimbus framework to use CertiCloud. It launches the `node_tcrr` script to perform the TCRR protocol, to retrieve the symmetric key of the encrypted VMI and to decrypt it.

7.8.2 Existing/Extended Nimbus Component

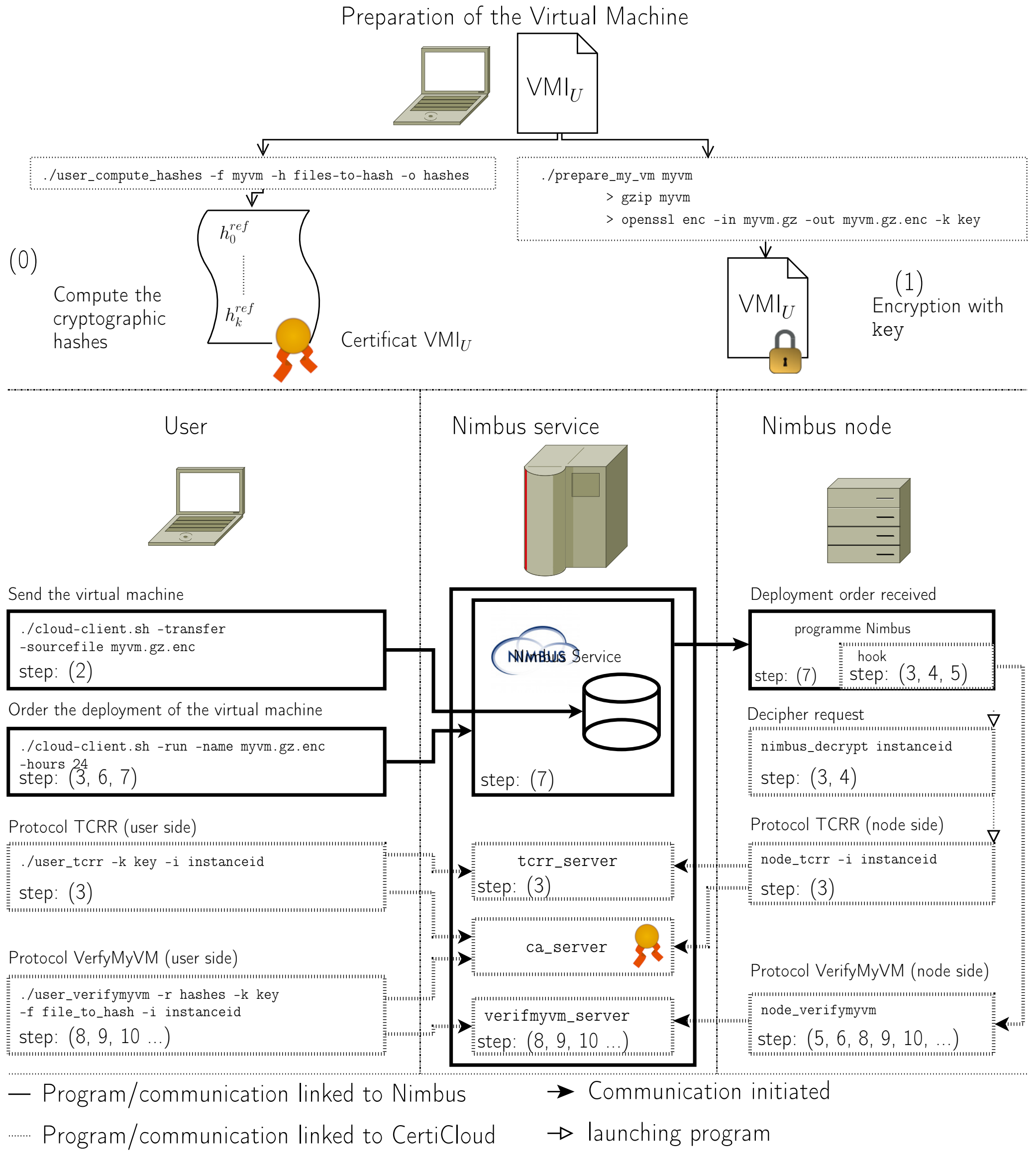
- `cloud-client.sh` given by Nimbus, allows to send requests for VM and to manage users VMI *e.g.* to transfer VMI from user to server, to launch, save or terminate VMs.
- `Nimbus` ***hook*** which integrates the CertiCloud's API to Nimbus. This part is linked to Nimbus and should be adapted specifically to any other Cloud Platform.

7.8.3 Details of the CertiCloud Implementation in Nimbus

The different steps composing CertiCloud showed in Section 7.6 and the different modules developed for the Nimbus implementation are explained in the following sections.

Preparation of the VMI

As explained in Figure 7.9, the user must have generated his VMI containing the software and libraries that he needs. Then he has to perform the **step(0)** which is the computation of the reference checksums for his VM. He uses the `user_compute_hashes` program to compute the hash values of the files and folders given in the argument. These hash values will be used as reference values to verify on the fly that its VMI has not been modified. The user can afterwards compress its VMI for faster transfer and then proceed to the **step(1)** which is the encryption of his VMI using respectively `gzip` and `openssl` [159].



```
./user_compute_hashes -f myvm -h files-to-hash -o hashes
```

```
./prepare_my_vm myvm
        > gzip myvm
        > openssl enc -in myvm.gz -out myvm.gz.enc -k key
```

```
./cloud-client.sh -transfer
-sourcefile myvm.gz.enc
```

```
./cloud-client.sh -run -name myvm.gz.enc
-hours 24
```

```
./user_tcrr -k key -i instanceid
```

```
./user_verifymyvm -r hashes -k key
-f file_to_hash -i instanceid
```

```
nimbus_decrypt instanceid
```

```
node_tcrr -i instanceid
```

```
node_verifymyvm
```

Figure 7.9: Overview of the implementation of CertiCloud in Nimbus.

Transfer of the VMI and deployment order

For the **step(2)**, the transfer of the VMI on the CC platform, the Nimbus script `cloud-client.sh` is used. The same script is used as well for the deployment command to request the launch of a VM on an available physical machine.

Execution of the TCRR protocol

The changes applied to Nimbus to integrate CertiCloud are negligible, they consist of the modification of a `Python` script `ImageEditing.py` on every Nimbus node. This file is responsible for the edition of the VMI on the node, for instance it detects if the images are compressed depending on their names, *i.e.* if the file name ends with `.gz`, it performs the decompression of the images. This `Python` script has been modified to take into account the `.enc` extension, assumed to correspond to an encrypted VMI. In this case, the `nimbus_decrypt` script is invoked to initialized the TCRR protocol on nodes' side using `node_tcrr`.

The user launches at the same time `user_tcrr`. These two modules communicate through the `server_tcrr` allowing the **step(3)** verifying the integrity of the node and sending the symmetric key used to encrypt the VMI. The **step(4)** consists of the actual decryption of the VMI by the Nimbus node using `nimbus_decrypt`.

Execution on-demand of the `VerifyMyVM` protocol

At the conclusion of the successful previous decryption and decompression operations, the modified `ImageEditing.py` script sends a “start” signal to the `node_verifymyvm`, which triggers the **step(5)** assigning an available virtual TPM to the VM. The **step(6)** is the initialisation of the PCRs values using the first round of the `VerifyMyVM` protocol. The source code of the virtual TPMs has been modified to allow a parallel execution of multiple virtual TPMs on the same node by changing their ids, therefore it is possible to have many `/dev/tmpX`.

Then the **step(7)** can be started and the VM can be deployed. In parallel to the machine start, the `node_verifymyvm` client running on the node connects to the `server_verifymyvm` to be ready for any requests from the user to verify his VM using the `VerifyMyVM` protocol.

The user can afterwards launch **step(8)** which is the on-demand verification of the integrity of his VM as many times as he wants using the `user_verifymyvm` script which asks to the virtual TPM assigned to his VM to verify the integrity of the files contained in the VMI. This can be done while the VM is running.

7.8.4 Impact of the Implementation of CertiCloud in Nimbus

The Figure 7.9 allows to have a clear view of the modifications and the additions done in Nimbus to implement CertiCloud. The Cloud Computing (CC) service provider has only to install on his nodes the scripts `nimbus_decrypt`, `node_tcrr`, `node_verifymyvm`, their dependences, the configuration files as well as the certificates of the CA and the patch Nimbus to integrate the hook.

The CC service provider has to configure as well three new servers `server_ca`, `server_tcrr` and the `server_verifymyvm` allowing distribution of the certificates as well as communication for the TCRR and `VerifyMyVM` protocols between the user and the node.

The user has to crypt his VMI and compute the reference hash values using `user_compute_hashes`, and he must download the scripts `user_tcrr`, `user_verifymyvm`. He must as well own a valid certificate signed by the CA of the CC service provider. Having CertiCloud on the Nimbus platform does not force the user to use it, *i.e.* the modifications of Nimbus do not require an encrypted VMI and in case of a normal VMI the modified Nimbus just will not start the CertiCloud services.

7.9 Experiments & Validation in a Typical IaaS Scenario

To validate our approach, a prototype of CertiCloud has been developed using the Xen virtualization hypervisor [22] and the QEMU/KVM hypervior on the following machines:

- Processor: Intel core i7 870 (2.93GHz), 16 GB DDR RAM;
- OS: Ubuntu 12.04, kernel 2.6.35-30-server;
- Virtualisation: qemu-kvm-0.12.5;
- ETHZ TPM emulator (version 1.2.0.7 of the specifications);
- STMicroelectronics TPM (version 1.2.8.8 of the specifications);
- TPM/J Java application, API 0.3.0 (alpha) [141].

As for the user VMs, the classical Linux environments deployed on IaaS platforms are of two kinds: either a dedicated server running a minimal set of services or a desktop environment closer to Laptop configuration. With this in mind, we based our experiments on two kinds of VMs: a small one (336 MB) labeled `Debian` that runs a Debian Lenny distribution featuring a classical LAMP (Linux/Apache/MySQL/PHP) server. The second environment is of bigger size (2.3 Go) and referred to as `Ubuntu` as it runs an Ubuntu hardy distribution. As CertiCloud has been designed in the stepwise approach, we evaluated independently each involved step, focusing on the most important ones.

The common criteria chosen for performance evaluation is the execution time on the above platform. To reach a statistical significance of the presented results, at least 100 runs of each tests have been conducted, knowing that a single VM (the one analyzed) is run on the VMM to limit external system perturbations. We have conducted similar experiments on a more realistic test case for IaaS platforms where each physical resource runs at least 10 VMs. The results did not significantly differ from the ones presented in this section. The Figure 7.10 is showing the time for the different steps of CertiCloud measure on the Debian VM.

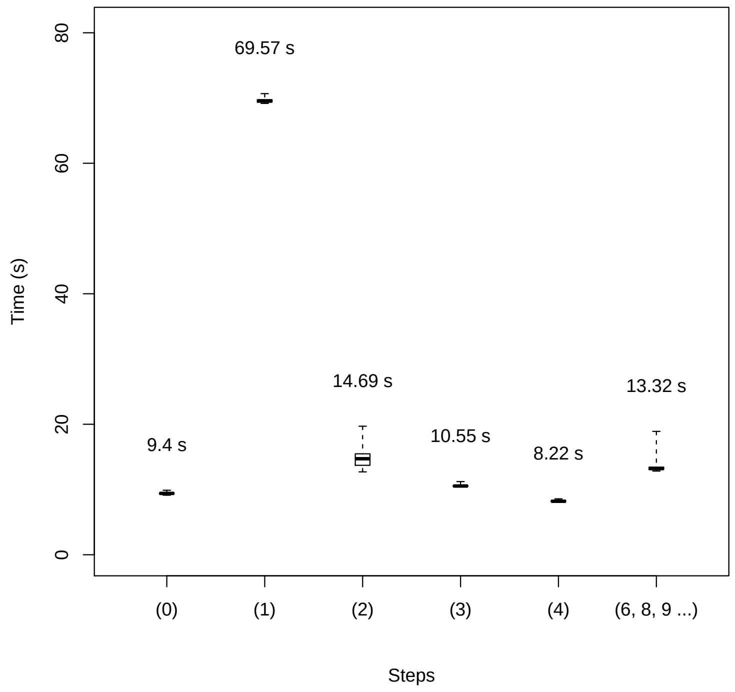


Figure 7.10: Execution time for the different steps of CertiCloud defined in Section 7.6 (measured on the Debian VM).

First of all, the performances of the TCRR protocol (step 3 in the CertiCloud process) have been evaluated as illustrated in figure 7.11.

On overage, less than 10s are required to complete the protocol, *i.e.*, to certify the Cloud resource and exchange the private key. We also detail the contributions of the two main operations conducted by the TPM in TCRR (namely `TPM_Quote` and `TPM_Unbind` – see Section 7.5). Then, at step 4 of the CertiCloud process, a decryption operation takes place on the VM Manager. The performance of this step (assuming an encryption scheme based on AES-256) are summarized in table 7.3: it is of course a very costly yet unavoidable operation to preserve the confidentiality of user's environment. Note that it happens only before the deployment and does not impact the performance of the VM of

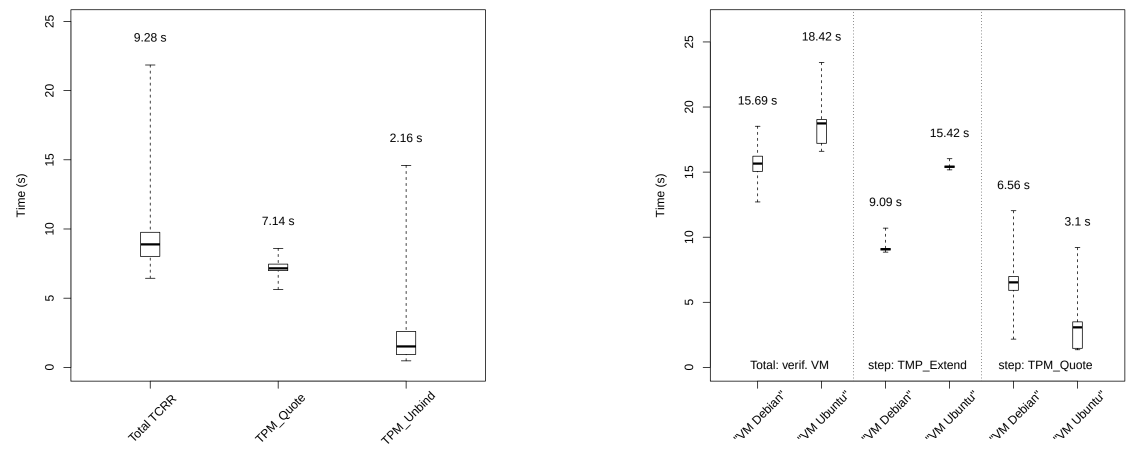


Figure 7.11: Performance evaluation of the TCRR and the `VerifyMyVM` protocols in CertiCloud.

the user.

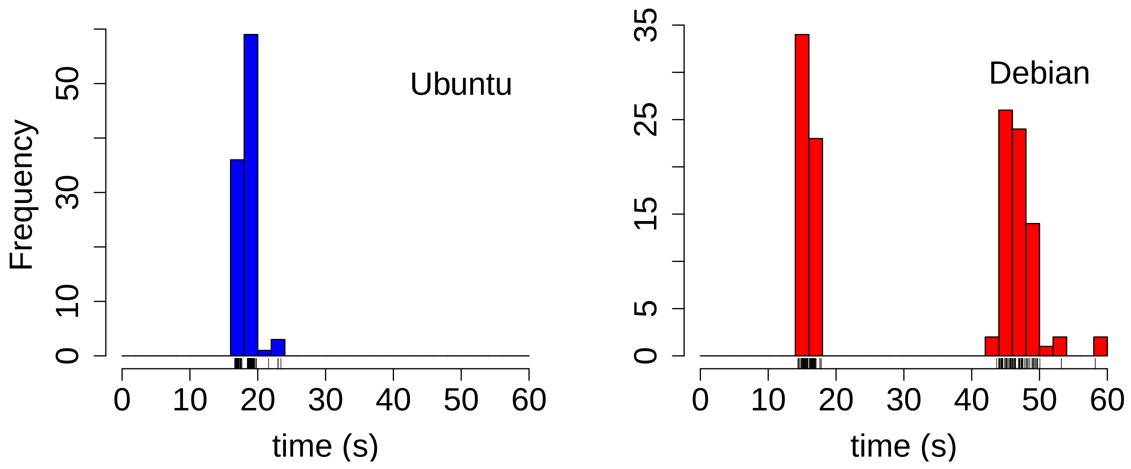


Figure 7.12: Detection times after unauthorized modifications of the system in CertiCloud.

We then focus on the evaluation of the `VerifyMyVM` protocol. As for the measure of the TCRR performances, the total execution time of the protocol together with the contributions of the two main operations conducted by the emulated TPM $TPM_{emulator}(VM_U)$ in this protocol (`TPM_Extend` and `TPM_Quote` – see Section 4.2). The experimental results are depicted in figure 7.11. Less than 19s are required to certify the integrity of a running VM which highlights a relatively low overhead induced by this protocol.

Finally, it appeared important for us to evaluate the reactivity of CertiCloud against active corruptions (rootkit installation etc.). In order to measure this reaction time a modification of a binary in the running VM was performed at the time $t_0 = 0s$. In practice, a rootkit installation has been simulated by pushing on the system altered `ls` and `top` binaries using a simple `scp` command as root. Then the `VerifyMyVM` protocol was initiated and we computed the time to reach a final state.

First of all, `VerifyMyVM` always finished in the `KO` state. Then, table 7.3 and figure 7.12 present the detection time for the two kinds of VM: between 14 and 58s are required to conclude. Something quite interesting and counter-intuitive appears here as the Ubuntu VM, which is bigger, seems to be faster than the Debian VM. This can be explained by the two different disk synchronization policies;

VM type:	Debian	Ubuntu
Size (zipped/total)	528MB/2.1GB	1.4GB/21GB
decryption time	23.324s	1m5.296s
min detection time	14.36989s	16.60140s
average detection time	33.1792s	18.63032s
max detection time	58.21868s	38.52942s

Table 7.3: Average decryption time of the VM images and detection time after unauthorized modification in CertiCloud.

when a modification is done in the Ubuntu VM, it writes with almost no delay the files on the disk, whereas in the Debian system, the modifications are not written directly on the disk because of a caching mechanism which permits to avoid the continuous usage of the disks, therefore they cannot be detected and need the expiration of a certain delay in order to be written.

To conclude, the experiments conducted in this section highlight the relatively low overhead induced by the CertiCloud protocols ($\sim$ 10s for TCRR, less than 20s for `VerifyMyVM`). Reactivity measures show a 100% detection rate in less than 60s (less than 20s with the appropriate I/O buffer configuration) which is quite reasonable for a dynamic checking in the CC context.

7.10 Conclusion

Many open security issues should be solved to transform the current euphoria on Cloud Computing into a wide acceptance. The CertiCloud framework developped in this section tackles integrity and (to a minor measure) confidentiality aspects of the Infrastructure-as-a-Service (IaaS) cloud paradigm.

Until recently, only pure software approaches were proposed to check the integrity of a remote system leading to the fact that it was not possible to fully trust a remote platform. For this reason, it is impossible nowadays for a user having strong security expectations to really trust Cloud providers. Yet with the advent of the Trusted Platform Module (TPM) specifications by the Trusted Computing Group (TCG), together with their integration at low cost in nearly all recent motherboards, it is now possible to imagine novel ways to treat this problem. Indeed, TPM offers the access to a strong cryptographic processor, embedded with a shielded memory and ways to execute cryptographic primitives to authenticate itself and report the state of its associated system status (BIOS, MBR, kernel, OS etc).

Exploiting the TPM capabilities (therefore relying on hardware elements), the TPM-based Certification of a Remote Resource (TCRR) protocol has been presented. It permits to assert the integrity of a remote resource and exchange a user-defined private symmetric key. The extensive analysis of TCRR by AVISPA and Scyther, two industrial-reference tools for the automatic verification of security protocols has been carried out. Both toolsuites assert the security of TCRR which proves the validity of the design.

Then CertiCloud, a novel approach for the protection of IaaS platforms that implements the TCRR protocol has been introduced. CertiCloud proposes a stepwise approach to secure a user's VM, from the setup on user's side to the secure storage on the Cloud and to the deployment on cloud resources. These resources are assumed to host a (physical) TPM and a virtualization framework based on the Xen or QEMU hypervisor. This permits to run users' VMs, each of them being associated upon deployment to an emulated TPM. All these elements (the physical resource, the virtualization framework and the emulated TPMs) are certified using the TCRR protocol.

Then, a novel verification mechanism called `VerifyMyVM` has been designed to allow a user of the platform to check his running VM and to detect any system corruption.

As TCRR, the validation of the `VerifyMyVM` protocol by AVISPA and Scyther justifies the design choices and guarantees the security of the framework from the communication point of view. The performed experiments prove the feasibility of the approach on commodity hardware and demonstrate the low overhead induced by the CertiCloud protocols.

As the TPM measurements are vulnerable to the TOCTOU attacks and as the CertiCloud framework is based on this measurement, the framework is as well vulnerable to these attacks. Indeed if an attacker managed to hide himself during the measurement by restoring initial files, he won't be detected. Using passive hardware, it is impossible to counter TOCTOU attacks. Indeed a corrupted system can always provide the valid measurement to the passive hardware. More generally, solutions which are checking the integrity of a system periodically, such as tripwire [104], are vulnerable to these attacks.

The perspectives of this work are numerous: deeper analysis of CertiCloud scalability, development of a migration protocol and integration into other existing cloud middleware such as Eucalyptus [77], OpenNebula [10] or OpenStack [11].

Part III

Code Protection using Obfuscation Techniques (JShadObf)

Chapter 8
The Obfuscation Paradigm: an Overview

Contents

As seen in Section 5.1 software protection can intervene at different levels: at the OS level, at the middleware level or the application one. The first part of this dissertation elaborated a way to increase the security at an OS and middleware level, by checking the physical node as well as the VMI. In this part, we propose a novel source-to-source obfuscation process able to increase the level of protection at an **application level**. The objective is to obfuscate the source code of a program to conceal its purpose, its logic and maybe some secrets without altering its functionality, thus preventing the tampering or the reverse engineering of the program.

This chapter reviews the key concepts and notions associated with code obfuscation.

8.1 Code Obfuscation: Definitions

We now provide the preliminary definitions associated to an obfuscating process, most of them being defined in the work of Collberg [62].

Definition 11 (Obfuscating Transformation)
Let $P \xrightarrow{\mathcal{T}} P'$ be a transformation of a source program P into a target P'. $P \xrightarrow{\mathcal{T}} P'$ is an obfuscation transformation *if P and P' have the same* observable behaviour. *More precisely, if P fails to terminate or terminates with an error condition, then P' may or may not terminate, otherwise P' must terminate and produce the same output as P.*

Observable behaviour can be defined as being the behaviour experienced by the user. For example, if the obfuscated program P' has side effects such as file creation or network communications that are not initially present in P and which are not noticed by the user, it can still be considered as having the same observable behaviour from a user point of view.

The quality of obfuscated transformations is the composition of three properties:

- Potency which is a measure of transformation usefulness in its task of hiding the intent of the programmer. Potency can be seen as a measure of an obfuscation transformation efficiency towards human readers.
- Resilience which measures the efficiency an obfuscation transformation against automatic deobfuscators (as an opposition to potency).
- Cost which measures the penalty introduced by the transformation: a transformation can make the program use more memory or more time.

The next subsections detail each of these key concepts.

8.1.1 Transformation Potency

Definition 12 (Transformation Potency) *Let τ be a behaviour-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P. $\tau_{pot}(P)$, the* potency *of τ with respect to a program P is a measure of the extent to which τ changes the complexity of P. It is defined as:*

$$\tau_{pot}(P) = \frac{E(P')}{E(P)} - 1$$

Therefore, τ is a potent obfuscating transformation *if* $\tau_{pot}(P) > 0$.

Many metrics exist in computer science and some are presented in Section 8.2 and they can be used or combined to measure the complexity E used in this definition.

Software complexity metrics are linked to the programmer's ability to understand a source code, they are therefore subjective. The potency can be pictured as a measure of a transformation usefulness toward human readers. Some transformations will increase a program complexity according to the selected metrics and will indeed increase the difficulty for a human reader to understand the code but it does not take into account that it might be easy for a machine to deobfuscate these transformations.

8.1.2 Transformation Resilience

To measure a transformation usefulness against automatic deobfuscators, resilience has to be introduced. Resilience takes two parameters into consideration :

- *Programmer Effort* (the amount of time taken to build an automatic deobfuscator that will efficiently reduce the potency of τ).
- *Deobfuscator Effort* (the execution time and the memory space required by the obfuscator to reduce efficiently the potency of τ).

Definition 13 (Transformation Resilience) *Let τ be a behaviour-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{res}(P)$ is the* resilience *of τ with respect to a program P.*

$\tau_{res}(P) =$ one-way *if information is removed from P such that P cannot be reconstructed from P'. Otherwise:*

$$\tau_{res} = Resilience(\tau_{Deobfuscator_{effort}}, \tau_{Programmer_{effort}})$$

Where Resilience is the function defined by the matrix defined in the matrix in Figure 8.1.

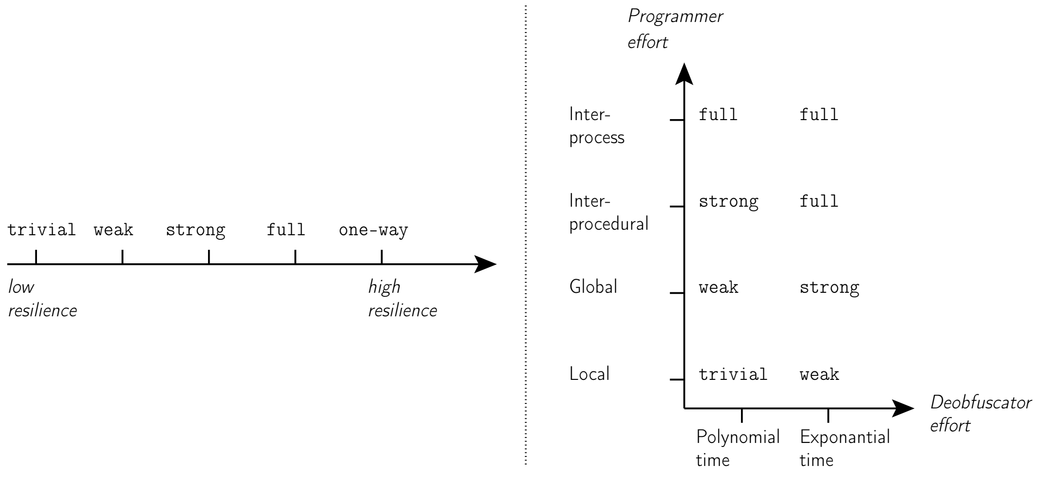


Figure 8.1: Resilience of an obfuscating transformation: Scale of values (left) and resilience matrix (right).

8.1.3 Transformation Cost

Transformations often introduce some loss of efficiency in the program. For example, the program might need more memory space or more time to perform the computations after a transformation. Transformation cost introduces this notion.

Definition 14 (Transformation Cost) *Let τ be a behaviour-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{cost}(P)$ is the extra execution time/space of P' compared to P. $\tau_{cost}(P)$ is said to be:*

dear *if executing P' requires* exponentially *more resources than P*

costly *if executing P' requires $O(n^p), p > 1$, more resources than P*

cheap *if executing P' requires $O(n)$ more resources than P*

free *if executing P' requires $O(1)$ more resources than P*

8.1.4 Transformation quality

The three evaluations of transformations defined previously (potency, resilience and cost) can be used to compose the quality metric of obfuscating transformations.

Definition 15 (Transformation quality) *$\tau_{qual}(P)$, the quality of a transformation τ, is defined as the combination of the potency, resilience, and cost of τ:*

$$\tau_{qual}(P) = (\tau_{pot}(P), \tau_{res}(P), \tau_{cost}(P))$$

These transformation evaluations require the definition of metrics to be computed. The next section will present some existing metrics coming from the software engineering area.

8.2 Obfuscation Metrics

Computing the complexity of a program is not an easy task as the complexity is a subjective measurement depending on what is considered as complex by a reverse engineer or by a programmer. It is therefore linked to the programmer's knowledge and competence. It is as well dependent on the programming language used as certain structures of program might be language dependent, for example, a metric, such as [61], which counts the number of methods within an object or the length from the class to the root class it inherits from, can only be applied to Object Oriented (OO) programming languages.

The following metrics are well-known and have been referenced by Collberg et al. in [63] and in [62] and used in the JShadObf obfuscation software (see Section 9):

μ_1 Program Length [87]. The more operators and operands P has, the more complex it gets.

μ_2 Cyclomatic Complexity [115]. The complexity of a function is measured by the number of predicates it contains.

μ_3 Nesting Complexity [88]. The more conditionals of a function are nested, the more complex that function is.

μ_4 Data Flow Complexity [130]. The complexity of a function increases with the number of variables references in inter-basic blocks.

μ_5 Fan-in/out Complexity [90]. A function is more complex if it has more formal parameters, its complexity also increases with number of global data structures it reads or writes.

μ_e **or** μ_6 represent the efficiency of the code measured as the average execution time of the code on a reference machine, using one or more test cases representing normal execution of the program.

These metrics can be used to evaluate the pertinence of transformations on the complexity of a program. The transformations are described in the next section.

8.3 Transformations

The following obfuscation transformations have been classified by quality by Collberg in his paper [62]. Obfuscating transformations affect different aspects of a program structure and can be classified into three main categories:

1. *Data obfuscation*: composed of all the transformations that modify and obscure the data structures used in a program.

2. *Layout obfuscation*: the transformations changing the information included in the code formatting, *e.g.*, scramble identifier names or remove code indentation.

3. *Control obfuscation* which affects the aggregation, ordering or computations performed within the program control-flow.

These three categories are detailed in the next sections.

8.3.1 Data Obfuscation

Transformations performing a data obfuscation are making the data structures used more obscure. For example, splitting a vector in two vectors is a data obfuscation technique. Data obfuscation transformations can be divided into three subcategories: transformations affecting the storage and encoding, the ordering and the aggregation of the data.

Data structures are often depending on the data the programmer wants to store, for example, storing all the values of the program using the `little endian` instead of `big endian` on a machine reading data using the `big endian` convention is a data obfuscation as the deobfuscator or the programmer would not expect data to be encoded in `little endian`.

Encoding exchange

An example of a transformation modifying the encoding would be the use of an affine function to "encode" and "decode" without provoking an integer overflow. For instance, if we want to transform the variable k in P, we can use constants and use $c_1 * k + c_2$ instead of k in P' (as in 8.1). This however can be easily analysed by a deobfuscator or even an optimiser and be removed automatically but that would increase the time of execution of the program.

P

```
int k;
for (k=1;k<100;k++)
{
        ... vect[k] ...
}
```

$P'(c_1 = 5c_2 = 2)$

```
int k;
for (k=7;k<502;k++)
{
        ... vect[(k-2)/5] ...
        k+=4;
}
```

Listing 8.1: Example of an encoding transformation.

Promoting variables

Promoting a variable consists of storing a variable in an object, for example, an integer typed variable can be replaced by an Integer class. The variable promotion could also be a lifetime increasing, by for example changing the scope of a variable from local to global like in Listing 8.2. Such a transformation increases the number of global variables used by the program functions.

P

```
void foo() {
        int i;
        ... i ...
}

void bar() {
        int k;
        ... k ...
}
```

P'

```
int c;
void foo() {
        ... c ...
}

void bar() {
        ... c ...
}
```

Listing 8.2: Example of a variable promotion transformation.

Splitting variables

Splitting a variable i consist of storing the information contained into i within a set of variables $(i_1, ...i_k)$ like in Listing 8.3. Three pieces of information have to be given : a function $f(i_1, ..., i_k)$ that maps the $i_1, ..., i_k$ to i, a function $g(i)$ that maps i to the corresponding $i_1, ..., i_k$ and operations on $i_1, ..., i_k$ corresponding to the operations available on i.

The potency and cost of such transformations increases with k.

P

```
int t = 12 ;
... t ... ;
```

P'

```
int foo ( v1 , v2)
{
  return v1 ^ v2 ;
}
int u = 456, v = 452;
... foo(u,v) ... ;
```

Listing 8.3: Example of a spliting variable transformation.

Converting static data to procedural data

This transformation replaces some static data by function calls returning the same data like in Listing 8.4. It is preferable not to store all the static data into one function but into many of them. It is as well possible to apply them recursively, *i.e.* the data stored into the function used to store the static data can be relocated into another function.

P

```
...
... "abc" ...
...
... "dfe" ...
```

P'

```
string foo(int a)
{
        if (a == 0)
                return "abc"
        elif (a == 1)
                return "dfe"
}
...
...     foo(0) ...
...
... foo(1) ...
```

Listing 8.4: From static data to function call.

Aggregation Transformations

This transformation can be seen as the opposite of the *splitting variables* transformation. It aggregates data from multiple variables into one using, for instance, a vector or a class. The idea is to decorrelate the variable from its semantic meaning.

The restructuring of arrays is an example of aggregation and splitting transformation: merging several arrays in one, splitting an array into several arrays, folding an array (increasing its dimension) or flattening an array (decreasing its dimension) allow to disconnect the data from the meaning of the variables.

These transformations often have low potency because complexity metrics cannot measure the fact that some of these transformations introduce new structures. For example, a programmer manipulating an image would declare a 2 dimension array. Manipulating a one dimension array or a 3 or more dimension array would significantly increase the obscurity of the program.

Ordering transformations

Modifying the order of method calls and operations in the code allows to create big code blocks with less semantic meaning for the block itself. It can design as well the reordering of data within an array like in Listing 8.5.

P

```
int A = [1,2,3,4,5,6,7,8];
for (i=1,i<8,i++)
{
        ... A[i] ...
}
```

P'

```
int A = [6, 1, 8, 3, 2, 7, 4, 5]
for (i=1,i<8,i++)
{
        ... A[f(i)] ...
}
```

Listing 8.5: Data ordering transformation.

8.3.2 Layout Obfuscation

Layout obfuscation transformations are all the transformations that change the information included in the code formatting. For example, scrambling identifier names or the code indentation are layout obfuscation techniques.

Layout transformations are often *one-way* and cost *free* while their potency may vary depending on the transformation.

8.3.3 Control Obfuscation

Control obfuscation modifies the program control-flow to conceal information about the initial order of operations in the program.

Applying control obfuscation technique often implies an increase in the duration of the program execution. The programmer will have to choose between the highly efficient program P he intends to distribute and its highly obfuscated, but slower alternative P'.

Opaque predicates

Opaque predicates or variables are values known *a priori* to the obfucator but which are hard for the deobfuscator to compute.

Definition 16 (Opaque constructs)

- *Opaque variable: a variable V is* opaque *at a point p in a program if V has a property q at p that is known at obfuscation time, it is expressed as V_p^q.*
- *Opaque predicate: a predicate P is opaque at a point p in a program if its value (True or False) is known at obfuscation time. We write P_p^T if P is True at p, P_p^F if P is False at p and $P_p^?$ if P is sometimes True and sometimes False at p.*

This definition is exposed in Figure 8.2. An opaque construct is said to be *trivial* if a deobfuscator can deduce its value by static local analysis and it is said to be *weak* if a static global analysis is required to deduce its value.

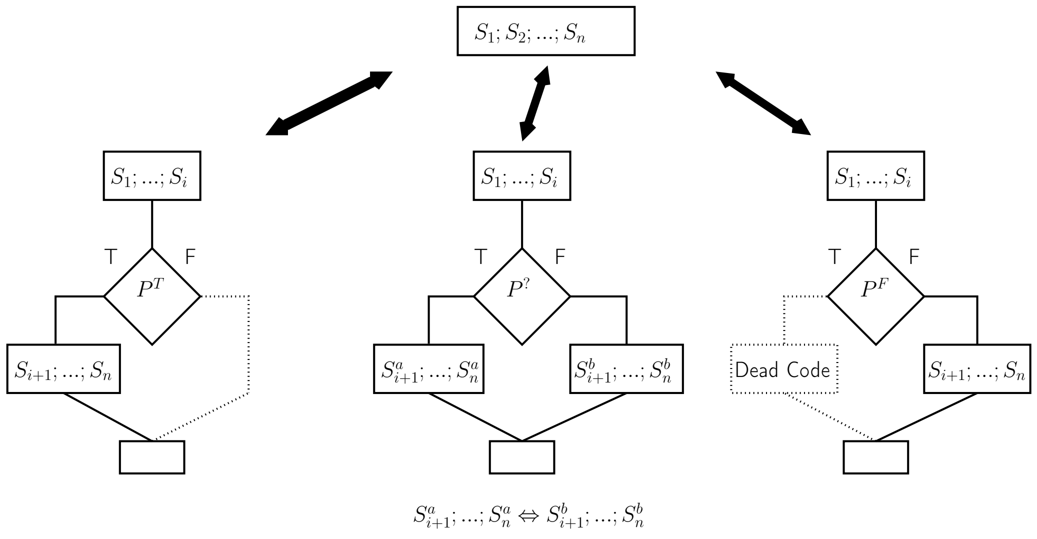


Figure 8.2: Opaque predicates.

Inserting dead code

Using opaque predicates enables the insertion of irrelevant code, for example it is possible to include harmful instructions inside the statement block of an `if` block with an opaque predicate P^F as the code within will never be executed. It is as well possible to take a block of existing code and move it in an `if` statement block using a opaque predicate P^T or putting two versions of the same code into an `if/else` statement using a $P^?$ opaque predicate, the deobfuscator will spend time trying to reverse-engineer the opaque predicate even though it does not matter which branch of the `if` statement will be executed.

Extending loop conditions

We can use opaque predicate to make loop termination conditions more complex without changing the number of iteration. For example, we could replace a condition C by $C\&\&P^T$.

Converting a reducible flow graph to a non-reducible one

Using the `goto` instruction combined with opaque predicates it is possible to make unused skips in the program resulting in an irreducible flow graph, *i.e.* a strongly connected graph.

Adding redundant operands

In the expressions, it is possible to add arithmetically redundant operands that will annihilate themselves as in Listing 8.6.

P	$P' with R^{=1}, P^{=2Q}, Q^{=P/2}$
`x=x+y;`	`x=x+y*R;`
`z=w+1;`	`z=w+(P/Q)/2;`

Listing 8.6: Redundant Operands.

Parallelizing code

Parallel programs might be harder to understand than sequential ones. Since nowadays there are plenty of tools for parallelism, we can use them to obscure the program control flow. The creation of *dead threads* that would appear as legitimate working threads and that could have some effect on the dead code inserted into the actual legitimate threads, would make this dead code necessary to the *dead threads*, therefore harder to detect. Of course, if the computer that runs the program cannot run more than one process at a time, these transformations will slow down the program, however in the obfuscation of a mono-threaded program, the cost would be smaller (assuming no deadlock).

Inlining or outlining functions

Inlining a function implies replacing calls to a function by the function code. Inlining is a *one-way* resilient transformation, it removes every abstraction set by the presence of the function.

Outlining instructions in a function means making a functions from a block of instructions. One use of outlining for obfuscation is to outline parts of semantically different procedures in a same function. (see 9.3).

P P'

```
s_1;                                  <typeA> foo(<args>){
s_2;                                  s_i+1;
...                                   ...
s_n;                                  s_j;
                                      }

                                      s_1
                                      ...
                                      s_i;
                                      foo(<args>)
                                      s_j+1;
                                      ...
                                      s_n;
```

Listing 8.7: Outlining of a block of statements.

Interleaving functions

Interleaving functions means merging two (or more) functions in one, merging body, arguments and returned results. The resulted function would take another argument for selecting which initial function to run.

Detecting function interleaving is really difficult for reverse engineers since it scrambles the semantics of the functions that were interleaved.

Cloning functions

For a given function, one writes several functions that have the exact same role and obfuscate each one in a different way. Then, each time the function is to be called, the programmer would call one of its clones instead.

Since the context of function call is used to understand the purpose of the function and since the body of the function is obfuscated, this transformation makes the understanding of the function role more difficult.

Loop transformations

Three loop transformations can be enumerated:

- Loop blocking means partitioning the loop iteration space in smaller loops. This is often used in optimisation as it usually reduces the CPU cache misses by making large vectors fit in the cache memory.
- Loop unrolling means replicating a loop body several times in order to reduce the number of iterations of the loop. This transformation is often used as a preliminary to the parallelization of the loop.
- Loop fission is a transformation that expands a loop with a compounded body into several loops with the same iteration space.

For each of these transformations, an example is given in Figure 8.8. Independently, these transformations have a fairly good potency but have a very low resilience since in most cases static analysis can counter these transformations. But when these transformations are used together, the program resilience increases dramatically.

P

```
for (i=1,i<=n,i++);
   for(j=1,j<=n,j++)
      a[i,j]=b[j,i];
```

P′ **(loop blocking)**

```
for (I=1,I<=n,I+=64)
   for(J=1,J<=n,J+=64)
      for(i=I,i<=min(I+63,n),i++)
         for(j=J,j<=min(J+63,n),j++)
            a[i,j]=b[j,i];
```

P

```
for (i=2,i<(n-1),i++)
   a[i] += a[i-1]*a[i+1];
```

P′ **(loop unrolling)**

```
for(i=2,i<(n-2),i+=2){
   a[i] += a[i-1]*a[i+1];
   a[i+1] += a[i]*a[i+2];
}
if (((n-2) % 2) == 1)
   a[n-1]+= a[n-2]*a[n];
```

P

```
for(i=1,i<n,i++){
   a[i] += c;
   x[i+i]=d+x[i+1]*a[i];
}
```

P′ **(loop fission)**

```
for(i=1,i<n,i++)
   a[i] += c;
for(i=1,i<n,i++)
   x[i+i]=d+x[i+1]*a[i];
```

Listing 8.8: Loop transformations (from top to bottom): Loop blocking, loop unrolling and loop fission.

Programmer's preference is to increase their code locality, making them more understandable. When obfuscating a program, we will want to *mix* pieces of the code (e.g. declarations of functions and variables) to remove the semantic meaning of this locality. Such transformations have low potency since they don't obscure the code that much, however, their resilience is *one-way* in most cases since once the transformation is applied, there is no information about the original order of the mixed pieces.

Building high resilience opaque structures

Predicate composed of simple arithmetical expression have *trivial* or *weak* resilience. Since the resilience of an opaque structure influences the quality of the transformation, one would like to have high resilience opaque structures. There are several methods for building resilient and cheap opaque structures ([64]).

One method to use is aliasing which uses the fact that two different pointers can point at the same data, one statement can therefore initiate this value to `True` while the second statement sets the value to `False` using the second pointer, the data referenced by the first pointer will be `False` even if it has been set to `True` previously. Trying to deduce properties from pointers is difficult since they refer to different memory spaces during the program execution.

Another method would be to take advantage of parallel processing of variable. A variable (or a pointer) modified by many threads would make a highly resilient opaque variable as it would be very difficult and time consuming to analyse statically as there are $n!$ ways to execute n parallel instructions on one processor.

8.3.4 Homomorphic Encryption as an Obfuscation Method

Homomorphic encryption is an encryption scheme which allows computation on encrypted data and returns an encrypted result. This means that the agent performing the computation never has access to plain text data.

For example, El Gamal [76] encryption algorithm is an homomorphic encryption scheme for the multiplication. Indeed, due its construction based on the discrete logarithm problem, the multiplication of e_1 and e_2 is the equivalent of the encrypted multiplication of plain text t_1 and t_2:

$$e_1 * e_2 = E(t_1 * t_2)$$

A agent which does not know the plain text can perform this multiplication and send the result to the owner of the key. The latter can then decrypt the data to retrieve the result of the multiplication. However the addition of encrypted data e_1 and e_2 does not lead to the encrypted addition of the plain text t_1 and t_2.

A fully homomorphic encryption scheme (for both addition and multiplication) based on lattices have been developed by Craig Gentry in [84]. However the time overhead is still too important to be used in practice. Another problem of such scheme is their lack of branching possibilities which is essential for a program to take decisions. Therefore a complete program transform in an encrypted one is not possible. Moreover the user of the obfuscated program needs to have access to the result of the computation, he therefore needs the key to decrypt the data. However, homomorphic encryption schemes could be used within the obfuscated program, one part would encrypt the data, another would perform the computation and the last could decrypt it.

8.3.5 Summary of the Obfuscation Metrics

The Table G.1 in appendix, extracted from [62] references the transformations previously mentioned. Transformations are classified by target and operation. The quality of each transformation is exposed and the metric(s) used to measure the transformations' potency is(are) enumerated according to the definitions in Section 8.1. On several cases, the transformation quality depends on the quality of opaque constructs or complexity of a data structure or function.

8.4 Deobfuscation

A deobfusactor takes an obfuscate program P' and tries to extract a program P'' which is as close as possible to the initial program P, or at least a more intelligible version of P'.

Some of the following deobfuscation's problems reduce to the *halting problem* making obfuscation undecidable:

- Eliminate dead code by determining whether a block of code will be reached or not.
- Evaluate at a point in the program if a variable is necessary to the computation of the result.
- Detect aliasing.

Appel ([39]) tackled the matter of a white box obfuscation of a program P. The obfuscator F is perfectly known to the public, but it uses a secret key K to obfuscate P, thus we have : $P' = F(P, K)$. The proof is based on an algorithm which tries to guess the source program S and the key K, verifying that the resulting program is the same as the obfuscated program. The different step takes at most polynomial time:

- Guess a source program S
- Guess a key K
- Compute $P' = F(S, K)$
- Check $P = P'$

Therefore white box deobfuscation is NP-easy, but this approach does not give any usable deobfuscation technique.

In practice deobfuscators try to eliminate bogus code that was inserted using opaque predicates. The detection of opaque predicates is therefore of first importance for deobfuscator. As they usually use pattern matching to identify opaque predicate, a way to avoid detection of the opaque constructs is to construct opaque structures as close as possible to real code.

Another technique used by deobfuscators is program slicing [164] used to reduce the deobfuscation problem into several smaller problems. But the aliasing or adding useless variable dependencies makes harder the identification of sliceable blocks.

When using static analysis, a deobfuscator can assume a construct to be opaque but it can be as well a legitimate construct. To prove that a predicate is indeed an opaque one, a reverse engineer can make a mutant version P_1 of the program P where the assumed opaque construct is set to its assumed value. If P and P_1 produce the same outputs for all possible inputs then the assumption was right.

The use of complex mathematical problems to build opaque structures can be investigated to avoid easy resolution of such structures, making opaque structures easy to compute for a specific case but hard for a more general one.

In practice, exploiting flaws of automatic deobfuscators and using `one-way` transformations can produce an obfuscated program which makes the reverse engineer unable to understand the program's code specially within a restricted time.

8.4.1 Deobfuscation is always possible?

From a more theoretical point of view a lot of work has been done in obfuscation and on its impossibility. Arbitrary programs have been proven impossible in 2001 [45] by Barak et al. and with auxiliary input in [85]. In [45] they proved that perfect obfuscation is impossible. In their demonstration, they considered an obfuscated program as a *virtual black box*. The *virtual black box* property stipulates that "Anything that can be efficiently computed form $O(P)$ can be efficiently computed given oracle access to P, *i.e.* the fastest way of reverse engineering the virtual black box is by analysing the input and the resulted output.

To prove that obfuscation is in the general cases impossible, they built a class of simple functions and proved by contradiction that it is impossible to obfuscate them.

However they did not prove this result for any function. There even exist simple functions such as point functions that can be obfuscated [57], [58] or in [163].

These results lead to two possible direction in obfuscation theory, either settle with a lower but still meaningful notion of obfuscation (for example *virtual gray box* [54])or build obfuscators for a restricted set of functions.

8.5 Conclusion

Software obfuscation has many applications, not only allowing the protection of a program's secrets (data or algorithms) but also, birthmarking, watermarking or even tamperproofing.

Like software compilation, program obfuscation is a matter of transformations that have to be applied in a correct order to provide an optimal result, a set transformations having different potency and resilience depending on their order. Since software complexity metrics can be ambivalent, the evaluation of the quality of an obfuscation transformation is not simple as some transformations can be more or less efficient depending on the program, on the programming languages used and on metrics selected. Moreover, some transformations may not be available in some languages. Optimisation techniques such as Evolutionary Algorithms (EAs) can be used to find good solutions to this problem.

Although it has been proved that, in theory, program obfuscation is impossible as *virtual black box*, today's deobfuscators have many flaws that can be exploited to slow down reverse engineers and to protect the secrets within the program at least for a limited period of time.

CHAPTER 9

JShadObf: a JavaScript Obfuscator based on Multi-objective Optimisation Algorithms

Contents

This Chapter presents one of the major contributions of the thesis, the obfuscation of `JavaScript` code using Evolutionary Algorithm (EA). These code modifications are performed by JSHADOBF using commonly used computer software metric to assess the level of obfuscation of the code. The different transformations implemented will be presented using examples on a small `JavaScript` code.

9.1 Context: Obfuscation of `JavaScript` Programs

The obfuscation of a JavaScript program has been addressed by the Internet community as well as by hackers to embed malicious code in websites, for instance, to redirect requests to another website so as to increase the number of visitors. In reaction, the research community has done many studies to detect malicious websites and thus obfuscated code through the analysis of JavaScript source code.

For instance, the authors in [56] analyse the string variables contained in the JavaScript source code to classify the websites as malicious or not. The metrics used in JShadObf are string related metrics (this includes string length, frequency of particular function or entropy of the function or variable names). Following the same trend, the authors of [78] have patched the SpiderMonkey [15] JavaScript interpreter (which is used in the well known web browser `firefox`) to make some statistics about the code, like counting the number of newly generated strings or counting the number of `eval` calls during the execution of the program to help detecting obfuscated code. As regards the obfuscation process in itself, there exist relatively few studies or tools in the literature.

Indeed the current JavaScript obfuscation techniques mostly use data obfuscation on string variables and the `eval` function provided by the JavaScript language allowing the dynamic execution of strings. These techniques are used for instance by the on-line JavaScript obfuscator [7] or in packer [12]. Another tool worth to mention is `UgligyJS`, a JavaScript obfuscator, compressor (minifier) or beautifier. Depending on the user's request, it performs many small optimisations and obfuscations to alter the initial JavaScript source code. The possible transformations are listed on the `UgligyJS` website [20]. ObfuscateJS has been as well included in the tools tested against JShadObf even if the last version of the software dates from 2006 [9]. The non-free JavaScript obfuscator, Jasob [6] has been used for the tests with the trial version. The other ones worth to mention in this section are minifiers, an operation that can be seen as a partial obfuscation as it complexifies the JavaScript code. The minifier of Yahoo, called YUI Compressor [23], works on a few simple examples but fails to parse complex codes such as the ones of the `JQuery` library. A more powerful alternative is the minifier of Google called `closurecompiler` [3].

To sum up, obfuscation of JavaScript programs is still at an early stage and there is place for huge improvements. That's where comes the main contribution (and interest) of JShadObf, an obfuscation framework based on evolutionary heuristics designed to optimize *for a given input JavaScript program*, the sequence of transformations that should be applied to the source code to improve its obfuscation capacity.

9.2 The JShadObf framework

The general obfuscating process operated by JShadObf is illustrated in Figure 9.1. From the initial program $\mathcal{P}$ to be obfuscated (`myfile.js`), a reference individual (in the Evolutionary Algorithm (EA) sense) I_{ref} is generated that represents $\mathcal{P}$. Then, a complete population of n individuals is generated by randomly applying a mutation (i.e. a transformation) on the reference individual I_{ref}.

The Multi-Objective Evolutionary Algorithm (MOEA) process (NSGA-II [71] or MOEAD [109], see Section 9.3) then intervenes to explore the search space induced by the six objectives reviewed in the preceding section, to evaluate the population and to apply the genetic operators (mutation and cross-over). This permits to exhibit at each generation non-dominated Pareto solutions, each of them representing a set of derived (and hopefully sufficiently obfuscated) versions of the program $\mathcal{P}$ that propose a good trade-off between each objectives, *i.e.* metrics.

The implementation of the solution had to answer the four following issues:

1. How to parse a JavaScript program? (Chapter B and Chapter A).

   ↪ The JavaScript code is parsed using the grammar AntLR [132] developed with the help of the ECMAScript Standardization document [96] to build an Abstract Syntax Tree (AST) of the considered program;

2. What transformations can be applied ? (Section 8.3 and Section 9.4).

   ↪ Twelve different transformations are applied to JShadObf individuals by the evolutionary operators;

3. How do you measure their efficiency? (Section 8.2).

   ↪ The six metrics permits to evaluate the obfuscation capacity ($\mu_1, \ldots, \mu_5$) or the performance (μ_e) of a given individual. They will be used in the corresponding fitness functions to compute the fitness values of the considered individual *i.e.* its quality. As we decided to explore the search space to minimize all these values, the fitness functions are defined as follows: $\forall x \in [1..5], \mathit{fitness}_x(I) = 1/\mu_x$. only the time of execution μ_e is directly the fitness value: $\mathit{fitness}_e(I) = \mu_e$;

4. How to use Evolutionary Algorithms (EAs) to increase the obfuscation level of an *individual*? (Section 9.3).

   ↪ MOEAD [109] and NSGA-II [71] MOEA are used in the selection step.

9.3 Evolutionary Algorithms (EAs)

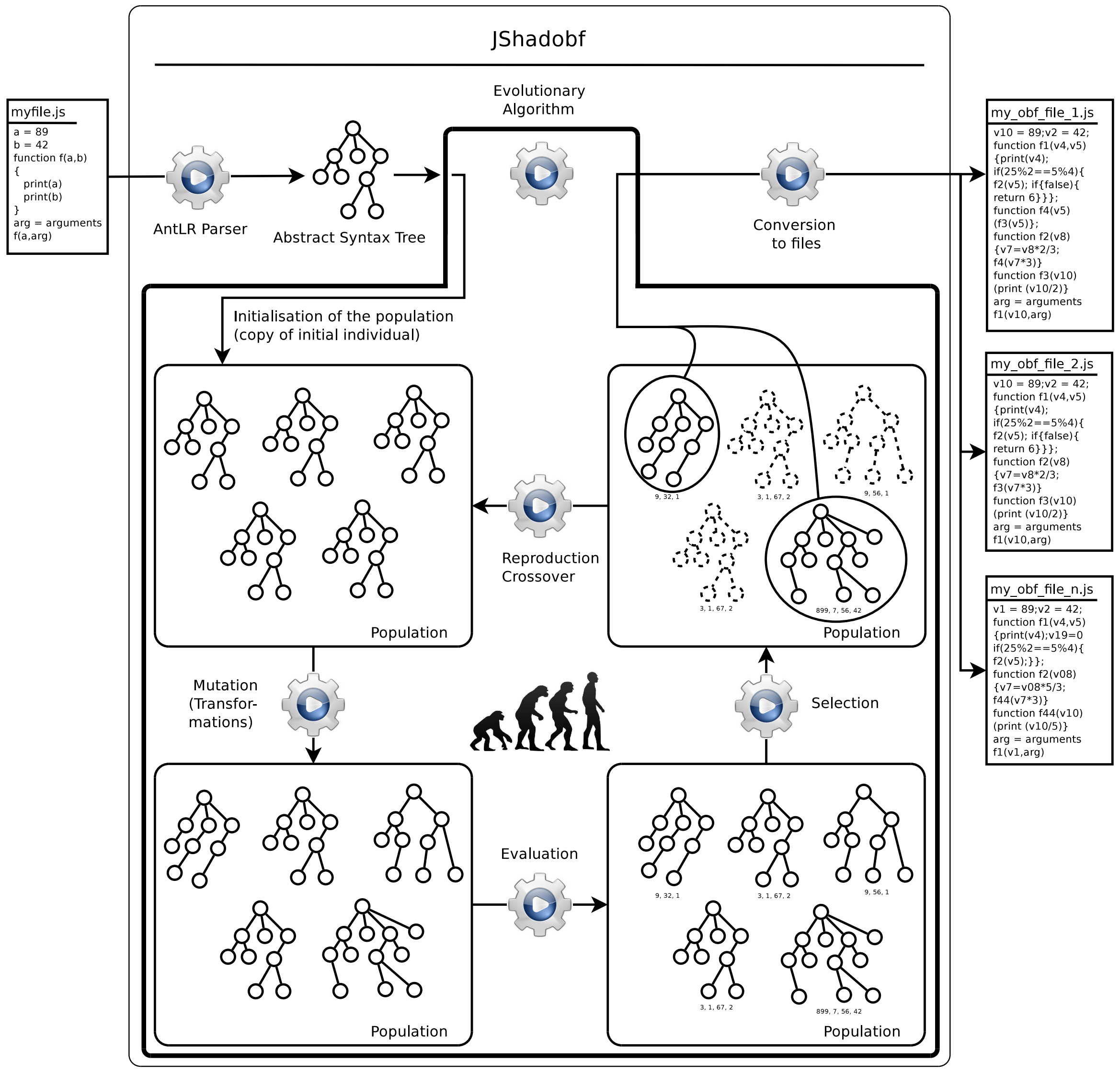
Evolutionary Algorithms (EAs) use mechanisms inspired from the Darwinian theory of evolution [70] to solve optimisation problems by the means of reproduction, mutation, recombination and selection. EAs involve generations of individuals, each individual being a potential solution of the problem to solve. Each generation is built from the previous one through mutation, recombination, etc. Then, the individuals are evaluated according to a fitness function which eventually involves multiple criteria. The algorithm halts when a stopping condition is met (typically, the fitness value has reached a given threshold or an optimal solution has been found). Note that the convergence of EAs towards "good" solutions has been formally proven in [139]. Since the convergence of an EA can be very long one can choose to stop an EA when a fixed number of generations have been computed or when the fitness of the individuals does not change much between two consecutive generations.

Execution of simple EAs requires high computational resources in case of non-trivial problems, in particular the evaluation of the population is often the costliest operation in EAs. There exist many useful models of EAs, yet a pseudo-code of a general execution scheme is provided in the Algorithm 9.3.

Algorithm 1 General scheme of an EA in pseudo-code.

```
$t \leftarrow 0$;
Generation($X_t$); // generate the initial population
Evaluation($X_t$); // evaluate population
while Stopping criteria not satisfied do
    $\hat{X}_t \leftarrow$ ParentsSelection($X_t$); // select parents
    $X'_t \leftarrow$ Modification($\hat{X}_t$); // cross-over + mutation
    Evaluation($X'_t$); // evaluate offspring
    $X_{t+1} \leftarrow$ Selection($X_t$,$X'_t$); // select survivors of the next generation
    $t \leftarrow t + 1$;
end while
```

EAs are popular approaches to solve various hard optimisation problems: on average, they generally converge to "*good*" solutions more quickly than the naive exhaustive search algorithm.



myfile.js
```
a = 89
b = 42
function f(a,b)
{
   print(a)
   print(b)
}
arg = arguments
f(a,arg)
```

my_obf_file_1.js
```
v10 = 89;v2 = 42;
function f1(v4,v5)
{print(v4);
if(25%2==5%4){
f2(v5); if{false){
return 6}}};
function f4(v5)
(f3(v5)};
function f2(v8)
{v7=v8*2/3;
f4(v7*3)}
function f3(v10)
(print (v10/2)}
arg = arguments
f1(v10,arg)
```

my_obf_file_2.js
```
v10 = 89;v2 = 42;
function f1(v4,v5)
{print(v4);
if(25%2==5%4){
f2(v5); if{false){
return 6}}};
function f2(v8)
{v7=v8*2/3;
f3(v7*3)}
function f3(v10)
(print (v10/2)}
arg = arguments
f1(v10,arg)
```

my_obf_file_n.js
```
v1 = 89;v2 = 42;
function f1(v4,v5)
{print(v4);v19=0
if(25%2==5%4){
f2(v5);}};
function f2(v08)
{v7=v08*5/3;
f44(v7*3)}
function f44(v10)
(print (v10/5)}
arg = arguments
f1(v1,arg)
```

Figure 9.1: Overview of JShadObf, describing the full process of the generation of new representations of the code.

EA components in JShadObf

We now review the key components of the EA at the heart of JShadObf when dealing with a population of N individuals.

Individual : An individual is an AST representing the source code.

Original individual : The first individual (from which all the individuals were mutated) representing the AST of the code to obfuscate.

Degenerate individual : A degenerate individual an invalid individual, which means that the output of the corresponding program is different from the original individual. Due to the complexity created by the multiple applications of the transformations, such individual might appear. This should not happen with well tested transformations.

Population : A set of individuals

Mutation:

Individual mutation corresponds to the application of a randomly selected transformation, on a randomly selected portion of the AST assuming it makes sense for the considered transformation. This transformation is applied at most n times, with n selected by the user at the beginning of the process, allowing the limitation of the number of transformations at each step. An history of the applied transformations and their application context is kept with each individual as it will be used in the cross-over step. Note that JShadObf always applies a mutation to each individual (*i.e.* the mutation probability is 1) thus leading to N new offspring.

Reproduction/Crossover:

Standard crossover operators such as one point crossover, two point crossover, cut and splice or any combination would have a very high probability of generating a degenerate individual. Indeed in the case of individual being source code, merging two individuals require cutting in the code at a random place which will lead in most cases to a degenerated individual. In the case of individual being AST this would be more feasible, replacing for example an obfuscated function $o_1(f)$ by the same initial function but obfuscated in the second individual $o_2(f)$. This however have good chances of leading to degenerated individual as well due to the different transformations which are changing the scope of the variables.

Therefore we introduced a safer cross-over operator as follows: two *individuals* I_1 and I_2 are randomly selected, they will act as parents. Then, a new offspring is generated by:

1. selecting randomly a transformation T from the history of mutations applied on I_1;

2. applying this transformation on I_2 and returning the newly created individual.

The cross-over step is applied for each individual of the population with a probability selected by the user thus leading to k new offspring ($0 \leq k \leq N$). In total, $2N + k$ individuals are available after the evolutionary operators (mutation and cross-over). Among them, N will survive to the next generation during the selection step.

The selection of already applied transformations in other individuals in the population to perform the cross-over relies on the fact that if these individuals have been selected in the past it is because they are better than the non selected one, and this is partly due to type of transformations which have been applied on them. In Figure 9.7, one individual has been selected in the approximated Pareto and its transformations have been applied to a 1000 individuals. And this newly created population performs better that the population of individuals based on a totally random sequence of transformations.

Evaluation:

This stage is performed by computing the fitness functions defined previously. First, the five static metrics are computed by browsing through the AST representation, then a sub-process is launched to run the *individual* with standard input, leading preferably to a deterministic output, and the time taken by the sub-process to complete its task is the sixth metric. To counter the non-regularity of the execution time of process, this dynamic metric is computed `n` times, and the average is taken as the result of the computation. This dynamic evaluation of the *individuals* guarantees as well that the population stays valid according to the definition of obfuscation.

Selection:

The two evolutionary algorithms (NSGA-II and MOEAD) compared in this thesis differ in this stage and will be explained in the next sections (Section 9.3.1 and Section 9.3.2). The selection uses the values of the fitness function computed during the evaluation part of the EA.

As the different metrics used are nearly independent from each other, they have to be taken into account in the EA, thus JShadObf relies on Multi-Objective Evolutionary Algorithm (MOEA) to search for solutions of the obfuscating process. MOEA is a subclass of EA which is considering the optimisation of more than one parameter at the same time. It therefore requires adapted selection mechanisms, as for a single objective optimisation it is more obvious to designate the best individual, for multi-objective it is less evident. The set of good individuals in a multi-objective problem is represented by a Pareto front. This notion comes from the economist Vilfredo Pareto [131] and is the set of solutions that are not dominated by other solutions. An *individual* is said to be non-dominated if it is not dominated by any other *individual* in the population.

The concept of dominance is the following (in the case of minimization):

Definition 17 (Domination) *An* individual *I with the objectives values $f_{obj}(I)$ is said to be dominated by J if*

$$\forall obj \in objectives, f_{obj}(J) < f_{obj}(I)$$

The Algorithm 2 extracts the Pareto front from a population.

Algorithm 2 Pareto front creation.

```
$Pf = \emptyset$;
for $i$ in $P$ do
    $dominated = False$ ;
    for $j$ in $P$ do
        if $i \prec j$ then // if j is dominating i i.e. has better values for all objectives
            $dominated = True$ ;
            $break$ ;
        end if
    end for
    if not $dominate$ then
        $Pf = Pf + i$ ;
    end if
end for
```

9.3.1 Selection in NSGA-II.

Throughout NSGA-II [71] (one of the reference selection algorithm for MOEAs considered in our initial design), *individuals* are selected by taking into account the non-domination criteria and the distance from one to the others to guarantee a good diversity as well as the leading *individuals* of the population.

NSGA-II is selecting all the non-dominated solutions of the population, and if the size of the new population is lower than the maximum size, NSGA-II is selecting again all the non-dominated solutions of the old population but this time excluding the already selected *individuals* (see Algorithm 3).

In this Algorithm 3 the `extractPf` function is similar to the Algorithm 2 except that it removes the *individuals* from the Population. The `Crowding_distance` function is computing the distance between the *individuals* using the objective functions, and to select the *individuals* with the longer distance from the closest one.

9.3.2 Selection in MOEAD.

The MOEAD [109] selection is based on weight vectors depending on the number of objectives used to weight the objective values of each *individual.* There are as many vectors as the size of the population and one *individual* can be the best solution for more than one vector. This solution is then associated with the weight vector. The weight vector in 2 dimensions is the following $[r, 1-r]$

Algorithm 3 NSGA-II selection algorithm (pseudo-code).

```
$NewPopulation = \emptyset$
while sizeof($NewPopulation$) <$PopulationMax$ do
    $Pf$ = extractPf($Population$); // extract and remove the Pareto front from the population
    $leftSize$ = $PopulationMax$−sizeof($NewPopulation$) ;
    if sizeof($Pf$)< $leftSize$ then // If there is enough space left
        $NewPopulation = NewPopulation \cap Pf$ ; // add the whole next Pareto front to the new population
    else
        // add only a subset of the Pareto front $Pf$ chosen considering the distance between the individual in $Pf$
        $NewPopulation = NewPopulation \cap$Crowding_distance($Pf$,$leftSize$) ;
    end if
end while
```

with $r \in [0, \frac{1}{pop\ size-1}, \frac{2}{pop\ size-1}, ..., 1]$. The sum of each vector should be equal to 1. Each vector has a neighbourhood of size N, which are the N closest other weight vectors (using classical distance function). To perform the selection, MOEAD picks one weight vector v_I and its associated *individual* I, then picks another *individual* J and its associated weight vector v_J either in the neighbourhood of v_I, or in the whole population, and compares them using the following function:

$$g^{te}(I, v) = \max_{obj \in objectives} (v_{obj} * |f_{obj}(I) - z_{obj}|)$$

with z_{obj} corresponding the best objective value within the population:

$$z_{obj} = \min_{I \in Population} (f_{obj}(I))$$

If $g^{te}(I, v_J) < g^{te}(J, v_J)$, then I become the *individual* associated to v_J. The procedure is reiterated until the algorithm reaches a certain number of replacements or if the neighbourhood has been browsed completely. This allows to keep *individuals* in the population even if dominated by other *individuals*. Such selections generate a higher diversity of the population and is performing better with many objectives.

The cycle of the EA continues until a certain number of generations has been reached, or until the different fitness values obtained are small enough and fit the user's requirements. However, for the *individuals* to evolve, transformations have to be developed and applied on the population.

9.4 Considered Transformations

The different transformations developed within the framework of JShadObf are applied directly on the AST, and are not modifying the output of the program. They are simple in order to be easier to check, but applied many times on different parts of the program, they make the program harder to read.

To illustrate the different transformations, the `JavaScript` code in Listing 9.1 will be used.

Listing 9.1: Reference program for presenting the transformations.

```
function fibo(n)
{
  if( n <= 1 )
  {
    return n;
  }
  var res = fibo(n−1) + fibo(n−2);
  return res;
}
var res = fibo(10)
print(res)
```

9.4.1 Renaming

This transformation is modifying and changing the name of some identifier randomly (except the identifiers which are used globally). It is as well possible to avoid the renaming of some names by specifying them beforehand, indeed some names are used by convention, *e.g.* the variable `arguments` in `nodejs` reefers to the arguments given to the script through the command line,

Listing 9.2: An instance of the renaming transformation on the sample example.

```
function a93 (a30) {
  if((a30 <= 1)){
    return a30;
  }
  var a40 = (a93((a30 - 1)) + a93((a30 - 2)));
  return a40;
}
var a40 = a93(10);
print(a40);
```

9.4.2 Outlining

The outlining transformation takes a set of statements and outlines them. This creates a new function either in the same scope or in an higher scope depending on the side effects of the selected set (see Section 8.3.3).

Listing 9.3: Outlining.

```
function fibo (n) {
  if((n <= 1)){
    return n;
  }
  var res = (fibo((n - 1)) + fibo((n - 2)));
  return res;
}
var res = fibo(10);
function dummyvar398 () {
  print(res);
  return [];
}
var dummyvar491 = dummyvar398();
```

9.4.3 Dummy If Insertion

This transformation adds dummy `if` statements with randomly generated predicate as in Listing 9.4.

Listing 9.4: Dummy-if insertion.

```
function fibo (n) {
  if((n <= 1)){
    if((860 < ([200,474,598,884,572,581,860,91,230,655,808,150]  [6]) )){

    }
    else{

    }
    return n;
  }
  if  ((([421,914,568,577,811,92,353,7,90,963,861,95,332,479,555,645,848,141,828]   [0])  == 844)){
    var res = (fibo((n  -  1)) + fibo((n  -  2)));
    return res;
  }
  else{
    var res = (fibo((n  -  1)) + fibo((n  -  2)));
    return res;
  }
}
if  ((([768,200,244,113,155,337,481,871,291,273,105,37,866,916,686,792,942,38]   [16])  != 46)){
  var res = fibo(10);
}
else{
  var res = fibo(10);
}
print(res);
```

9.4.4 Dummy Variable Insertion

This transformation adds unused variables in the code as in Listing 9.5.

Listing 9.5: Dummy-variables insertion.

```
function fibo (n) {
  if((n <= 1)){
    var dummyvar122;
    return n;
  }
  var res = (fibo((n  -  1)) + fibo((n  -  2)));
  return res;
  var dummyvar423;
}
var res = fibo(10);
var dummyvar459;
print(res);
```

9.4.5 Dummy Expression Insertion

The generation of random expressions and their insertion at random position in the code as in Listing 9.6.

Listing 9.6: Dummy-expressions insertion.

```
function fibo (n) {
  'kbnzuwcn icbyqlip xlvfgkmk wvinbmzr';
  if((n <= 1)){
    '' ^ true;
    return n;
  }
  var res = (fibo((n  - 1)) + fibo((n  - 2)));
  return res;
}
['',true,true,'tybcfmcx aujvvzpf mqttmldw',false,true,true,true];
var res = fibo(10);
print(res);
```

9.4.6 Replace Static Data by Variables

When applicable, the transformation presented in Listing 9.7, replaces static data by variables (see Section 8.3.1).

Listing 9.7: Replace static data.

```
function fibo (n) {
  var dummyvar384 = 1;
  var dummyvar459 = 2;
  var dummyvar156 = 1;
  if((n <= dummyvar384)){
    return n;
  }
  var dummyvar132 = dummyvar156;
  var res = (fibo((n  - dummyvar132)) + fibo((n - dummyvar459)));
  return res;
}
var res = fibo(10);
print(res);
```

9.4.7 Aggregate Data

This transformation aggregates constant values into a vector as in Listing 9.8 (see Section 8.3.1).

Listing 9.8: Aggregate data.

```
var dummyvar179 = [10];
function fibo (n) {
  var dummyvar184 = [2,1,1];
  if((n <= (dummyvar184 [1]))){
    return n;
  }
  var res = (fibo((n  -  (dummyvar184 [2]))) + fibo((n -  (dummyvar184 [0]))));
  return res;
}
var res = fibo((dummyvar179 [0]));
print(res);
```

9.4.8 Modifying Control Flow

This transformation replaces a sequential code by a switch structure inside a while loop, with the initial code cut and randomly distributed into the different cases of the switch structure like in Listing 9.9.

Listing 9.9: Modifying control flow.

```
function fibo (n) {
  if((n <= 1)){
    var dummyvar4;
    (dummyvar4 = 77074);
    while ((52563 != dummyvar4)) {
      switch (dummyvar4) {case 77074 : return n;
        (dummyvar4 = 52563);
        break ;}
    }
  }
  var res = (fibo((n  -  1)) + fibo((n  -  2)));
  return res;
}
var res = fibo(10);
print(res);
```

9.4.9 Changing the List Declarations

In this case, a static list definition is divided into definitions and concatenations of subsets of the initial list. This Transformation could not have been applied to the reference program as there is no list in it. Therefore Listing 9.10 presents the transformation applied to Listing 9.8.

Listing 9.10: Changing the list declarations.

```
var dummyvar288 = [2];
var dummyvar240 = [1,1];
var dummyvar179 = [10];
function fibo (n) {
  var dummyvar184 = dummyvar288.concat(dummyvar240);
  if((n <= dummyvar184[1])){
    return n;
  }
  var res = (fibo((n  -  dummyvar184[2])) + fibo((n -  dummyvar184[0])));
  return res;
}
var res = fibo(dummyvar179[0]);
print(res);
```

9.4.10 Duplicating Functions

This transformation copies a part of the code of a function and changes the name to an unused one. This is like dead code insertion, but the inserted code is similar to the existing code as in Listing 9.11.

Listing 9.11: Function duplication.

```
function fibo (n) {
  if((n <= 1)){
    return n;
  }
  var res = (fibo((n  - 1)) + fibo((n  - 2)));
  return res;
}
function fibo12 (n) {
  return res;
}
var res = fibo(10);
function fibo11 (n) {
  return res;
}
function fibo1 (n) {
  if((n <= 1)){
    return n;
  }
  return res;
}
print(res);
```

9.4.11 Use `eval` Obfuscation Power

In `JavaScript`, the `eval` function is parsing and executing the string given in argument as `JavaScript` code. It executess this code within the context of the `eval` function call. Used with string transformations, this function makes the code less readable for a human programmer.

This transformation selects a block of code and gives it as a string argument to the evil function `eval` as in Listing 9.12.

Listing 9.12: Eval function usage.

```
eval('function fibo (n) {if((n <= 1)){return n;}var res = (fibo((n - 1)) + fibo((n
    - 2)));return res;}');
eval('var res = fibo(10)');
eval('print(res)');
```

9.4.12 Re-formatting String Constants

For this operation, the string constants are replaced by concatenation of sub strings contained in variables which have been declared in the same scope of the program (see Section 8.3.1). This Transformation could not have been applied to the reference program as there is no string in it. Therefore Listing 9.13 presents the transformation applied to Listing 9.12.

Listing 9.13: Outlining.

```
var dummyvar673 = "{return'";
var dummyvar47 = 'fibo((n - 1)) + fibo((n';
var dummyvar498 = "'tion fib";
var dummyvar662 = '))';
var dummyvar34 = 'n;}var res = (';
var dummyvar166 = 'o (n) {if((n <= 1';
var dummyvar184 = eval((dummyvar498 + dummyvar166 + dummyvar662 + dummyvar673));
var dummyvar154 = '"';
var dummyvar351 = ' - 2)));return res;}\'"';
var dummyvar135 = eval((dummyvar154 + dummyvar34 + dummyvar47 + dummyvar351));
var dummyvar119 = ' ';
var dummyvar67 = "'func";
eval(eval((dummyvar67 + dummyvar184 + dummyvar119 + dummyvar135)));
eval('var res = fibo(10)');
eval('print(res)');
```

These transformations have been tested on a test-suite of JavaScript programs, to ensure that the first requirement of the definition of a obfuscation holds, *i.e.* keeping the functionality of the program intact. Because transformations applied on the source code can interfere between themselves, JShadObf uses very simple transformations tested on multiple JavaScript programs as well as many different combinations to ensure their validity. The composition of the simple transformations developed in this work is commutative because they operate independently on the individuals. However, as individuals in JShadObf records any applied transformations, it allows to take into account the case of non-commutative transformations, *i.e.*, the application on an individual I of Obf_a forbids a future application of Obf_b or on the contrary Obf_b requires beforehand the application of Obf_a on I. This can indeed appear when dealing with more complex transformations where the set of applicable transformations depends on the previous sequence of modifications [89].

9.5 Experiments

We have validated our approach over two concrete examples: one pedagogical (a classical matrix multiplication program `matmul.js` which is outputting the result of the multiplication of two matrix) and one more serious on the most popular and widely used JavaScript library, named `JQuery` [30]. For the Matrix multiplication program the MOEA used a population of a size 1000 *individuals*, performing over 50 generations. The size of the population for the `JQuery` framework is 100 *individuals* which is smaller due to the size of the initial program which is approximately 8000 lines of code (160ko without comments), and the number of generations as well 50. JShadObf can as well be run with as target not the number of generation but rather some values for the objective functions. This however could lead to infinite computation if the objectives are not reachable.

9.5.1 Experiments on a Matrix Multiplication Program

The graphs in Figure 9.2, Figure 9.5.1 and Figure 9.5.1 illustrate the relation between two of the obfuscation metrics (μ_1 and μ_2, μ_1 and μ_6, μ_4 and μ_6) every tenth generation for the Matrix multiplication program in a 2D front view. The graphs in Figure 9.8 are showing the approximated Pareto front for the 70th generation of the two different algorithms, *i.e.* NSGA-II and MOEAD.

The Pareto fronts are easily distinguishable on the different graphs in Figure 9.5.1, Figure 9.5.1 and Figure 9.2, they have different shapes as they are projections of the results of a MOEA algorithm with six objectives. On the different graphs some alignment of the *individuals* may appear (like when

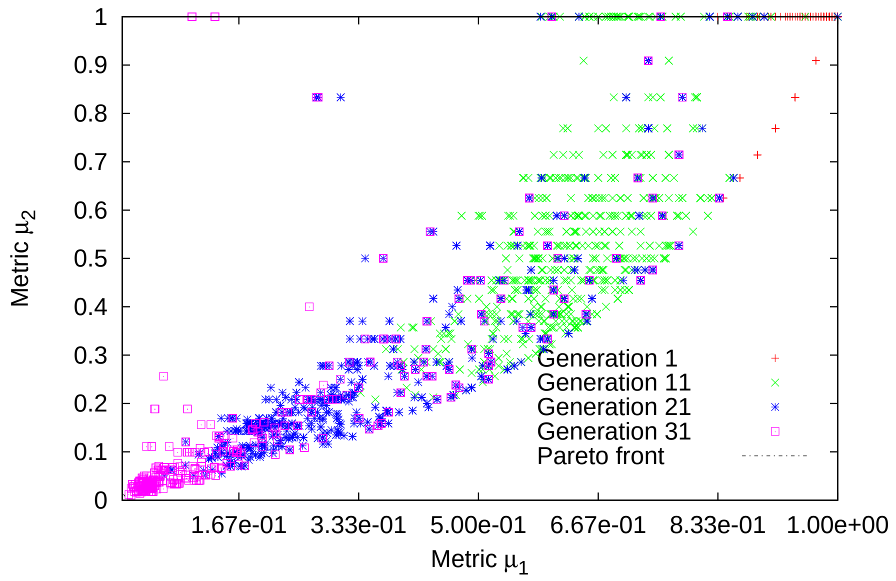


Figure 9.2: 2D Pareto front (μ_1 and μ_2) obtained for the obfuscation of a Matrix Multiplication program by JSHADOBF. With μ_x actually being $\mu_{x_{init}}/\mu_x$.

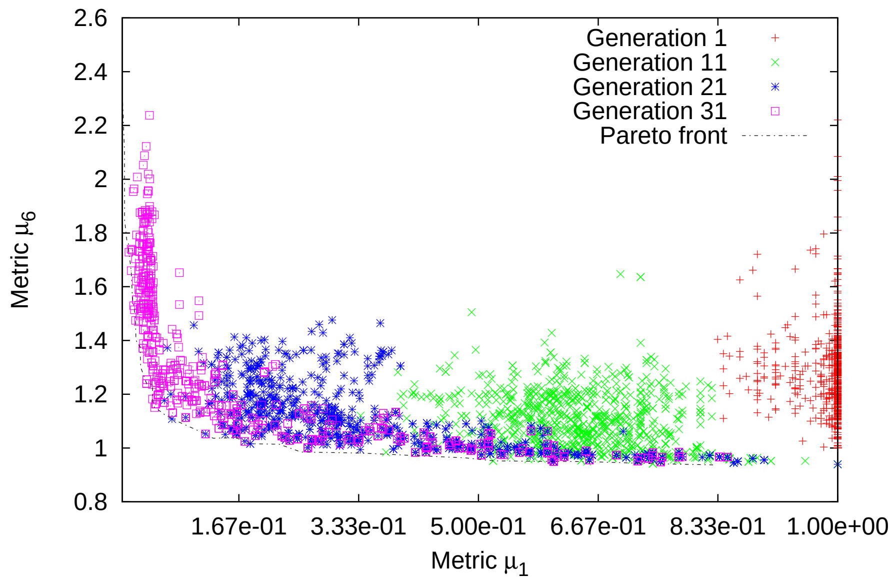


Figure 9.3: 2D Pareto front (μ_1 and μ_6) obtained for the obfuscation of a Matrix Multiplication program by JSHADOBF. With μ_1 actually being $\mu_{1_{init}}/\mu_1$ and μ_6 actually being $\mu_e/\mu_{e_{init}}$.

comparing μ_2 and μ_6, or when evaluating μ_1 and μ_2), this is explained by the way the fitness function is computed, indeed it is often the inverse of an integer value.

The Figure 9.5 represents the mean values $mean_x(n^{th})$ of the fitness function $fitness_x$ of a

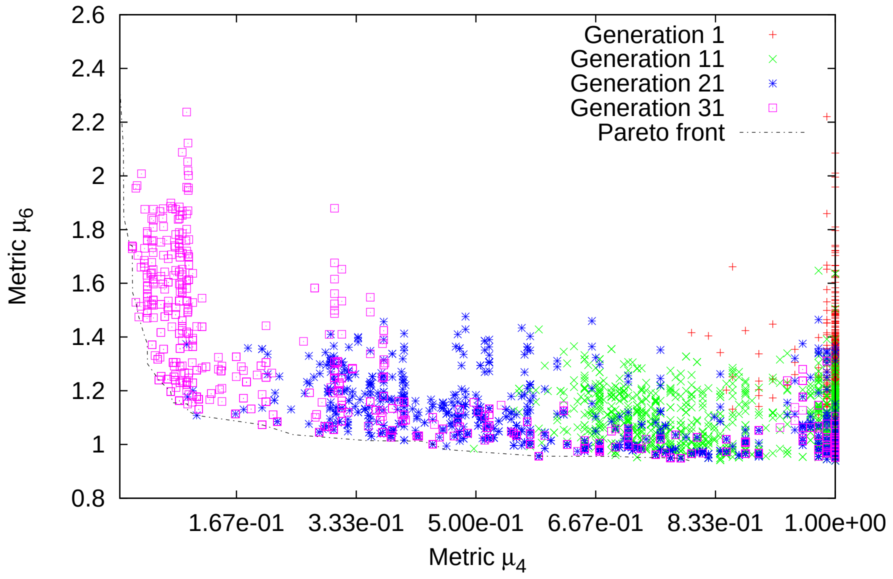


Figure 9.4: 2D Pareto front (μ_4 and μ_6) obtained for the obfuscation of a Matrix Multiplication program by JSHADOBF. With μ_4 actually being $\mu_{4_{init}}/\mu_4$ and μ_6 actually being $\mu_e/\mu_{e_{init}}$.

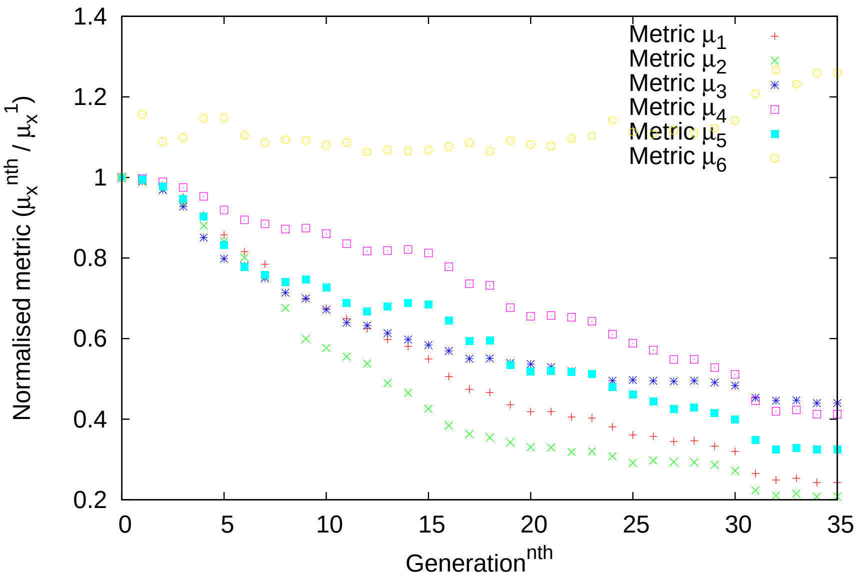


Figure 9.5: Evolution of the mean values of the different metrics (the lower the better) used in the obfuscation of a Matrix Multiplication program by JSHADOBF

generation n^{th} of size m which is computed as follows:

$$mean_x(n^{th}) = \frac{\sum_{i=1}^{m} fitness_x(I_i^{n^{th}})}{m}$$

These mean values are then normalised with the mean values of the first generation to be able to represent them on the same graphic:

$$f_x(n^{th}) = \frac{mean_x(n^{th})}{fitness_x(I^{1^{st}})}$$

The function f_x is computed with an argument, *i.e.* the generation's number n^{th}, for all the six metrics μ_x selected. One can then see the evolution of mean of the population regarding the selected metrics. On one of the individuals of the last generation, which has been chosen a priori, the metrics presented in the table 9.2 has been computed to show the evolution from the initial program and to compare with other obfuscator / minifier.

9.5.2 Experiments on `JQuery`

For the second experiment, the decision to apply JShadObf on a more serious application was made. For this reason, one of the most popular and widely used JavaScript library, *i.e.* `JQuery` [30], has been selected, in the hope that it can illustrate the robustness and the usability of the technique presented here. `JQuery` is a fast and concise JavaScript Library that simplifies HTML document traversing, event handling, animating, and Asynchronous JavaScript and XML (AJAX) interactions for rapid web development. `JQuery` in its development version is distributed with a test-suite verifying, in the version used, 3884 assertions.

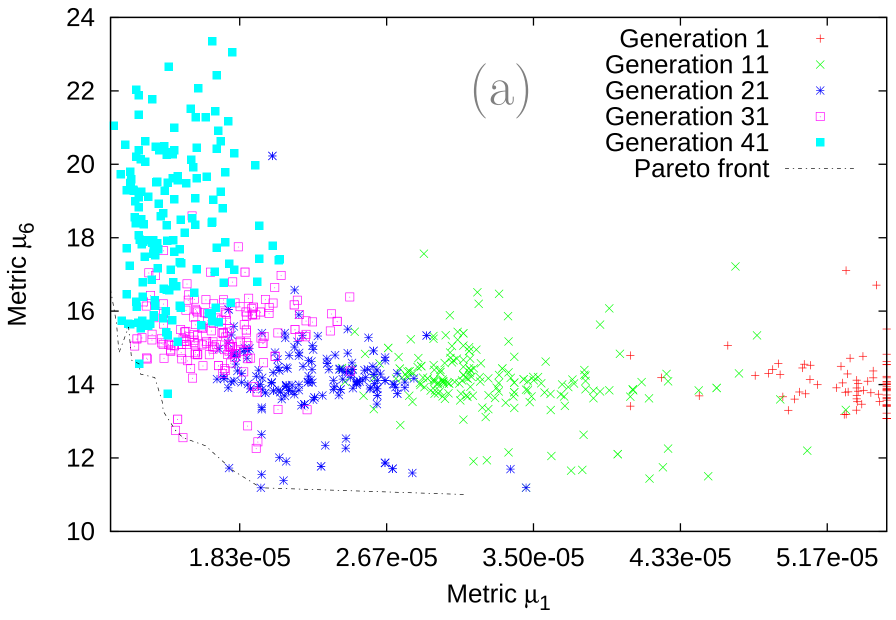
We use this test suite as a way to demonstrate the correct behaviour or the library, knowing that the original non-obfuscated version of `jquery.js` fails on 7 assertions over the 3884. The figure 9.6 depicts the evaluation of the different obfuscators/minifiers for the `JQuery` framework against this test suite. It is worth to notice that when the considered framework is able to parse completely the library and thus to generate an obfuscated version, we observe the same behaviour *i.e.* 7 assertions failed. Obviously, it would have been very unlikely to decrease this value as all frameworks derive the original code. At least it proves that the obfuscated/minified versions do have the same observable behaviour.

9.5.3 Comparison of MOEA with random selections of transformations

In order to compare JShadObf with a random selection of transformations applied on the matrix multiplication program, one individual has been selected from the approximated Pareto front of the population created with the MOEAD algorithm. This individual had 48 different transformations applied by JShadObf. Two different new populations have been created from some of this individual information, first a population containing 1000 individuals on which have been applied 48 randomly chosen transformations and a second population composed of 1000 individuals on which have been applied the same 48 transformations, in the same order. But as there is a part of randomness in each transformation, the results of this last population will be different from the selected individual, *e.g.* when applying the outlining transformation on two same individuals JShadObf might chose to outline one part of the program in the first case and another part of the program in the second case. Therefore the same sequence of transformations will give different individuals.

The results are presented in Figure 9.7 and show that the population composed of individuals on which 48 randomly chosen transformations have been applied are the farthest away from the approximated Pareto front. The second population created from the same transformations used in the selected individual came after, and finally the populations created with NSGA-II algorithm and the closest to the approximated Pareto front *i.e.* the population created with the MOEAD algorithm. This shows that the MOEAs perform better than the random selection of transformations and better as well than a preselected sequence of transformations that worked well on one individual, *i.e.* in the approximated Pareto front.

One can note as well that this preselected sequence perform better than the totally random sequence, therefore preforming the crossover using one transformation from one individual combined with the



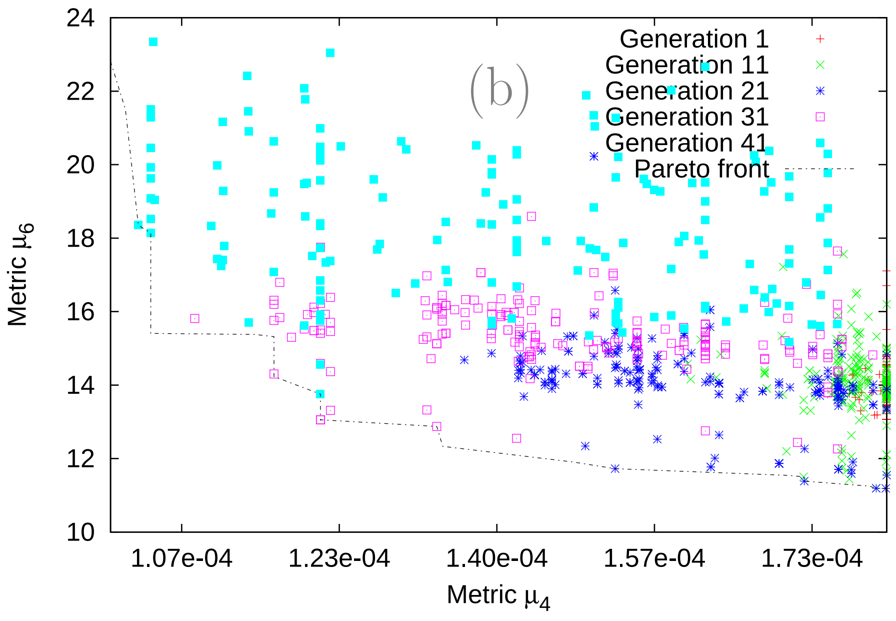


Figure 9.6: Experimental results obtained during the obfuscation of the `jquery.js` program by JShadObf. (a), (b): Set of 2D Pareto fronts approximation for the JQuery program using NSGA-II.

AST of another individual to generate a new offspring is a possibility that allows the population to evolve faster to the Pareto front.

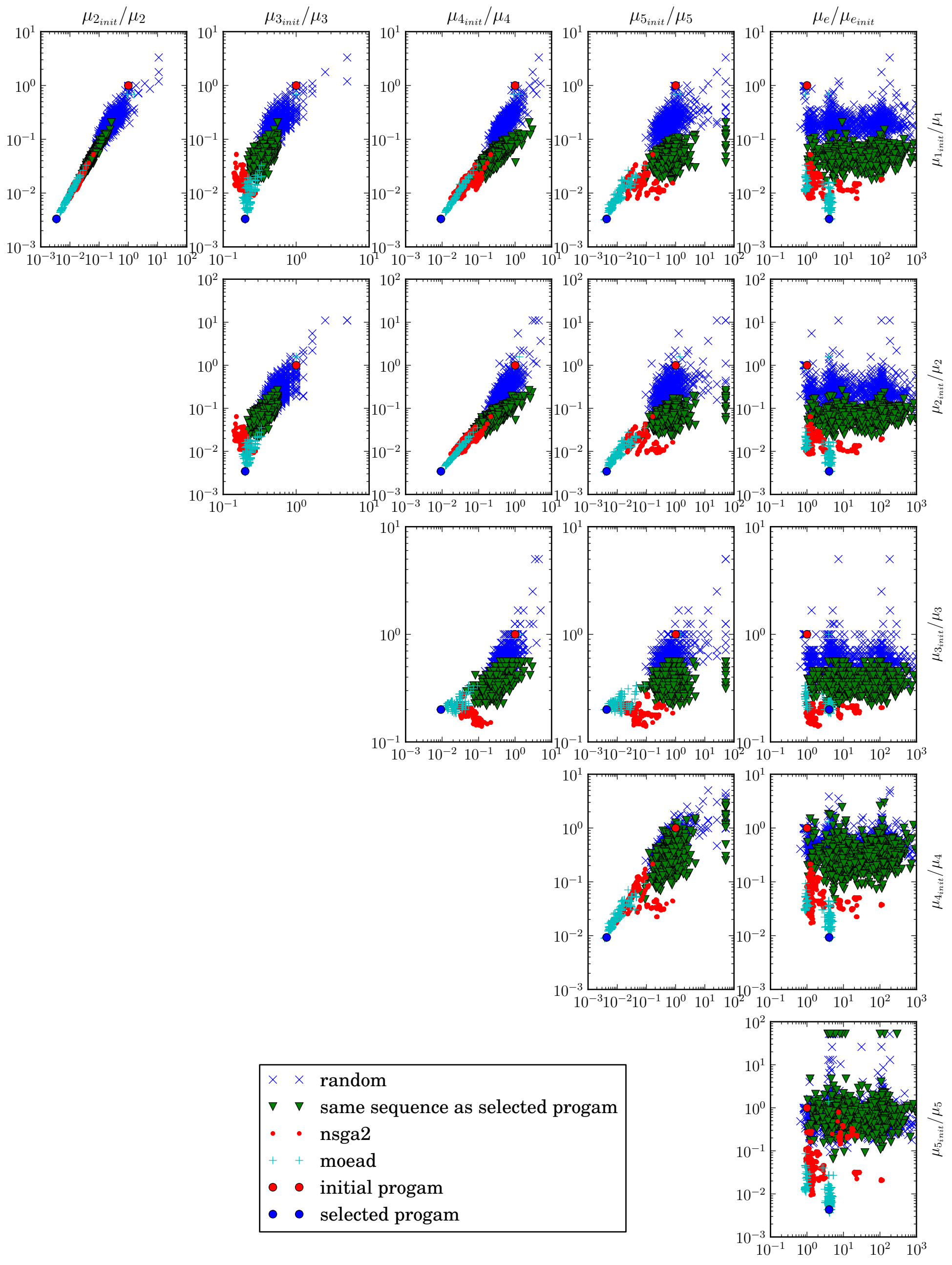


Figure 9.7: Set of 1000 individuals obtained for the obfuscation of a Matrix Multiplication program by JSHADOBF using NSGA-II and MOEAD and a random selection of transformations at the 40th generation.

9.5.4 Comparison MOEAD / NSGA-II

As shown by the Figure 9.8 the MOEAD has better results than NSGA-II, which means that the approximated Pareto front of MOEAD is closer to 0. MOEAD has as well a better repartition of its *individuals* contained in the Pareto front, when NSGA-II's Pareto front is less distributed. This means that the variety in the population is higher with MOEAD. MOEAD is indeed known to perform better in case of many objectives (more than 2) due to the more distributed repartition of its population provoked by the ponderation of the objectives using the weight vectors.

The table 9.2 shows the values of the metrics computed on *individuals* after application of different obfuscators/minifiers. This highlights the good result of JShadObf. We used two techniques to select *individuals* in the case of JShadObf, both of them are picking an individual in the approximated Pareto front. The first one (1 med) tries to find the *individual* which is the closest to the median values of every *individual* which is in the Pareto front, allowing to have a good trade off between all the objectives. The second one (2 `eucl_dist`) is taking the distance of the first 5 objectives to the vector null, then takes the n smallest one, and from this set, selects the fastest *individual*.

9.5.5 Summary of the Results

The table 9.2 summarised the results obtained with JShadObf and other obfuscators available in the literature.

The table 9.1 shows the coefficients of correlation of the different metrics computed for the 200 *individuals* of the 40 first generations on the `JQuery` program. The white cells (with numbers superior to 0.75) are reflecting highly correlated values. This shows that for the selected transformations the μ_1, μ_2 and μ_3 are correlated between themselves and the μ_4 and μ_5 are as well correlated between themselves. This however depends on the selected transformations and might not occur when selecting a different set of modifications. As an illustration of the obfuscation power of JShadobf, we proposed a set of selected output individuals for the `JQuery` library and `matmul` on the website of JShadObf[1].

	μ_1	μ_2	μ_3	μ_4	μ_5	μ_e
μ_1	1.0000000	0.9585020	0.8961660	0.5712278	0.3797580	-0.5521950
μ_2		1.0000000	0.8407987	0.4467864	0.2861682	-0.4422623
μ_3			1.0000000	0.5999287	0.3358829	-0.5806230
μ_4				1.0000000	0.7937583	-0.5900691
μ_5					1.0000000	-0.4642683
μ_e						1.0000000

Table 9.1: Correlation of the metrics for the 20 first generations of the `JQuery` program.

9.5.6 Comparison of JShadObf with other Obfuscators/Minifiers

We compared the results obtained with the JShadObf program with other existing obfuscators/minifiers. The comparison uses the metrics already shown in the Section 8.2. The results shown in table 9.2 reveal that JShadObf program performs better than the others on the selected metric. Indeed, JShadObf focuses on these metrics and uses EA to increase these values, whereas the others do not target these specific metrics. The transformations used by UglifyJS are not inserting dead code, but rather modifying the different expressions and even sometimes removing unreachable code. Closure Compiler is as well reducing the source code and not only the length but the number of expressions, indeed this is shown by the values in the two tables.

[1]http://jshadobf.gforge.uni.lu/

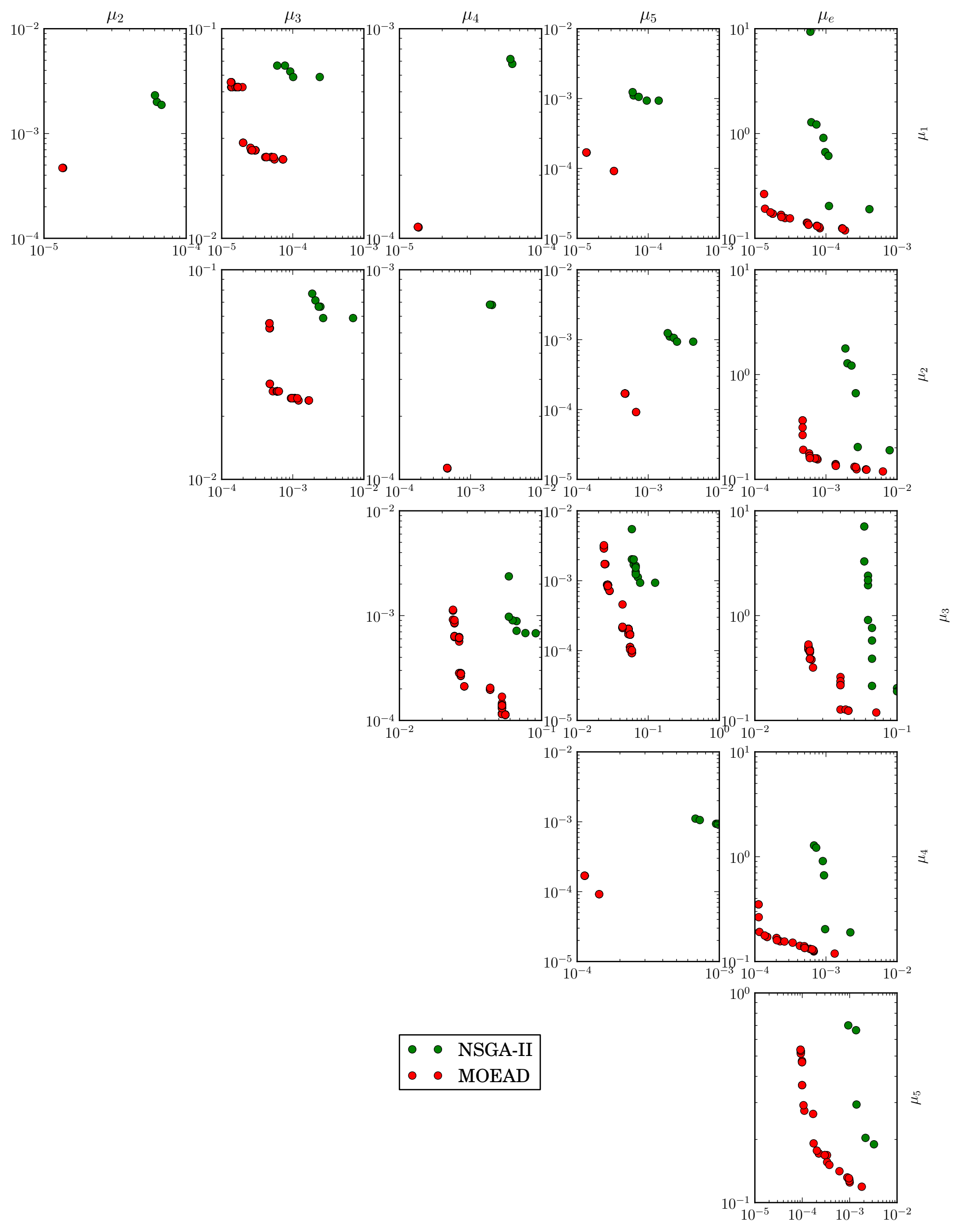


Figure 9.8: Set of 2D Pareto fronts approximation obtained for the obfuscation of a Matrix Multiplication program by JShadObf using NSGA-II (green) and MOEAD (red) at the 70th generation.

`matmul.js`	μ_1	μ_2	μ_3	μ_4	μ_5	μ_e/μ_e(**ref**)
Original sources	254	10	4	84	51	1
UglifyJS [20]	252	8	4	84	51	1.0024
YUI compressor [23]	254	10	4	84	51	1.0086
`javascriptobfuscator` [7]	318	10	4	97	35	1.6055
Closure Compiler [3]	243	8	4	81	51	0.992706
Jasob [6]	254	10	4	84	51	1.0052
ObfuscateJS [9]	254	10	4	84	51	1.0058
Packer [12]	127	4	3	40	91	1.7918
JShadObf NSGA-II (`med`)	3364	117	11	379	237	5.51
JShadObf NSGA-II (`eucl_dist`)	10865	334	16	1107	589	6.23
JShadObf MOEAD (`med`)	35017	1394	16	4049	2234	16.02
JShadObf MOEAD (`eucl_dist`)	13754	596	42	895	330	11.91
`jquery.js`	μ_1	μ_2	μ_3	μ_4	μ_5	μ_e/μ_e(**ref**)
Original sources	18169	673	9	5519	16511	1
YUI compressor [23]	–	–	–	–	–	not parsed
ObfuscateJS [9]	–	–	–	–	–	not parsed
`javascriptobfuscator` [7]	–	–	–	–	–	not parsed
UglifyJS [20]	17990	359	9	5318	18673	1.009
Jasob [6]	18005	669	10	5463	16405	1.0074
Packer [12]	127	4	3	40	91	1.008
Closure Compiler [3]	17677	374	8	5486	28051	1.000
JShadObf	88843	7030	15	7410	20735	1.013

Table 9.2: Summary of the obtained results. Green cells indicate the best results so far.

9.6 Conclusion

In this chapter, we have presented JShadObf, a source to source `JavaScript` obfuscator based on Multi-Objective Evolutionary Algorithms (MOEAs). Our proposed framework optimizes simultaneously six metrics – five evaluating the obfuscation capacity of the population being evolved and one quantifying the performance of each individual solution by measuring the execution time on a reference computing machine. Experimental results on two concrete JavaScript applications have been provided. JShadObf has been first tested on a simple pedagogical example, namely a matrix multiplication function where it outperformed the few existing other tools. We then decided to validate our approach on one of the most popular and widely used JavaScript library: `JQuery`. When some of the existing tools were not even able to parse successfully the library (Table 9.2), JShadObf managed to parse it completely and to generate the different obfuscated versions to effectively explore and measure the trade-off that might be selected among the six objectives analysed.

Of course, this work opens many perspectives. We are currently investigating the implementation of additional transformations (such as the loop unrolling, the inlining of functions, etc) but also other metrics. In particular, we are investigating ways to adapt the data structure complexity to the case of the JavaScript programming language, and also ways to take into account the string length and the number of calls to the `eval` function which most of the time decreases the readability of a program, like in [78] because the complexity added by packer [12] cannot be measured by the selected metrics.

9.6.1 Application to Other Languages

Similar method to obfuscate other languages such as `C` code has been studied [52] and resulted into a prototype named ShadObf using the Automatic Parallelizer and Code Transformation Framework (*Paralllisation Interprocdurale de Programme Scientifiques*) (PIPS) parallelizer [98] as a front-end to parse `C` code and to apply transformations. Being a source to source optimizer, PIPS already have many transformations that one can apply on the `C` code.

The main issue of the applications of JShadObf's method to other languages is the dependence of the transformations on the language. All the implemented transformations have to be modified and tested to fit the new language. A parser generating a usable parser tree of the program is as well needed to apply the transformations and to compute the different metrics.

However the MOEA implementations can be reused and are independent from the language selected.

Part IV

Conclusion

Chapter 10
Conclusions & Perspectives

Contents

This Chapter recalls the context of the dissertation, summarizes the different contributions developed during the thesis and details the challenges and perspectives opened by this work.

10.1 Summary

The CC paradigms, based on existing techniques and technologies, express the user's need to have available and flexible services while allowing companies to develop a pay-for-use model. As presented in Chapter 2, there are three main types of Clouds: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), all three targeting a different type of user. While the first one (IaaS) let a total control over the VM, the second one (PaaS) provides a more restricted access to the resource and the last one (SaaS) only allows the user to perform some limited operations. These three types of services have a different degree of resource pooling. Indeed in IaaS the CC provider often guarantees at least the access to a full CPU all the time, therefore the number of VMs running on a machine is bounded. In SaaS the provider has to guarantee a service rather than resources, letting the provider more freedom to pool the different resources and to optimise his infrastructure, procedures and programs. However, the externalisation of ICT services to CC providers requires a high level of security to convince users and organisations fearing, understandably, data losses and disclosure of information [59] to migrate their data and applications.

In this context, this PhD dissertation focuses on increasing security on CC platforms from the user's point of view by investigating and designing novel mechanisms to cover the following domains:

- **Integrity and confidentiality of Infrastructure-as-a-Service (IaaS) infrastructures** through the proposed CertiCloud framework, which is relying on a security based on hardware component, namely TPMs, and verified security protocols (see Part II) to provide guarantees on programs and data running in a virtualized environment, either before, during or after a deployment on the CC platform.

- **Software protection on Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS) architectures**, using the proposed JShadObf framework (see Part III).

This thesis weaves together multiple domains of computer science, starting from hardware component to parsing techniques, obfuscation techniques, cryptography, protocol verification, evolutionary algorithm, `JavaScript` and CC middlewares.

The main contribution of this thesis is the development of two novel frameworks: **CertiCloud** and **JShadObf** as well as their implementations.

10.1.1 Contribution to IaaS Platform Security (CertiCloud)

The passive hardware cryptographic co-processor, namely TPM, which is present on the motherboard of many computers and servers can be used not only to increase the security of the user's machines but also to certify distant servers to ensure their security.

CertiCloud was developed to increase the integrity of users' VMs in IaaS Cloud Computing platform and to provide a verification method of the cloud resource (see Chapter 7). Based on TPM hardware security developed in the Trusted Computing Group (TCG), the verifications are done using two specially developed protocols: TCRR and `VerifyMyVM`. While the first one asserts the integrity of a remote resource and permits to exchange a private symmetric key, the second authorizes the user to detect *trustfully* and *on demand* any tampering attempt on its running VM. These protocols being key components in the proposed framework, we take very seriously their analysis against known cryptanalytic attacks. This is testified by their successful validation by AVISPA and Scyther, two reference tools for the automatic verification of security protocols. CertiCloud relying on the above protocols, provides secure storage of users' environments and their safe deployment onto a virtualisation framework. To work, this protection needs the presence of a TPM on each resource of the CC provider.

An implementation based on Nimbus [126] has been created to validate the approach and to perform experiments on the detection of modifications within the VM. This implementation is described in Section 7.8 and proposes to modify lightly one of Nimbus' scripts to integrate a hook allowing CertiCloud to be launched on the nodes during the deployment of a VM. The implementation also includes servers to communicate between the user and the machine which has been selected by the middleware, *i.e.* Nimbus, to run the VM of the user. All these additions are almost independent of Nimbus to ease the integration of CertiCloud to other middlewares.

The TPM handling was done using the Java library TPM/J [141] allowing direct communication with the component and providing with all the available standard functions from the specifications [150].

Conclusion

In conclusion, an IaaS protection has been developed increasing the security of the virtual environment of the user and ensuring the safety of the host resource. The experiments using the CertiCloud implementation in the Nimbus framework proved experimentally the feasibility of the concepts behind CertiCloud and allowed a verification of the anomaly detection.

This work can be used by CC provider that wish to propose a higher degree of security to their clients. The user of these platforms augmented with CertiCloud will be granted with a verification capability of both their VM and the resource's provider machine. This work lead to publication in the following conferences [48], and journal [49].

10.1.2 Contribution to PaaS and SaaS Platforms Security (JShadObf)

With the democratisation of server-side `JavaScript` using `nodejs` [68], the obfuscation of the `JavaScript` code running on a PaaS or even on a SaaS Cloud Computing platforms allows to defend the code of the user from a reverse engineer, for a least some time.

JShadObf is a `JavaScript` Obfuscator measuring different aspects of a code such as execution time, size of the code and number of predicates to apply transformations, making the source code unintelligible (see Chapter 9). The first contribution, that has been developed for JShadObf, is a complete `JavaScript` ANTLR [132] grammar (see Section D.1) which is able to generate an AST for any `JavaScript` code. For instance, the widely used `JQuery` library which is used by numerous websites, can be parsed using this grammar.

The second contribution is the JShadObf framework itself, providing all the tools needed to execute the Evolutionary Algorithm implemented (MOEAD [109] and NSGA-II [71]) for the framework. The transformations presented in Chapter 9 have been implemented and tested multiple times using vast number of possible combinations of transformations applied on different `JavaScript` programs together with unittests. This procedure ensures that the transformations respect the main objective of obfuscation being that the observable behaviour of the running obfuscated program should be the same as the original program. The obfuscation criteria is based on five well-known metrics, coming from Software Engineering, to evaluate the opacity of the source code. Together with the time of execution, these metrics are optimized simultaneously thanks to the different Multi-Objective Evolutionary Algorithms (MOEAs) developed within JShadObf.

Finally JShadObf is able to produce many obfuscated versions of the same initial program with different values for the obfuscation metrics and the time of execution. The user only has to choose one or more versions depending on his requirements.

Many experiments have been done to validate the process and to compare the efficiency of the developed EA algorithms, showing that MOEAD performs better than NSGA-II in a many-objectives problem. They allow to verify that the composition of the developed transformations lead, most of the time, to valid obfuscated code.

Conclusion

In conclusion, a software protection has been developed to increase the security in SaaS and PaaS platforms. Prior to the transmission of their programs to the Cloud, the users will be able, using JShadObf, to obfuscate their program preventing the disclosure of information contained within the source code. JShadObf can also be used to obfuscate any `JavaScript` code distributed by a classical client/server web application, increasing the scope of this framework.

This work lead to publication in [50]. A `C` version of the obfuscator based on the same principles, developed using PIPS [98] source to source optimiser, modified to include the computation of the metrics and the EAs, has been studied and published in [51].

10.1.3 General Conclusion

This dissertation tries to tackle the problem of security in CC platforms from a user's point of view in order to increase user's confidence in the security of the Cloud. Users should not only believe that the resources they use are safe, they should also be able to verify it themselves. They should as well protect the information they are storing on these platforms.

In order to achieve these goals, two solutions have been studied to provide users of different types of Cloud with higher security. CertiCloud focuses on IaaS platform while JShadObf protect users' code to allow an execution on PaaS or SaaS platforms.

In case of an IaaS usage, both solutions can be used in parallel, indeed `JavaScript` language can be used to develop obfuscated server side applications with JShadObf, running on the user's VM deployed in a IaaS CC platform compatible with CertiCloud to ensure the user that neither the physical node nor his VM has been compromised.

10.2 Perspectives and future work

In this thesis two methodologies to increase the security have been explored and different tools to prove the feasibility of the developed concept have been implemented.

Concerning CertiCloud, a more advanced prototype considering the very promising middleware platform OpenStack [11] would allow to verify the scalability of the approach. OpenStack is as well one of the most promising open-source solutions for Cloud middleware, it gathers a huge number of well known companies for its development. It allows to drive not only open-source virtualisation software but also proprietary ones such as VMware [21] Virtual Machine Manager (VMM).

In this context new migration protocols need to be developed in order to tackle the migration of VMs in secure way while allowing the user to take part in the verification procedure. A more user-friendly interface to manage the CertiCloud framework is mandatory as the focus is on the user's point of view. This interface would allow the user to perform all the verification actions by himself (see Chapter 7), as well as other actions such as generation of VMIs, secure migrations of VMs, backup of VMs, etc.

The perspectives for JShadObf are as well numerous. The environment has been already developed for testing transformations and for applying them using MOEAs. It is now possible to imagine and implement new transformations including harder opaque predicates or encryption on the full source code or only part of it. New metrics to measure the opacity of these new transformations would as well be necessary. This would allow a better diversity in both the obfuscated code and measurement, leading to better obfuscated solutions.

Another possibility would be to develop a transformation encrypting parts or full programs using a key that would be transmitted during the execution of the program for on-the-fly decryption. This key would be sent only on the condition that the machine on which the VMs are running has been certified using CertiCloud. The exchange of the key would be protected using the session key exchanged during the CertiCloud's TCRR protocol.

Cloud Computing (CC) is a big step in the ICT domain, comparable to the advent of Internet. It is an easy, flexible and scalable way of providing services to users ranging from email inbox to VM dedicated to HPC. CC is still a recent concept and the increase in the number of research projects both from the industry and the academia suggests a big interest in making CC more flexible, reliable and secure. However, not all ICT problems can find a solution in the Cloud, especially if the requirements of the user do not fit the services offered by the Cloud, such as very high level of security. As the CDW [59] tracking poll shows, the main reason why organizations are reluctant to use Cloud Computing (CC) services is security concerns.

Therefore the main challenge for Cloud Computing (CC) providers is not only to increase the security of their infrastructures but also to grant the user mechanisms allowing him to verify the security of the platform. This is a crucial point, indeed the user should be able to check the state of the machines of the CC provider, for him to verify himself that the environment, where his programs are running, is indeed safe. If the CC services were more secure, part of the organizations reluctant to use them due to security concerns or legal issues would certainly step out and plunge into the CC paradigm.

References

[1] Avispa. http://avispa-project.org/.

[2] Bison - gnu parser generator. http://www.gnu.org/software/bison/.

[3] Closure compiler. https://developers.google.com/closure/compiler/.

[4] flex: The fast lexical analyzer. http://flex.sourceforge.net/.

[5] Gdb: The gnu project debugger. http://www.sourceware.org/gdb/.

[6] Jasob. http://www.jasob.com/ .

[7] javascriptobfuscator. http://www.javascriptobfuscator.com/.

[8] Nasa. http://www.nasa.gov/.

[9] Obfuscatejs. http://tools.2vi.nl/.

[10] Opennebula. http://www.opennebula.org/.

[11] Openstack. http://www.openstack.org/.

[12] packer. http://dean.edwards.name/packer/.

[13] Proverif: Cryptographic protocol verifier in the formal model. http://www.proverif.ens.fr/.

[14] Rackspace. http://www.rackspace.com/.

[15] Spidermonkey. https://developer.mozilla.org/en-US/docs/SpiderMonkey.

[16] coreboot. http://www.coreboot.org.

[17] GNU GRUB: a multiboot boot loader. http://www.gnu.org/software/grub/.

[18] grub-tcg. http://trousers.sourceforge.net/grub.html.

[19] Trusted Computing Group. www.trustedcomputinggroup.org.

[20] Ugligyjs. https://github.com/mishoo/UglifyJS.

[21] Vmware virtualization software. http://www.vmware.com/.

[22] Xen hypervisor, the powerful open source industry standard for virtualization. http://www.xen.org/.

[23] Yui compressor. http://developer.yahoo.com/yui/compressor/.

[24] Secure Hash Standard (SHS) SHA-1, SHA-256, SHA-384, and SHA-512. Federal Information Processing Standards (FIPS) Publication 180-2, National Bureau of Standards, Washington DC, 2002.

[25] A signature scheme for distributed executions based on macro-dataflow analysis. In *2nd Intl. Workshop on Remote Entrusting (Re-Trust 2009)*, Riva del Garda, Italy, Sept 2009.

[26] A signature scheme for distributed executions based on macro-dataflow analysis. In *Grande Region Security and Reliability Day (SecDay 2010)*, Saarbrcken, Germany, Mar. 18 2010.

[27] Open Cloud Computing Interface - OCCI. *Online: http://occi-wg.org/*, 2011.

[28] Tpm-based approaches to improve cloud security. In *Grande Region Security and Reliability Day (SecDay 2011)*, Trier, Germany, Mar. 25 2011.

[29] Using evolutionnary algorithms to obfuscate code. In *Evolve 2011*, Luxembourg, May 25-27 2011.

[30] Jquery. `http://www.jquery.org/`, 2012.

[31] Optimisation d'obfuscation de code source au moyen d'algorithmes évolutionnaires multi-objectifs. In *Proc. des 20 ème rencontres francophones du parallèlisme (Compas'13)*, Grenoble, France, January 1518 2013.

[32] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):1–40, 2009.

[33] D. G. Abraham, G. M. Dolan, G. P. Double, and J. V. Stevens. Transaction security system. *IBM Syst. J.*, 30(2):206–229, March 1991.

[34] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[35] Amazon. Amazon simple storage service.

[36] Amazon. Elastic load balancing. `http://aws.amazon.com/elasticloadbalancing/`.

[37] Amazon. Amazon elastic compute cloud, 2006–. `http://aws.amazon.com/ec2/`.

[38] Amazon. Auto scaling, 2006–. `http://aws.amazon.com/autoscaling/`.

[39] Andrew Appel. Deobfuscation is in np. *unpublished manuscsript, preprint available from http://www. cs. princeton. edu/˜ appel/papers/deobfus. pdf*, 2002.

[40] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[41] L. Buttyn M. Felegyhazi B. Bencsth, G. Pk. skywiper (a.k.a. flame a.k.a. flamer): A complex malware for targeted attacks, 2012.

[42] John Backus. Fortran: The ibm mathematical formula translating system. `http://www.fortran.com/`.

[43] JW Backus. The syntax and semantics of the proposed international algebraic language. information processing. In *International Conference on Information Processing, Paris, France*, 1959.

[44] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. Cloud computing synopsis and recommendations. *NIST special publication*, 800:146, 2012.

[45] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil P. Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In *Proceedings of the 21st Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '01, pages 1–18, London, UK, UK, 2001. Springer-Verlag.

[46] B. Bertholon, S. Varrette, and P. Bouvry. Certicloud: une plate-forme cloud iaas sécurisée. In *Proc. des 20ème rencontres francophones du parallélisme (RenPar'20)*, St Malo, France, May 10–13 2011.

[47] Benoît Bertholon, Christophe Cérin, Camille Coti, Jean-Christophe Dubacq, and Sébastien Varrette. Practical security in distributed systems. *Distributed Systems: Design and Algorithms*, pages 237–300.

[48] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Certicloud: a novel tpm-based approach to ensure cloud iaas security. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 121–130. IEEE, 2011.

[49] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Certicloud, une plate-forme cloud iaas sécurisée. *TSI. Technique et science informatiques*, 31(8-10):1121–1152, 2012.

[50] Benoît Bertholon, Sébastien Varrette, and Pascal Bouvry. Jshadobf: A javascript obfuscator based on multi-objective optimization algorithms. In *Network and System Security*, pages 336–349. Springer, 2013.

[51] Benoît Bertholon, Sebastien Varrette, and Sebastien Martinez. Shadobf: A c-source obfuscator based on multi-objective optimisation algorithms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, pages 435–444. IEEE Computer Society, 2013.

[52] Benoît Bertholon, Sebastien Varrette, and Sebastien Martinez. Shadobf: A c-source obfuscator based on multi-objective optimisation algorithms. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, pages 435–444, Washington, DC, USA, 2013. IEEE Computer Society.

[53] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak sponge function family main document. *Submission to NIST (Round 2)*, 3, 2009.

[54] Nir Bitansky and Ran Canetti. On strong simulation and composable point obfuscation. In *Advances in Cryptology–CRYPTO 2010*, pages 520–537. Springer, 2010.

[55] Norris Boyd et al. Rhino: Javascript for java. *Mozilla Foundation*, 2007.

[56] Hyun-Chul Jung Byung-Ik Kim, Chae-Tae Im. Suspicious malicious web site detection with strength analysis of a javascript obfuscation. *Intl. J. of Advanced Science and Technology*, 26, January 2011.

[57] Ran Canetti. Towards realizing random oracles: Hash functions that hide all partial information. In *Advances in CryptologyCRYPTO'97*, pages 455–469. Springer, 1997.

[58] Ran Canetti, Guy N Rothblum, and Mayank Varia. Obfuscation of hyperplane membership. In *Theory of Cryptography*, pages 72–89. Springer, 2010.

[59] CDW. Cdw 2011 cloud computing tracking poll, 2011.

[60] David Champagne and Ruby B Lee. Scalable architectural support for trusted software. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[61] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. 1994.

[62] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 1st edition, 2009.

[63] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report 148, July 1997. http://www.cs.auckland.ac.nz/∼collberg/Research/Publications/CollbergThomborsonLow97a/index.html.

[64] Christian Collberg, Clark Thomborson, and Douglas Low. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–196. ACM, 1998.

[65] Don Coppersmith. The data encryption standard (des) and its strength against attacks. *IBM journal of research and development*, 38(3):243–250, 1994.

[66] C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Ph.D., Eindhoven Univ. of Technology, 2006.

[67] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard.* Springer, 2002.

[68] Ryan Lienhart Dahl. nodejs: a platform built on chrome's javascript runtime for easily building fast, scalable network applications. `http://www.nodejs.org/`, 2009–2013.

[69] Chris I. Dalton, David Plaquin, Wolfgang Weidner, Dirk Kuhlmann, Boris Balacheff, and Richard Brown. Trusted virtual platforms: a key enabler for converged client devices. *SIGOPS Oper. Syst. Rev.*, 43(1):36–43, 2009.

[70] C. Darwin. *The Origin of Species.* John Murray, 1859.

[71] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist multiobjective genetic algorithm : Nsga-ii. *IEEE Transactions on Evolutionary Computation*, Vol 6, No 2, 2002.

[72] Mathieu Desnoyers and M Dagenais. Os tracing for hardware, driver and binary reverse engineering in linux. *CodeBreakers Journal*, 1(2), 2006.

[73] Whitfield Diffie and Martin E Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[74] Danny Dolev and Andrew C. Yao. On the security of public key protocols. In *22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, 1981.

[75] D. Eastlake and P. Jones. US Secure Hash Algorithm 1 (SHA1). Request for Comments (RFC) 3174, Network Working Group, November 2001.

[76] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *Information Theory, IEEE Transactions on*, 31(4):469–472, 1985.

[77] D Nurmi et al. and R Wolski, C Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. A technical report on an elastic utility computing architecture linking your programs to useful systems. Technical report, 08/2008 2008.

[78] Ben Feinstein and Daniel Peck. Caffeine monkey: Automated collection, detection and analysis of malicious javascript. In *DEFCON 15*, 2007.

[79] David Flanagan. *JavaScript: The Definitive Guide Activate Your Web Pages.* O'Reilly Media, Inc., 6th edition, 2011.

[80] Forbes. Cloud computing forecasts, Sept 2013. `http://www.forbes.com/sites/louiscolumbus/2013/11/16/roundup-of-cloud-computing-forecasts-update-2013/`.

[81] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International J. of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[82] Simson Garfinkel. *PGP: pretty good privacy.* O'reilly, 1995.

[83] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. *SIGOPS Oper. Syst. Rev.*, 37(5):193–206, October 2003.

[84] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC '09: Proceedings of the 41st annual ACM symposium on Theory of computing*, pages 169–178, New York, NY, USA, 2009. ACM.

[85] Shafi Goldwasser and Yael Tauman Kalai. On the impossibility of obfuscation with auxiliary input. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 553–562. IEEE, 2005.

[86] Michal Peeters Guido Bertoni, Joan Daemen and Gilles Van Assche. The keccak sponge function family, 2013. `http://keccak.noekeon.org/`.

[87] M. H. Halstead. Elements of software science. 1977.

[88] Warren A. Harrison and Kenneth I. Magel. A complexity measure based on nesting level. *SIGPLAN Notices*, 16(3):63-74, 1981.

[89] Kelly Heffner and Christian Collberg. The obfuscation executive. In *Information Security*, pages 428–440. Springer, 2004.

[90] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, Vol SE-7 , No 5, 1981.

[91] Grace Hopper. Cobol: Common business-oriented language.

[92] Google Inc. Gmail. `https://www.gmail.com/`.

[93] Google Inc. Google apps. `https://www.google.com/apps`.

[94] Google Inc. Google documents. `http://docs.google.com/`.

[95] Intel. Intel anti-theft technology. `http://www.intel.com/content/dam/doc/product-brief/mobile-computing-protect-laptops-and-data-with-intel-anti-theft-technology-brief.pdf`.

[96] E. C. M. A. International. *ECMA-262: ECMAScript Language Specification.* ECMA (European Association for Standardizing Information and Communication Systems), Geneva, Switzerland, third edition, December 1999.

[97] United Internet. 1 and 1, 1998–.

[98] Franois Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization. *ICS '91 Proceedings of the 5th international conference on Supercomputing*, -1, 1991.

[99] ISO, Geneva, Switzerland. *ISO/IEC 11889-1: Information technology – Trusted Platform Module – Part 1: Overview*, 2009.

[100] David Kahn. The codebreakers: The comprehensive history of secret communication from ancient times to the internet author: David kahn. 1996.

[101] Yuichiro Kanzaki, Akito Monden, Masahide Nakamura, and Ken-ichi Matsumoto. Exploiting self-modification mechanism for program protection. In *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 170–179. IEEE, 2003.

[102] Ramnath K.Chellappa. Intermediaries in cloud-computing: A new computing paradigm,, October 1997.

[103] Auguste Kerckhoffs. *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef.* Librairie militaire de L. Baudoin, 1883.

[104] Gene H. Kim and Eugene H. Spafford. The design and implementation of tripwire: a file system integrity checker. In *Proc. of the 2nd ACM Conference on Computer and communications security (CCS'94)*, pages 18–29. ACM, 1994.

[105] Henryk Klaba. Ovh: Solutions de tlphonie et dhbergement internet. `https://www.ovh.com/`.

[106] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. pages 388–397. Springer-Verlag, 1999.

[107] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, London, UK, UK, 1996. Springer-Verlag.

[108] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923, 1994.

[109] Hui Li and Qingfu Zhang. Multiobjective Optimization Problems With Complicated Pareto Sets, MOEA/D and NSGA-II. *IEEE Transactions on Evolutionary Computation*, 13(2):229–242, April 2009.

[110] J. Linn. Generic Security Service Application Program Interface – Version 2, Update 1. Request for Comments (RFC) 2743, Network Working Group, Janvier 2000.

[111] Gavin Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR. In *TACAS 96*, pages 147–166. Springer-Verlag, 1996.

[112] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1-2):53–84, 1998.

[113] Matias Madou, Patrick Moseley, Bjorn De Sutter, and Koen De Bosschere. Software protection through dynamic code mutation. In *in Proceedings of the 6th International Workshop on Information Security Applications (WISA*, pages 194–206, 2005.

[114] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[115] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software engineering*, Vol SE-2, No 4, 1976.

[116] John McCarthy. Lisp: List processing.

[117] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *Electronic Computers, IRE Transactions on*, EC-9(1):39–47, 1960.

[118] Catherine Meadows. Language generation and verification in the nrl protocol analyzer. In *Computer Security Foundations Workshop, 1996. Proceedings., 9th IEEE*, pages 48–61. IEEE, 1996.

[119] N Mehta and S Clowes. Shiva: Advances in elf binary encryption. *CanSecWest. http://cansecwest.com/core03/shiva. ppt*, 2003.

[120] Microsoft. BitLocker Drive Encryption. `http://technet.microsoft.com/en-us/library/cc732774.aspx`.

[121] Microsoft. Gold parsing system. `http://www.goldparser.org/`.

[122] Microsoft. Microsoft azure, 2010. `http://www.microsoft.com/windowsazure/`.

[123] Antonio Muñoz, Antonio Maña, and Daniel Serrano. Protecting agents from malicious hosts using tpm. *IJCSA*, 6(5):30–58, 2009.

[124] Peter Naur, John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, et al. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 6(1):1–17, 1963.

[125] Roger Needham and Michael Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12), December 1978.

[126] Nimbus. Nimbus project, 2006. `http://www.nimbusproject.org/`.

[127] NIST. Sha-3 competition, 2007–2012. `http://csrc.nist.gov/groups/ST/hash/sha-3/index.html`.

[128] Travis Ormandy. An empirical study into the security exposure to hosts of hostile virtualized environments. Technical report.

[129] Marek Ostaszewski. *On Capturing Vital Properties Of Denial Of Service Attacks Using Metaheuristic Approaches*. PhD thesis, 2010.

[130] E. I. Oviedo. Control flow, data flow, and program complexity. *Proceedings of IEEE COMPSAC*, pages 146–152, 1980.

[131] Vilfredo Pareto. Cours d\'economie politique. 1896.

[132] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL (k) parser generator. *Software-Practice and Experience*, 25(7):789–810, 1995.

[133] Jean-Christophe Courrge-Anne-Sophie Rivemale Jrme Quvremont Vincent Van der Leest Ilze Eichhorn Patrick Koeberl, Christian Wachsmann. New methodologies for security evaluation. Technical report.

[134] Diego Perez-Botero, Jakub Szefer, and Ruby B. Lee. Characterizing hypervisor vulnerabilities in cloud computing servers. In *Proceedings of the 2013 international workshop on Security in cloud computing*, Cloud Computing '13, pages 3–10, New York, NY, USA, 2013. ACM.

[135] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, E-SMART '01, pages 200–210, London, UK, UK, 2001. Springer-Verlag.

[136] Vincent Rijmen and Paulo S. L. M. Barreto. The Whirlpool hash function. online document, 2001.

[137] R. Rivest. The MD5 Message-Digest Algorithm. Request for Comments (RFC) 1321, Network Working Group, April 1992.

[138] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[139] Günter Rudolph. Convergence of evolutionary algorithms in general search spaces. In *Evolutionary Computation, 1996., Proceedings of IEEE International Conference on*, pages 50–54. IEEE, 1996.

[140] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Workshop on Hot Topic in Cloud Computing (HotCloud'09)*, June 2009.

[141] Luis Sarmenta. TPM/J Java-based API for the Trusted Platform Module. `http://projects.csail.mit.edu/tc/tpmj/`.

[142] Yu Sasaki, Yusuke Naito, Noboru Kunihiro, and Kazuo Ohta. Improved Collision Attack on MD5. Cryptology ePrint Archive, Report 2005/400, 2005.

[143] M Schallner. Beginners guide to basic linux anti anti debugging techniques. *CodeBreakers magazine, Security & Anti-Security–Attack & Defense*, 1(2), 2006.

[144] Sergei P. Skorobogatov. Semi-invasive attacks – a new approach to hardware security analysis, 2005.

[145] A. Stanik, M. Hovestadt, and Odej Kao. Hardware as a service (haas): The completion of the cloud stack. In *Computing Technology and Information Management (ICCM), 2012 8th International Conference on*, volume 2, pages 830–835, 2012.

[146] Mario Strasser. Software-based TPM Emulator. `http://tpm-emulator.berlios.de`.

[147] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[148] Christopher Tarnovsky. Attacking tpm part 2: A look at the st19wp18 tpm device, 2012.

[149] TCG. TCG Specification Architecture Overview – Revision 1.4. Technical report, TCG, 2007.

[150] TCG. Trusted Platform Module (TPM) Specifications, 2007. Online at `http://www.trustedcomputinggroup.org/resources/tpm_main_specification`.

[151] ARM Security Technology. Building a secure system using trustzone technology. Technical report, 2009.

[152] TESO. Burneye elf encryption program, 2002.

[153] Ken Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):419 – 422, 1968.

[154] Randy Torrance and Dick James. The state-of-the-art in ic reverse engineering. In *Cryptographic Hardware and Embedded Systems-CHES 2009*, pages 363–381. Springer, 2009.

[155] Michel Valdrighi. Wordpress, 2001–. `http://wordpress.org/`.

[156] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[157] Sébastien Varrette, Benoît Bertholon, and Pascal Bouvry. A signature scheme for distributed executions based on control flow analysis. In *Security and Intelligent Information Systems*, pages 85–102. Springer, 2012.

[158] John Viega. Cloud computing and the common man. *Computer*, 42:106–108, 2009.

[159] John Viega, Matt Messier, and Pravir Chandra. *Network Security with OpenSSL*. O'Reilly, 1st edition, June 2002.

[160] Z. Vrba, P. Halvorsen, and C. Griwodz. Program obfuscation by strong cryptography. In *Availability, Reliability, and Security, 2010. ARES '10 International Conference on*, pages 242–247, 2010.

[161] X. Wang, Y.L. Yin, and H. Yu. Finding Collisions in the Full SHA-1. In *Advances in Cryptology, Crypto05*, 2005.

[162] Lawrence C. Washington and Wade Trappe. *Introduction to Cryptography: With Coding Theory*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2002.

[163] Hoeteck Wee. On obfuscating point functions. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 523–532. ACM, 2005.

[164] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[165] Mark Zuckerberg. Facebook. `https://facebook.com/`.

Part V

Appendix

Acronyms

AES Advance Encryption Standard
AIK Attestation identity key
AJAX Asynchronous JavaScript and XML
ALU Arithmetic logic unit
AMI Amazon Machine Image
ANTLR ANother Tool for Language Recognition
API Application Programming Interface
AST Abstract Syntax Tree
BNF Backus-Naur Form
CA Certificate Authority
CBC Cipher Block Chaining
CC Cloud Computing
CFB Cipher FeedBack
CFG Context Free Grammar
CFL Context Free Language
CL-Atse Constraint-Logic-based Attack Searcher
CPU Central Process Unit
CRC Cyclic Redundancy Check
CRTM Core Root of Trust for Measurement
CSC Computer Science and Communications
CTR CounTeR
DDoS Distributed Denial of Service
DES Data Encryption Standard
DFA Deterministic finite automata
DFT Discrete Fourier Transform
DoS Denial of Service
EA Evolutionary Algorithm
EBNF Extended BNF
EBS Elastic Block Store
ECB Electronic CodeBook
ECMA European Computer Manufacturer's Association
EC Elliptic Curve
EK Endorsement Key
ET Expression Tree
FS File System
GA Genetic Algorithm
GEP Gene Expression Programming
GP Genetic Programming
GRUB GRand Unified Bootloader
HLPSL High Level Protocols Specification Language
HMAC Hash Message Authentication Code

HPC High Performance Computing
HaaS Hardware as a Service
ICT Information and Communications Technology
IDS Intrusion Detection System
IF Intermediate Format
IRP Integrity Reporting Protocol
IaaS Infrastructure-as-a-Service
JVM Java Virtual Machine
KVM Kernel-based Virtual Machine
LFSR Linear Feedback Shift Register
MBF Mean Best Fitness
MBR Master Boot Record
MD5 Message-Digest Algorithm 5
MITM Man In The Middle
MOEA Multi-Objective Evolutionary Algorithm
MPP Massively Parallel Processor
NFA Non-deterministic finite automata
NFS Network File System
NIST National Institute of Standards and Technology
NSA National Security Agency
OCCI Open Cloud Computing Interface
OFB Output FeedBack
OFMC On-the-fly Model-Checker
OO Object Oriented
OS Operating System
PCC Proof Carrying Code
PCR Platform Configuration Register
PGP Pretty Good Privacy
PIPS Automatic Parallelizer and Code Transformation Framework (*Paralllisation Interprocdurale de Programme Scientifiques*)
PKI Public-Key Infrastructure
PRNG [Pseudo]-Random Number Generator
PaaS Platform-as-a-Service
QoS Quality of Service
RAM Random Access Memory
RSA Rivest Shamir Adleman
RTM Root of Trust for Measurement
RTR Root of Trust for Reporting
RTS Root of Trust for Storage
SAT-MC SAT-based Model-Checker
SHA Secure Hash Algorithm
SLA Service Level Agreement
SRK Storage Root Key
SaaS Software-as-a-Service
TA4SP Tree Automata based Automatic Approximations for the Analysis of Security Protocols
TBB Trusted Building Blocks
TCCP Trusted Cloud Computing Platform

TCG Trusted Computing Group
TCRR TPM-based Certification of a Remote Resource
TC Trusted Computing
TOCTOU Time Of Check Time Of Use
TPM Trusted Platform Module
UL University of Luxembourg
VLAN Virtual Local Area Network
VMI Virtual Machine Image
VMM Virtual Machine Manager
VM Virtual Machine

Appendix A
Compilers and Parsing

Contents

The different manipulations of code performed by JShadObf in Chapter 9 require a tree based representation of the source code. Indeed working on a tree is easier to keep the coherency of the code and transformations *only* need to modify its structure to change the code, this relieve the obfuscation transformations of syntactical check.

A tree representation of the code is often generated by compilers and interpretors representing a fraction of their work called parsing techniques. This chapter will briefly presents these techniques which have been used for the generation of tree representation of `JavaScript` code used in JShadObf.

A.1 Compilers

In this section we will see first what is a language processor and more specifically how compilers – which are one type of language processor – work and can be used for software protection in addition to the hardware protections seen previously.

A.1.1 Languages Processors

A language processor is a program which is reading a source code and data to perform certain actions. In the case of a compiler the source code is read and translated into a program (Figure A.1). The created program can then be run with the data as input and generate the expected output.

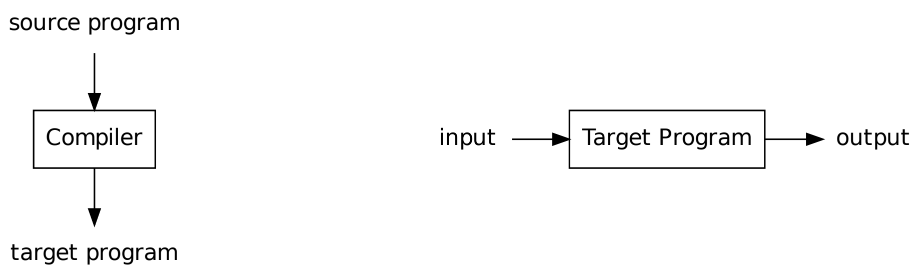


Figure A.1: Compilation and execution of the target program.

The other alternative is to interpret the source code. This is the job of an interpreter which takes as input the source code and the input data and produces the output by interpreting the source code as in Figure A.2.

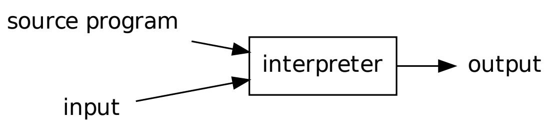


Figure A.2: Interpret the source code with the input data.

The combination of the two methods is as well possible, *e.g.* `Java` source code is compiled into bytecode which is a language close to the machine code, then the Java Virtual Machine (JVM) is interpreting the bytecode previously generated. This allows portable programs as the bytecode is not dependent on architecture.

A.1.2 Compiler Phases

The compilation of the source code of a program is a complex process, such that it has been divided into sub-problems to decrease the difficulty of such a task. The Figure A.3 extracted from [34] exposes the different phases of a program's compilation, which are now reviewed:

1. The lexical analyser takes a character stream, *i.e.* the source code and extract the different token (or lexemes) from it using regular expression (see Section A.3).

2. The syntax analyser generates a tree from the token stream (see Section A.4).

3. The semantic analyser is using the tree structure of the syntax tree to check the semantic consistency. In typed languages the type checking is part of the semantic analysis.

4. The intermediate code generator generates an intermediate language from the syntax tree. This language should have two important following properties: it should be easy to produce and to translate to machine code *e.g.* translating expressions such as:

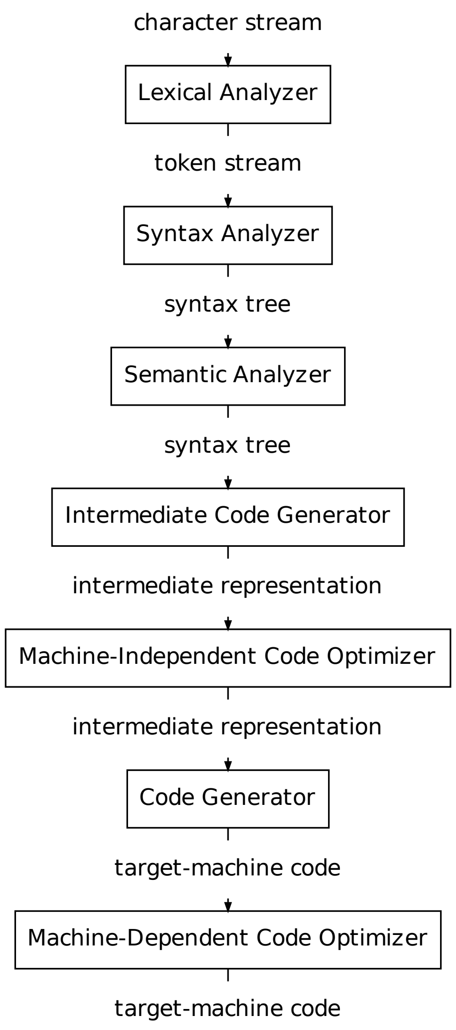


Figure A.3: Phases of a compiler.

```
                                           t1 = float(10)
id = a + b * float(10);        into        t2 = b * t1
                                           t3 = a + t2
                                           id = t3
```

5. The machine-independent code optimiser will optimise the intermediate language code by, for example, removing unused or unreachable parts of the code, removing intermediaries variables, calling functions which do not have side effects. The precedent code would be converted from:

```
t1 = float(10)
t2 = b * t1                    into        t2 = b * 10.0
t3 = a + t2                                id = a + t2
id = t3
```

6. The code generator which will translate the intermediate language into machine code.

7. The machine dependent code optimiser will then perform all the optimisation related to the machine architecture.

A.2 Parsing Techniques

Programming language is one of the most important basic block for computer scientist. It is used to describe the computation and data operation the computer has to perform. Usually a program source code is not written by the programmer directly into machine code, therefore not directly readable by a computer. Indeed machine code is not a human friendly language, it is assembly instructions and data given to the processor under the form of binary code, *i.e.* ones and zeros. In the early 1950's programmers used mnemonic representations of machine instructions instead of their equivalent in hexadecimal. This was followed by the use of macro instructions so that frequently used code would not need to be rewritten.

In the late 1950's, higher-level languages came with the development of Fortran [42], Cobol [91] and Lisp [116], but together with these languages, more advanced parsing and compiling techniques were needed to translate a source code from a human readable language to a machine readable format.

To parse languages both lexer (see Section A.3) and parser (see Section A.4) are used, as in Figure A.4, allowing to divide the work. The lexer is responsible for token recognition using simple techniques such as regular expressions, and the parser implements more advanced techniques based on grammar to generate a parse tree.

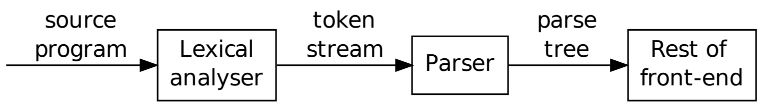


Figure A.4: Lexer and parser in compiler model.

A.3 Lexical Analysis

A source code can be seen as a character stream, for example considering the following instructions:

```
var a = 13;
 print(a);
```

They are read by the lexical analyser as a character stream of the following characters:
`’v’ ’a’ ’r’ ’ ’ ’a’ ’ ’ ’ ’ ’=’ ’ ’ ’1’ ’3’ ’;’ ’`
`n’ ’p’ ’r’ ’i’ ’n’ ’t’ ’(’ ’a’ ’)’ ’;’`

The role of the lexical analyser is to extract the lexemes and to produce as output a sequence of tokens. The following definitions (Definitions 18, 19 and 20) explain the basic vocabulary for lexical analysis.

Definition 18 (Token) *A token is the combination of a token name and an optional attribute. The token name is an abstract symbol representing a kind of lexical unit* e.g. *an* `Identifier`*, the optional attribute is often the lexeme itself.*

Definition 19 (Pattern) *A pattern is the description of the form that the lexemes of a token may take,* e.g. *in the case of a reserved word the pattern is the word itself.*

Definition 20 (Lexemes) *A lexeme is a sequence of characters that match a pattern for a token.*

An example of a *lexeme* would be the reserved word `var` matching the sequence of characters `'v'` `'a'` `'r'` corresponding to the *pattern* `var`. This would create a *token* <VAR, 'var'>. The pattern used to detect the lexemes is described by regular expressions (cf Section A.3.1) which is concise and expressive enough to represent the lexemes. The implementation of regular expressions in the lexer is using finite automata (cf Section A.3.2).

A.3.1 Regular expressions

A regular expression (*regexp* for short) is a sequence of characters describing a pattern that a text can match or not. Matching occurs when the text given in input is part of the set (finite or infinite) described by the regular expression.

For example the regular expression `letter_a` described by `a` is the set composed by { `a` }, *i.e.* only the text `a` can match this regular expression. The expression `(a|b)(c|d)` is matched by the four following texts: `ac`, `ad`, `bc`, `bd`.

The following table A.1 is showing the possible syntax for regular expressions.

Expression	Matches	Examples
c	the character c (if not an operator)	a
\\c	the character c (if an operator)	*
"s"	the string s	"*ab"
.	any character except new lines	.*
ˆ	beginning of the line	
$	end of the line	
[s]	any character present in s	[acijpsrtv]
[ˆs]	any character but the ones in s	[ˆaeiouy]
r*	match the regexp r any number of time (0 to ∞)	a*
r+	match the regexp r at least once (1 to ∞)	a+
r?	match the regexp r never or once (0 to 1)	a?
r{n,m}	match the regexp r between n and m times	a1,4
r_1r_2	the r_1 regexp followed by the r_2 regexp	ab
r_1—r_2	either the regexp r_1 or the regexp r_2	a—b
(r)	used to increase precedence level	(a—b)*c

Table A.1: Regular Expression Syntax.

The minus sign inside square brackets is used to avoid writing all the characters comprised between the two bound, *e.g.* `[a-j]` is matched by all the small case letters from `a` to `j`.

Using these rules, it is now possible to describe any lexeme definition encountered in the grammar file. For example the token `Identifier` is associated with the following regular expression:

```
[a-zA-Z_\$][a-zA-Z_\$0-9]*
```

A.3.2 Finite Automatas

Finite automatas can be divided into two categories: Non-deterministic finite automata (NFA) and Deterministic finite automata (DFA), both of which are capable of recognizing the regular language, *i.e.* the language used to define regular expressions. They can be represented as ordered graphs with conditional transitions on the edges, a start state (or initial state) and a set of accepting states (or final states). NFA are non-deterministic which means that from one state, multiple edges with the same conditional transition are possible. The Figure A.5 is representing the regular expression `a(a|b)+b` using the NFA representation.

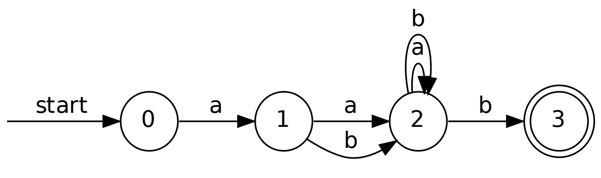


Figure A.5: NFA representation of `a(a|b)+b`.

To construct automatically the NFA from the regular expression, one can use the McNaughton-Yamada-Thompson algorithm described in [34] based on the work of [117] and [153].

The non-determinism is however a problem for the implementation, indeed you need to implement rollback in case the wrong choice has been made. For example, for the following text `abbb` an algorithm without rollback would first go in state 1 then in state 2, but with `b` as input it has two choices, either to go in state 3 or to stay in state 2. If it chooses to go in state 3 then as there is still a character `b` left and no more possible transitions in the NFA, it will result in a non-matching text which is false.

Hopefully every regular expression and every NFA can be converted into a DFA accepting the same regexp. And as the DFA simulation is simpler it does not need a rollback mechanism. The NFA in Figure A.5 is therefore translated into its DFA version in Figure A.6.

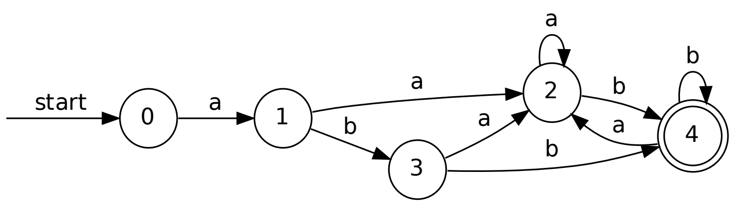


Figure A.6: DFA representation of `a(a|b)+b`.

In Figure A.6 the transitions are deterministic, therefore at every state and for any input there is only one possible path in the graph.

A.3.3 Lexical Analysis on a Concrete Example

To show what a lexer produces, the following source code has been given to the AntLR lexer [132] using the grammar Appendix D.1 developed in the context of this thesis.

```
var a = 13;
print(a);
```

It resulted in the token stream in Listing A.1 which will be given in input to the syntax analyser to produce a tree representation of the code (see Section A.4). The token names in Listing A.1 are sometimes not intuitive, for example, the string `var` is represent by `T__129`. This is due to the automatic generation of the lexer from the grammar. In the grammar these keywords are not given any token name, so the lexer generator chooses an unused name for the token, and as they are terminals, this name starts with the letter T.

```
<T__129,'var'>
<WhiteSpace,'␣'>
<Identifier,'a'>
<WhiteSpace,'␣'>
<WhiteSpace,'␣'>
<T__131,'='>
<WhiteSpace,'␣'>
<NumericLiteral,'13'>
<T__128,';'>
<LT,'\n'>
<Identifier,'print'>
<T__125,'('>
<Identifier,'a'>
<RPAREN,')'>
<T__128,';'>
<LT,'\n'>
<EOF,'None'>
```

Listing A.1: Token stream generated.

A.4 Parser

This section is devoted to parsing methods used in compilers. The programming languages have by design strict rules describing well formed source codes. These rules are defined in grammars which in certain cases can be used by automatic tools to produce lexer and parser programs. The grammar is often described using a specific form called the Backus-Naur Form (BNF) (defined by [43] and modified by [124]) or context-free grammar. This form gives an easy-to-understand specification of a language.

A.4.1 Notion of Context Free Grammars

Let's use the following grammar in Listing A.2 to understand a grammar syntax (this grammar can be found as well in Appendix C.1 written for AntLR [132]).

```
expression         ::= additionterm '+' expression
expression         ::= additionterm
additionterm       ::= soustractionterm '-' additionterm
additionterm       ::= soustractionterm
soustractionterm   ::= multiplicationterm '*' soustractionterm
soustractionterm   ::= multiplicationterm
multiplicationterm ::= divisionterm '/' multiplicationterm
multiplicationterm ::= divisionterm
divisionterm       ::= '(' expression ')'
divisionterm       ::= Decimalnumber // Decimalnumber is a terminal
```

Listing A.2: Simple grammar to parse numerical expressions.

Context free grammars are used to represent a language. It consists of terminals, non-terminals, a start symbol and productions.

- The terminals are the basic symbols, they are the name of the token given by the lexer, *e.g.* in Listing A.2, the `Decimalnumber`, and the symbols `+`, `-`, `*`, `/`, `(` and `)`.

- The non-terminals are syntactic variables that denote sets of strings, *e.g.* in Listing A.2, the *expression*, *additionterm*, *soustractionterm*, *multiplicationterm* and *divisionterm* are non-terminals.

- One of the non-terminals (conventionally the first non-terminal in the grammar) is distinguished as the start symbol, *e.g.* in Listing A.2 the *expression* non-terminal.

- The productions of a grammar specify how the terminals and the non-terminals are arranged to form non-terminals. On the left side of the definition the name of the non-terminal is specified, *e.g.* `divisionterm`, then the symbol `::=` separate the left and the right side, then the right side contains the rule to produce the non-terminal. This rule is composed by terminals, *e.g.* `(` and `)` and non-terminals, *e.g.* *expression.divisionterm.*

Most of the programming languages **cannot** be represented by context free grammar, indeed often source code cannot be taken out of its context, *e.g.* in `C++` the following code depends on the previously defined variables:

```
my_function < my_type > (arguments);
```

It can be seen either as a function call using a type as template with arguments or if `my_function`, `my_type` and `arguments` are three integers then it is two consecutive comparisons, *i.e.* the production rule depends of the types of `my_function`, `my_type` and `arguments`. As it is dependent on the context, it cannot be parsed by a context free grammar.

The `C` language is not context free either, indeed the `x * y;` sentence can be either a multiplication or the definition of the pointer `y` of type `x`. This cannot be parsed by a context free grammar and needs additional rules to be implemented in the parser.

`JavaScript` [96] do not has such context problems and can be parsed using a context free grammar even if it is not a Context Free Language (CFL). Indeed `JavaScript` has automatic semicolon insertion which performs the insertion of semicolon when it is not present at the end of a statement. This behaviour cannot be represented using a context free grammar. Such language specifications with standards using rules defined in English, make almost no computer languages CFL.

Order of precedence

One important notion to have in mind when designing a parser is the order of precedence. This is the order in which the operators are applied, *e.g.* in the expression `1+2*3` it is conventional to first compute the multiplication and then perform the addition, the multiplication has therefore an higher degree of precedence. The order of precedence is an intrinsic rule in a context free grammar, the closer the non-terminal is from the terminal, the higher the order of precedence it has. For example, in Listing A.2 the multiplication non-terminal is closer to the `Decimalnumber` terminal than the addition. This order can be bypassed using, for example, parenthesis as in Listing A.2, allowing an addition between parenthesis to be computed first.

Context free grammars *vs* regular expressions

Context free grammars are more powerful than regular expressions because they can describe more complex language. Actually the set of languages that can be described using regular expressions is a subset of the languages that can be described by context free grammars. Indeed any regular expression can be translated into a context free grammar, conversely all the context free grammars cannot be represented by regular expressions.

A.4.2 LL Parser vs LR Parser

There are two methods to parse a text using a grammar:

- LL(k) parsers: Left-to-right, Leftmost derivation. They are performing a **Top-Down** parsing which means they are trying to build the syntax tree from the start symbol to the terminal. Using this kind of parser implies that no left recursive or ambiguous grammar can be used. They are simpler because no backtracking is needed. They need however to recognise the productions strings using only the firsts k symbols. The leftmost derivation means that the parser tries to go down to the terminals, as the token stream is read from left to right, it derives first the leftmost non-terminal *e.g.* `additionterm`:

  `expression ::= additionterm '+' expression` *//leftmost derivation (additionterm is derived first)*

- LR(k) parsers: Left-to-right, Rightmost derivation. They are based on **Bottom-Up** parsing, which means that they start to build the parse tree from the leafs. They are more powerful than LL parser because the class of grammars that can be parsed using LR parser is a superset of the class of grammar that the LL parser can parse. The rightmost derivation is the consequence of the bottom up parsing, indeed the rules need to be reversed as the parser is trying to reconstruct the tree from the bottom and it reads the token stream from left to right:

  `expression ::= expression '+' additionterm`*//rightmost derivation*

Both parser types can be used to generate an AST or syntax tree.

A.4.3 Abstract Syntax Tree (AST)

The AST is the result of the parsing phase of a compiler. It represents the tokens given by the lexer using a tree model respecting all the rules described in the grammar.

For example, the parser created using the previously presented grammar Listing A.2 implemented in ANTLR in Appendix C.1 parses successfully the following expression:

```
(1+2)*3/(4+5*(6-7));
```

It produces, as a result, the tree presented in Figure A.7. In this picture one can clearly see the terminals which are on the leafs and the non-terminals which are above. However some terminals do not appear on the AST, because they have been consumed during the process and do not have any useful information left *e.g.* parenthesis are used to change the order of precedence, after the generation of the AST they are not used anymore.

A.5 ANTLR Parsing

ANother Tool for Language Recognition (ANTLR) [132] is a predicated LL(k) parser (Section A.4) written in Java, it is capable of generating parser in multiple languages such as `Java`, `C`, `C++` or `Python`. It combines the lexical and the syntactical definitions in a single file. Its development started in 1992 and is still widely used.

A.5.1 ANTLR Grammar

The ANTLR language grammar is based on Extended BNF (EBNF) for the production rules and have the following basic structure:

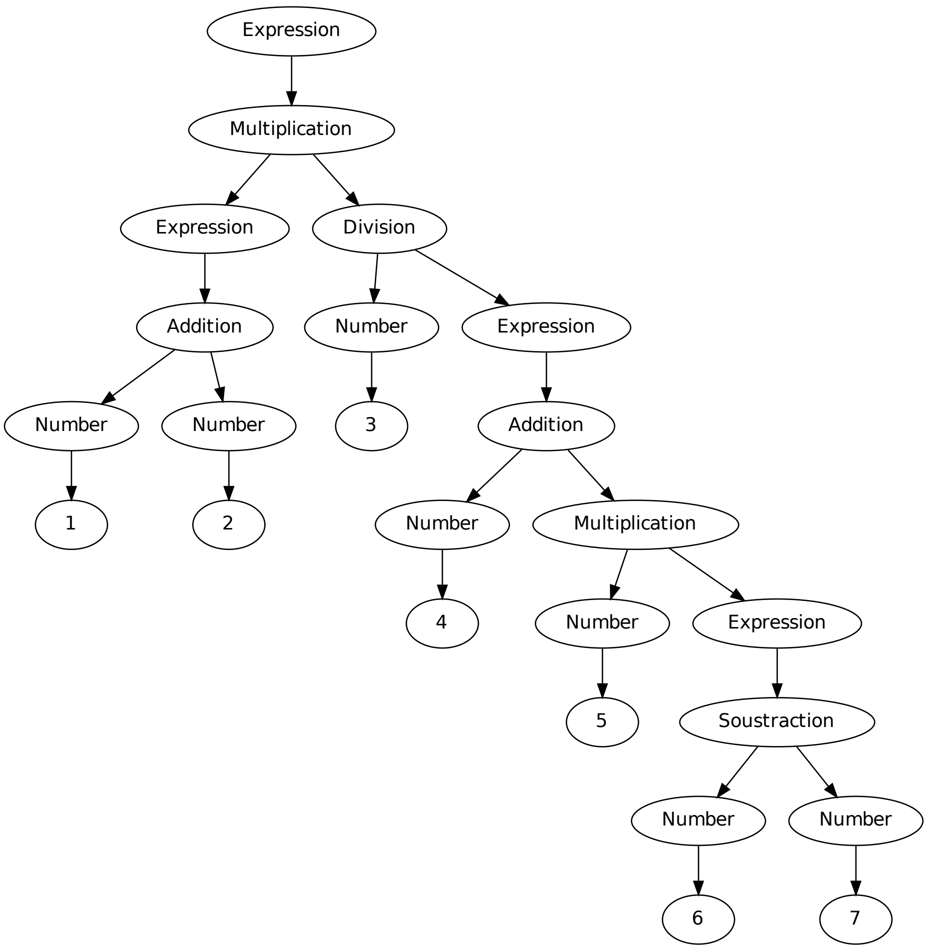


Figure A.7: AST for the expression: `(1+2)*3/(4+5*(6-7));`.

```
/* Comments */
grammar grammar_name;

options specifications
tokens specifications
attribute scopes
actions

rule1: production string 1
            | production string 2
            | ...
            ;

//lexical rules
Terminal1: regular expression
            | fragment1 (regular expression)?
            ;

fragment fragment1: regular expression ;
```

Listing A.3: Basic structure of ANTLR grammar file.

Specifications

The specifications part of the grammar can give some options to the ANTLR parser generator such as the language of the future parser, the expected output of the parser, *e.g.* AST, etc. The token specifications are used to create Terminals that will be used to construct the tree as in Listing C.1 where the type of operation is stored using tokens.

Actions and Scopes

The actions are source codes written in the language in which the parser will be generated and that will be directly included in the parser or in the lexer at different position depending on the scope contained in the action name *e.g.* `@header` will include the code following the action name in the beginning of the parser that will be generated. The action name starts with the symbol '@'.

Production Rules

The production rules contain the "valid sentences" of the language. They are composed of a rule name (that can be defined only once) and the possible alternative for a valid sentence. For example, `divisionterm` rule in Listing A.4 is either another expression within parenthesis or a Decimalnumber (which is a terminal).

```
divisionterm: '(' expression ')'
            | Decimalnumber
        ;
```

Listing A.4: `divisionterm` rule in ANTLR.

Lexical Rules

The lexical rules which are starting with a capital letter are used to specify terminals, they can use both regular expressions and fragment rules. The fragment rules allow to clarify the grammar. A fragment is only used to define more complex terminal rule, it does not produce any token.

A.5.2 ANTLR Tree Generator

One of the ANTLR output form is the AST output. This is possible by modifying the grammar and by specifying to ANTLR how to build the tree. At the end of each production string is specified how to build the tree, for example, the following production string would generate the `operator` at the root and the `expr1` and `expr2` at the leaves as in Listing A.5.

```
expr1 operator expr2 -> ^(operator expr1 expr2 ..)
```

Listing A.5: Tree generation in ANTLR.

A.5.3 ANTLR against other parsing software

Bison & Flex

The well known Bison [2] & Flex [4] (previously YACC & lex) software are very powerful LR parser and lexer. Bison is a Generalized LR parser *i.e.* a LR parser which is able to solve some conflicts by trying all the possible combinations until reaching inconstant state.

The format of the grammar is a machine readable BNF, plus some extra `C` definition and code.

GOLD

The GOLD parser generator [121] is a LALR(k) parser, which means k-Look Ahead LR parser. A LALR(k) parser is able to look k token ahead to take a decision about which production rule to choose in case of conflicts. This parser has the particularity of having a grammar written in pure BNF syntax. Having a pure BNF syntax is however a problem for the ambiguities of languages specifications such as in C the `typedef` problem mentioned in Section A.4.1 cannot be resolved. GOLD parser builder is available only for the Windows OS, but it is capable of generating parser in many different languages.

Why ANTLR?

ANTLR has been chosen for the following reasons:

- It is a widely used parser. For example, it is used by TWITTER to parse the search query, by NetBeans IDE to parses C++.
- It is multi-platforms because written in `Java`.
- It can generate AST easily using the AST output option.
- It supports a large variety of languages such as ActionScript, Ada95, `C`, `C++`, C#, `Java`, `JavaScript`, Objective-C, Perl, Python, Ruby.

A.6 JavaScript Parsing

This chapter describes briefly the required knowledge needed to understand parsing techniques. Parsing is divided into two sequential steps, the first one is the extraction of the tokens using the lexer. The second step is the generation of the AST which is performed by the parser.

The parser and the lexer are both produced by ANTLR using a grammar file to specify the syntactic rules of the language. A simple ANTLR example to parser simple arithmetic expression and to generate an AST has been explained.

Implementing a framework for the obfuscating a Javascript code assumed the correct and complete parsing of input codes such that one of the hidden contribution of this thesis was the definition of an accurate parser for the Javascript language in Appendix D.1.

It happened to be a non-trivial task such that it was necessary to develop our own ANTLR-based parser to be able to control every aspect of the generated tree.

The JavaScript code is parsed using the ANTLR grammar in Appendix D.1 developed with the help of existing `JavaScript` grammars available on the ANTLR website [1] and the ECMAScript Standardisation document [96] to generate AST. Thanks to this grammar, the ANTLR Parser is able to parse a file containing complex JavaScript source code and some JavaScript 1.8 "dialect" syntax. This grammar has been validated on the well known `JQuery` framework [30] using the tests suite developed by the developers of `JQuery`.

ANTLR generates the AST representation of a JavaScript source code. Source code of programs translated into an AST representation are easier to work with when dealing with any kind of modifications. Indeed, this representation allows the programming of a generic `tree_walker` used to compute the metrics but as well to perform the different transformations. This `tree_walker` is given in parameter to the function used depending on the desired action, either the computation of a metric or some modifications on the AST. The modifications performed on the AST do not have to deal with syntax formatting, *e.g.* the parenthesis are automatically added, the end-line character `;`, etc. Using AST, the addition, the modifications or the deletion of part of the tree are easier than the application

[1] http://antlr.org/

of transformation directly on the source code. This transformations are presented in Chapter 8 and their implementation in JSHADOBF in Chapter 9.

Appendix B
The Web Applications' Programming Language: `JavaScript`

Contents

The presentation of the `JavaScript` language's basis is presented in this chapter. The knowledge of the syntactical and lexical rules are necessary to understand the codes exposed in Chapter 9. This Chapter first depict the `JavaScript` language usages, then presents the rules that construct the language.

B.1 The `JavaScript` Programming Language

The following section presents the `JavaScript` language and a brief overview of its syntax and usage required to understand the different sources codes presented in Chapter 9 and, most importantly, the obfuscation framework we develop specifically for this language.

Quoting [79], `JavaScript` is the programming language of the Web. The overwhelming majority of modern websites use the `JavaScript` programming language and all modern web browsers – either on desktops, game consoles, tablets or smart phones – include `JavaScript` interpreters making it the most ubiquitous programming language in history. However `JavaScript` is not limited to client side only, it is as well possible to run `JavaScript` on the server side using a stand alone interpreter like `rhino` [55] or `nodejs` [68]. More concretely, `JavaScript` is a high-level, dynamic, untyped and interpreted programming language which is well-suited to object oriented and functional programming styles. `JavaScript` derives its syntax from Java, its first-class functions from Scheme, and its prototype-based inheritance from Self. Initially, many professional programmers denigrated the language for various reasons, ranging from design errors to buggy implementation in the first versions of the language. The standardization of the language within the European Computer Manufacturer's Association (ECMA) with the two version standards, ECMAScript 3 (1999) and 5 (2009), and, more importantly, the advent of Asynchronous JavaScript and XML (AJAX) returned `JavaScript` to the spotlight and brought more professional programming attention. It resulted in the proliferation of comprehensive frameworks and libraries, improved `JavaScript` programming practices, together with an increased usage of `JavaScript` outside of web browsers within server-side `JavaScript` platforms.

Generally speaking, `JavaScript` has long since outgrown its scripting-language roots to become a robust and efficient general-purpose language. The latest version of the language defines new features for serious large-scale software developments, which also explains the interest of all major vendors such as Microsoft or Google. In parallel, the recent explosion of novel web services that goes along with the early advent of the Cloud Computing (CC) paradigm increase the widespread adoption of `JavaScript` at the core of the development of these services. To cite a few well-known examples, one can mention Google Office Apps (featuring GMail or Google Docs), Dropbox (a popular web-based storage service – see `https://www.dropbox.com/`) or Doodle (a web-based scheduling service – see `http://www.doodle.com/`). Google is so deeply dependent on this language that they released their home-made development framework for `JavaScript` under the banner of the Closure Tools project [3]. Of interest for the work presented in this thesis, we can cite the Closure Compiler which compiles `JavaScript` into compact, high-performance code. This compiler removes dead code, then rewrites and minimizes what's left so that it downloads and runs quickly on the client's browser. It also checks syntax, variable references, and types, and warns about common `JavaScript` pitfalls. These checks and optimisation are meant to help writing applications that are less buggy and easier to maintain.

Let's see now what `JavaScript` looks like.

B.2 `JavaScript` Lexical Structure

The lexical structure of a language is the set of elementary rules used to write a program in this language. At this level, `JavaScript` is similar to the `C` programming language especially for the identifiers' definition, the comments and the statements delimiter. It is as well case sensitive which means that there is a distinction between small and capital letters. The style of comments is either `/*comments*/` or `// another comment until the newline`. The semi colon `;` is a delimiter between instructions, however it is not mandatory and can be replaced by a new line in unambiguous cases,however it is recommended for clarity and to avoid any misbehavior.

B.2.1 Identifiers

Identifiers are starting with a letter (small or big case), an underscore _ or a dollar sign `$` followed by letters, numbers, underscores or dollar signs. Some examples of identifiers are listed in B.1.

Listing B.1: Identifiers.

```
 identifier ;
$second_one;
_La$t_1;
```

However as in every programming language a list of reserved words listed in Appendix D.3 make some identifiers impossible to use as variables or functions.

B.2.2 Literals

The literals, *i.e.* data values appearing directly in the code, are either number literals, string literals, array literals, object literals, regular expression literals, `true`, `false`, and `null` like in listing B.2.

Listing B.2: Literals.

```
42 // number
4.2 // decimal number
"Hello World!!" // string
true
false
null // absence of object
{x : 0 , y : 0} // object literal
 [1,2,3]   // array literal
/foo/g // regular expression
```

B.3 Types, Values and Variables

`JavaScript` has two categories of types: *primitive types* and *object types*. The primitive types are numbers, strings, and boolean values, and are always copied, *e.g.* in `a = b = 2`, `a` and `b` contain a number which is `2`, and they are independent. Any other type is considered to be *object*, and they are referenced, *e.g.* in `c = d = [1,2,3]`, `c` and `d` are references to the same array and any modification of the array referenced by `c` will affected the array referenced by `d` as they are pointing to the same array.

B.3.1 Numbers

In `JavaScript` the numbers are encoded using the 64-bit floating point format defined by the IEEE 754 standard which is the same as the `double` type in `Java`, `C` and `C++`. `JavaScript` is then sensitive to rounding errors due to the representation of float numbers as binary fractions, *e.g.* `.3-.2 == .2-.1` return `false`. But unlike other languages there is no distinction between integer and floating-point values. Numbers can be represented in hexadecimal values using the prefix `0x` or octal value using `0`.

The typical operations, *i.e.* addition, subtraction, division, multiplication and modulo, are available using the standard operators `+, -, /, *, %`, and more advanced mathematical operations such as trigonometric or logarithm operations are available using the `Math` object.

B.3.2 Text and String Literals

In `JavaScript` strings are enclosed between two double or single quotes (`"` or `'`), they are a succession of UTF-16 characters. The symbol \ allows to escape the text and express special characters such as the new line character \`n`.

ECMAScript 5 is as well considering strings as read-only array, and allow access to it using square brackets *i.e.*, `[index]`.

B.3.3 Variables

`JavaScript` is an untyped language, *i.e.* no type declarations, a variable should be declared using the keyword `var`. Multiple declarations are possible and must be separated using coma as in listing B.3.

Listing B.3: Variable declarations in `JavaScript`.

```
var variable_1  = 3;
var number_1 = 3, string_2 = "hello", variable_3;
```

If a variable is assigned without being declared the interpretor will automatically declare it as a global variable which means that the code in listing B.4 will print `"Hello!"`.

Listing B.4: Implicit declarations in `JavaScript`.

```
function f(){
  hi = "Hello!"
}
f()
console.log(hi)
```

This however is not considered as good practice and leads to errors in ECMAScript 5 strict mode.

B.3.4 Scopes

The scope of a variable is the region of the code in which the variable is available. The main scope is the `global` scope, the variables declared in this scope are accessible anywhere in the code. Otherwise in `JavaScript` the scope of variables is at the function level unlike in `C` or `Java` where it is at the block level. A variable declared in a function will be in the `local` scope of the function. Functions can contain function definition with their own scope leading to multiple layers of scopes. The local version of a variable have always the priority, which means that if a variable is declared in the global and in the local scope, the local scope version will have the priority.

The result of the following code is then counter-intuitive for `C` or `Java` programmer where the scope of variable is at the block level, *i.e.* the curly brackets.

Listing B.5: Scopes in `JavaScript`.

```
var a = 3
function b(){
  console.log(a);
  var a = 4;
}
b()
```

This code is printing in the log console the "undefined" text. Indeed, functions are examined by the interpretor before execution for the discovery of the variable declarations anywhere in the function code (even in lower segment blocks). The function b contains the declaration of the variable `a`, at the beginning of the execution of the function `b`, the local variable `a` is then created and set to undefined. This is why the text `undefined` is returned. In `C` the equivalent code would print 3.

B.4 Statements and Expressions

Expressions and Statements are the basic block of a programming language. An expression is a piece of `JavaScript` code that can be evaluated to generate a value and a statement is a command that can be executed. This section is presenting the main expressions and statements present in the `JavaScript` language.

B.4.1 Expressions

Primary expressions

Primary Expressions are the "simplest" expressions, they are are constants, literal values, some reserved keywords or a variable references. The listing B.6 is presenting some examples of primary expressions.

Listing B.6: Primary Expressions in `JavaScript`.

```
3
"String"
true
null
this
```

Initializer expressions

Initializer expressions allow to generate objects or arrays with initial values. Array initializer is a coma separated list of expressions within square brackets and object initializer is a coma separated list of *properties* which are a collection of a key and a value within curly brackets. They can be called object and array literal as in listing B.2.

Assignment expressions

As in other programming languages assignement expressions are key expressions in the usage of `JavaScript`, they allow to store result of the evaluation of expressions into variables. The assignement expression `my_var = expr` is using the equal symbol `=` to assign the value of an expression *expr* to a variable *my_var*. This expression returns the value contained in the variable `my_var`.

Property access expressions

We have already seen how to create object and array literals, property access expressions are used to access values within these object and array. They are composed of the object (or most of the time its reference) followed by square brackets containing the index of the array or the name of the property as in listing B.16 and B.17. For object it is as well possible to use the dot operator "`.`" followed by the name of the property if the name of the property is an identifier.

Function definition expressions

Function definition expression is an expression composed of the keyword `function` followed by a list of arguments in parenthesis and a block of statements. In the listing B.7 the function expression is stored in the variable `my_function`.

Listing B.7: Funciton definition expression.

```
var my_function = function (arg_1,arg_2) {
  //code
} ;
```

Call expressions

Call expressions or invocation expressions are expressions representing the calling (executing) of a function. This is done using the name of the function or the expression pointing to the function followed by the coma separated list of arguments contained between parenthesis.

Listing B.8: Funciton call expression.

```
main()
Math.max(1,2,3)
```

Object creation expressions

The object creation expressions allow the creation of objects using the reserved keyword `new` followed by function responsible for the initialisation of the new object. For example, the expression `new Object()` will call the function `Object` responsible for creating a new object and will return the newly created object that can be stored using and assignment expression.

Arithmetic and comparison expressions

This expressions are the expressions using arithmetic and / or comparison operators such as `+, -, *, /, ==, !=, ===, <, <=, >, >=`. These operations have an order of precedence, *e.g.* `2 * 2 + 3`, `3 + 2 * 2` and `3 + (2 * 2)` all give `7` as the multiplication has a higher level of precedence than the addition. The order can be found in Appendix D.2. The parenthesis can be use to force the precedence order as the expression within the parenthesis has the highest order of precedence.

B.4.2 Statements

Expression Statements

An expression statement is simply a statement containing an expression. Any expression seen previously can be directly included in a statement.

Declaration Statements

The declaration statements are declaring variable using the keyword `var` as in listing B.3 or function with the keyword `function` followed by the function name, the coma separated list of arguments and the statement block as in listing B.9.

Listing B.9: Function declaration.

```
function function_name(arg1, arg2) {
        //function code.
        return true;
}
```

The return statement which is composed of the keyword `return` followed by an *expression* to return is often present in the code of function declarations. When a function is call within an expression the *expression* return by the return statement will be the value that the function call will be replaced by in the expression. This is one of the ways for a function to export the result of its computations, another way is to modify global variable or to modify the values of arguments given to the function in parameters as variables containing non-primitive object are only a reference to the object, a modification of the values contained in the object can allow the storage of computations performed by the function.

Conditional Statements

Conditional statements are mandatory to any programming language, they allow programs to perform actions depending on some conditions. As in many other languages, `JavaScript` has an `if/else` statement which syntax is exposed in the listing B.10.

Listing B.10: Conditional if/else branching.

```
// if/else statement
if (condition){
  // code
}
else{
  // code
}
```

The block statement following the if condition is executed if the condition evaluates to `true`, otherwise the block statement following the `else` keyword is executed. The "else" part of the branching, *i.e.* the `else` keyword and the corresponding statement block is not mandatory if no actions are needed in case of the evaluation of the condition to `false`.

Another common branching structure present in `JavaScript` is the `swich/case` structure shown in listing B.11. This structure is composed of the `switch` keyword followed by the expression to test, then a block statement containing the different possible values to compare with the initial expression using the keyword `case`. A default behaviour might be added using the keyword `default` in case the initial expression does not have an equivalent in the different cases.

Listing B.11: Conditional branching.

```
// switch statement
switch ( expression ) {
  case expression_1 :
    // code
    break;
  case expression_2 :
    // code
    break;
  default :
    // code
    break;
}
```

Loop Statements

The loop statements are statements containing loops. Loops are a way to execute the same block statement multiple times depending on a condition or on the size of an object or an array. The first kind of loop is the `for` loops. The classical `for` loop is the keyword `for` followed by , within parenthesis, an initializer, a condition and a expression executing after the block statement. Then a block statement containing the block of code to be executed within the loop (listing B.12). Another kind of `for` loop is the `for/in` loop which is browsing all the properties of an object and setting a variable at each round with the name of a property of the object (in listing B.12, the object `my_object`).

Listing B.12: For loops in `JavaScript`.

```
for(var a = 0 ; a < 10 ; a ++){
// code
}
var my_object = { 'h':'e','l':'l' , 'o':' ' , 'w':'o' , 'r':'l' , 'd':'!'}
for(a in my_object){
console.log(a)
console.log(o[a])
}
```

The `while` is executing a statement block while the condition given to the `while` loop is evaluated to `true` (listing B.13). A variant of the `while` loop is the `do/while` loop which is first executing one statement block and then re-executing it until the condition is `false`.

Listing B.13: While loops in `JavaScript`.

```
do{
// code
} while (condition)
while(condition){
// code
}
```

Special statements like `break` and `continue` statements allow to modify the behaviour of a loop. `break` statement allows for example to leave the loop without finishing the execution of the block statement and stop the execution of the loop. `continue` statement jumps directly at the end of the block statement, the loop might continue to execute if possible.

`try/catch/finally` Statements

When errors occur in `JavaScript` it is possible to avoid errors which lead to program termination using `try/catch/finally` statements (listing B.14). The statement is composed of the keyword `try` followed by the block of statements that might lead to an error, then the keyword `catch` followed by a variable which, in case of error, will contain error information and a block statement to be executed in case of error. Finally, using the keyword `finally` after the previous `try/catch` structure, it is possible to execute a block of statements after the `try catch` structure. This last statement block is always executed, even if the `try` statement block has a `return` statement, before returning to the caller the `finally` statement block will be executed. The `catch` and the `finally` parts of the `try` statement are optional.

Listing B.14: Try statements in `JavaScript`.

```
try{
// code
}
catch (error){
// catching error
}
finally{
// in all cases
}
```

B.5 Objects

`JavaScript` is a object oriented language which means everything is an object from numbers to string including function and of course objects defined by the programmer. The basic types are the different possible return values of the command `typeof`, which is either 'number', 'string', 'boolean', 'function', 'undefined' or 'object'.

The object creation is using object literals which are defined using curly brackets, and containing *properties* which are a collection of an identifier or number and a value which can be any expression.

Listing B.15: Object declaration.

```
var my_object = { name : value , other_name : other_value};
```

There are two ways to retrieve the value `value` corresponding to a name `name` contained in an object. The first one is to use the `dot` operator followed by the name `name` and the second one is to specify the name `"name"` as a string in square brackets.

Listing B.16: Object access.

```
console.log(my_object.name);
console.log(my_object["other_name"]);
```

An object in `JavaScript` is similar to dictionary or hashtable found in other languages, but with an additional `prototype` attribute which is inherited from another object and used to inherit properties.

Objects are dynamic which means that *properties*, *i.e.* methods and attributes can be added or removed dynamically to an instantiated object. In `JavaScript` Objects are passed around only by reference, which means that `var foo = bar;` will only create a new reference to the `bar` object and won't copy the data.

B.5.1 Arrays

An array is a collection of objects ordered using an index number which is a 32-bit integer value. It is an object which is inherited from the `Array` class. This class is providing the instantiated object with a set of methods and properties such as `length`, `push`, `pop`, `sort`, etc They are defined either using an array literal or creating an `Array` object as shown in listing B.17.

Listing B.17: List declaration and access.

```
var my_list = [ 1, 2, 3 ];
console.log(my_list [0]) ;
var second_list = new Array();
 second_list .push(0)
```

One feature of `JavaScript` arrays is that sparse arrays are available, *i.e.* arrays which are not complete and have some empty value between the bounds of the array.

B.6 Classes

There is not any reserved keyword to define classes in `JavaScript`, therefore a class can be created using different ways. The most similar way to `Java` is to use a function as a constructor and to add the methods of the class to the `prototype` propriety of the function. Indeed every object in `JavaScript` has a propriety named `prototype` which is used by the `new` operator to inherit the methods and objects from the constructor object. By calling the `new` keyword followed by the constructor function, the interpreter will create a new object and launch the function to construct the object. The newly created object will have all the methods defined in the `prototype` propriety of the constructor function. In listing B.18, `my_object` will inherit the `method1` from the constructor function `My_class`.

Listing B.18: Class definition.

```
function My_Class( init_arg ) {
  this.property1 = init_arg;
}
My_Class.prototype.method1 = function (arg) {
  return arg + this.property1
}
var my_object = new My_Class(10)
console.log(my_object.method1(10))
```

The `delete` function is unreferencing an object or a property which is afterwards removed by the garbage collector if not referenced by any other variable.

B.7 JavaScript & the Web

`JavaScript` language has the advantage of being easy to parse due to its simplicity. Indeed it is a scripting language with dynamic typing which means that the types are verified at run-time and not at parse-time therefore relieving the parser from checking types.

`JavaScript` is one of the most widely used languages animating web-sites with dynamic content and minimizing network communications between users and servers. It is an event-based language which means that it has been designed to answer to events such as the reception of a message from a server, an action of the user or the end of the page loading.

As it is a scripting language, the source code is directly distributed to the user, unlike in compiled languages where only the binary is shared. This aspect makes `JavaScript` source codes easy to reverse-engineer.

With the development of `JavaScript` on the client-side and the implementation of very fast interpretors within the browsers such as V8 from Google, `JavaScript` became more and more used on the server side as well. It is now possible to run `JavaScript` applications for the server using `rhino` [55] or `nodejs` [68]. Therefore it morphed from a web browser language to a full-fledged language.

Appendix C
Simple Expression Parser's Grammar

```
// grammar name
grammar simple_expr;

// options given to AntLR
options
{
     language=Python;
     output=AST;
     k=2;
     backtrack=true;
     memoize=true;
}

// token used for builing AST
tokens {
     Expression          ;
     Addition            ;
     Soustraction        ;
     Multiplication      ;
     Number              ;
     Division            ;
}

//nonterminals

//start symbol
statement:
     expression ';'  -> ^(Expression expression)
     ;

expression:
      additionterm '+' expression-> ^(Addition  additionterm expression)
     | additionterm
     ;

additionterm:
      soustractionterm '-' additionterm  -> ^(Soustraction soustractionterm additionterm )
     | soustractionterm
     ;

soustractionterm:
       multiplicationterm '*' soustractionterm
         -> ^(Multiplication multiplicationterm soustractionterm)
     |multiplicationterm
     ;

multiplicationterm:
       divisionterm '/' multiplicationterm  -> ^(Division divisionterm multiplicationterm )
     |  divisionterm
     ;
divisionterm:
      '(' expression ')'  -> ^(Expression expression)
     | Decimalnumber -> ^(Number Decimalnumber)
     ;


//Terminals
Decimalnumber:
      DecimalDigit * '.' DecimalDigit*
```

```
    | DecimalDigit
    ;
fragment DecimalDigit:
    ('0'..'9')
    ;

WhiteSpace:
    ('\n' | '\t' | '\v' | '\f' | '␣' | '\u00A0')    {$channel=HIDDEN;}
    ;
```

Listing C.1: Simple AntLR grammar example file.

Appendix D

JavaScript

D.1 JavaScript Grammar

```
/*
/[  Author:]/ Benoit Bertholon
*/

grammar JavaScript;

options
{
  language=Python;
    output=AST;
    k=2;
    backtrack=true;
    memoize=true;
}


tokens {
    Functioncall          ;    This                   ;    RegEx                  ;
    If                    ;    Regexchar              ;    Regexpid               ;
    Listarguments         ;    Var                    ;    Each                   ;
    Ident                 ;    Function               ;    While                  ;
    Program               ;    VarDeclaration         ;    Expr                   ;
    ListVarDeclaration    ;    Statements             ;    String                 ;
    Integer               ;    List                   ;    Assignment             ;
    Return                ;    Number                 ;    CaseClause             ;
    DoWhile               ;    Break                  ;    For                    ;
    Continue              ;    Switch                 ;    Throw                  ;
    DefaultBlock          ;    CaseBlock              ;    With                   ;
    Label                 ;    PropertyName           ;    Or                     ;
    And                   ;    Property               ;    New                    ;
    Ternary               ;    Index                  ;    Try                    ;
    BOr                   ;    BAnd                   ;    BXor                   ;
    MemberExpr            ;    Suffix                 ;    Null                   ;
    Ltrue                 ;    Lfalse                 ;    NaN                    ;
    EmptyStatement        ;    ExpressionStatement    ;    VariableDeclarationList ;
    VariableStatement     ;    UnaryExpr              ;    FunctionExpr           ;
    Delete                ;    CatchClause            ;    FinallyClause          ;
    Void                  ;    Typeof                 ;    ObjectLiteral          ;
    Yield                 ;    Let                    ;    ForIn                  ;
    ListCreation          ;    PropertyGet            ;    PropertySet            ;
    DebuggerStatement     ;
}


@lexer::members
{

    last = None
    def areRegularExpressionsEnabled(self):

        ret = True
        if self.last == None:
            ret =  True
        elif self.last.getType() == Identifier:
            ret =  False
        elif self.last.getType() == NumericLiteral:
            ret = False
        elif self.last.getType() == TNULL:
            ret = False
        elif self.last.getType() == TTRUE:
            ret = False
        elif self.last.getType() == TFALSE:
            ret = False
        elif self.last.getType() == TTHIS:
            ret = False
        elif self.last.getType() == DecimalLiteral:
            ret = False
        elif self.last.getType() == HexIntegerLiteral:
            ret = False
        elif self.last.getType() == StringLiteral:
            ret = False
        elif self.last.getType() == RBRACK:
            ret = False
        elif self.last.getType() == RPAREN:
            ret = False
        elif self.last.getType() == RSBRACK:
            ret = False
        else:
            ret = True;

        return ret

    def nextToken(self):
        self.result = Lexer.nextToken(self);
        if self.result.getChannel() != HIDDEN:
            self.last = self.result;
        return self.result;



}

RBRACK: '}';
RPAREN: ')';
RSBRACK: ']';

program:
     LT* sourceelements LT* EOF -> ^(Program sourceelements*)
    ;


sourceelements:
    sourceelement (LT!* sourceelement)*
    ;

sourceelement:
    functiondeclaration
  | statement ->  statement
    ;

// functions
functiondeclaration:
     'function' LT* Identifier  LT* formalparameterlist LT*  '{' LT* functionbody? '}'
       -> ^(Function  formalparameterlist ^(Statements functionbody?) ^(Ident Identifier))
    ;

functionexpression:
     'function' LT* Identifier? LT* formalparameterlist LT*  '{'  LT* functionbody? '}'
       -> ^(FunctionExpr  formalparameterlist ^(Statements functionbody?) ^(Ident Identifier)?)
    ;


formalparameterlist:
       '('  ')'   ->  ^(Listarguments)
    |  '(' (LT* Identifier (LT* ',' LT* Identifier)*)+ LT* ')'
        ->  ^(Listarguments ^(Ident Identifier)+)
    ;
```

```
functionbody:
       sourceelements LT!* ;

// statements
statement:
     statementBlock
    | variableStatement
    | functiondeclaration /* not compliant with ecma202 but in practice done by interpretors*/
    | emptyStatement
    | expressionStatement
    | ifStatement
    | iterationStatement
    | continueStatement
    | breakStatement
    | returnStatement
    | withStatement
    | labelledStatement
    | switchStatement
    | throwStatement
    | tryStatement
    | letStatement
    | yieldStatement
    | debbugerStatement
    ;

debbugerStatement:
    'debugger' (LT | ';')? -> ^(DebuggerStatement )
    ;

statementBlock:
    '{' LT* statementList? LT* '}'  -> ^(Statements statementList?)
    ;

statementList:
    statement (LT* statement)*
    ;

variableStatement:
    'var' LT* variableDeclarationList (LT | ';')* -> ^(Var variableDeclarationList)
    | 'let'  LT* variableDeclarationList (LT | ';')* -> ^(Let variableDeclarationList)
    ;

variableDeclarationList:
    v1=variableDeclaration (LT* ',' LT* variableDeclaration)* ->  variableDeclaration+
    ;

variableDeclarationListNoIn:
    v1=variableDeclarationNoIn (LT* ',' LT* variableDeclarationNoIn)*
        -> variableDeclarationNoIn+
    ;

variableDeclaration:
    Identifier LT* initialiser? -> ^(VarDeclaration ^(Ident Identifier) initialiser? )
    ;

variableDeclarationNoIn:
   Identifier LT* initialiserNoIn? -> ^(VarDeclaration ^(Ident Identifier) initialiserNoIn? )
    ;

initialiser:
   '=' LT* assignmentExpression -> ^(Expr assignmentExpression)
    ;

initialiserNoIn:
   '=' LT* assignmentExpressionNoIn -> ^(Expr assignmentExpressionNoIn)
    ;

emptyStatement: ';'  -> ^(EmptyStatement);

expressionStatement:
     expression (LT | ';')? -> expression
    ;




letStatement:
    'let'  LT* '('  variableDeclarationList  ')'  LT* statement? (LT | ';')?
         -> ^(Let variableDeclarationList statement?)
    ;

ifStatement:
    'if' LT* '(' LT* expression LT* ')' LT* s1=statement (LT* 'else' LT* s2=statement)?
        -> ^(If expression $s1? $s2?)
    ;

iterationStatement:
   doWhileStatement
    | whileStatement
    | forStatement
    | forInStatement
    ;

doWhileStatement:
   'do' LT* statement LT* 'while' LT* '(' expression ')' (LT | ';')?
         -> ^(DoWhile statement expression)
    ;

whileStatement:
   'while' LT* '(' LT* expression LT* ')' LT* statement -> ^(While expression statement)
    ;

forStatement:
   'for' LT* '(' ( LT* forStatementInitialiserPart)? LT*  ';' ( LT* e1=expression)? LT* ';'
        ( LT* e2=expression)? LT* ')' LT* statement -> ^(For  ^(Expr  forStatementInitialiserPart?)
           ^(Expr $e1?) ^(Expr $e2?) statement )
    ;

forStatementInitialiserPart:
   expressionNoIn
    | 'var' LT* variableDeclarationListNoIn -> ^(Var variableDeclarationListNoIn)
    ;

//EACH: 'each' ;

forInStatement:
 'for'  LT* Identifier? LT* '(' LT* forInStatementInitialiserPart LT* 'in' LT* expression
        LT* ')' LT* statement -> ^(ForIn  ^(Expr forInStatementInitialiserPart?)
            ^(Expr expression)  ^(Each Identifier?) statement)
    ;



forInStatementInitialiserPart:
   leftHandSideExpression
    | 'var' LT* variableDeclarationNoIn -> ^(Var variableDeclarationNoIn)
    | 'let' LT* variableDeclarationNoIn -> ^(Let variableDeclarationNoIn)
    ;

yieldStatement:
   'yield' expression (LT | ';')? -> ^(Yield expression)
    ;

continueStatement:
   'continue' Identifier? (LT | ';')? -> ^(Continue ^(Ident Identifier)?)
    ;

breakStatement:
   'break' Identifier? (LT | ';')? -> ^(Break ^(Ident Identifier)?)
    ;

returnStatement:
    'return' expression? (LT | ';')? -> ^(Return expression?)
     ;

withStatement:
    'with' LT* '(' LT* expression LT* ')' LT* statement -> ^(With expression statement)
     ;

labelledStatement:
    Identifier LT* ':' LT* statement -> ^(Label ^(Ident Identifier) statement)
     ;

switchStatement:
    'switch' LT* '(' LT* expression LT* ')' LT* caseBlock -> ^(Switch expression caseBlock)
     ;




caseBlock:
    '{'! (LT!* caseClause)* LT!* (LT!* defaultClause (LT!* caseClause)*)? LT!* '}'!
     ;


caseClause:
    'case' LT* expression LT* ':' LT* statementList? -> ^(CaseClause expression statementList?)
     ;

defaultClause:
    'default' LT* ':' LT* statementList? -> ^(DefaultBlock statementList?)
     ;

throwStatement:
    'throw' expression (LT | ';')? -> ^(Throw  expression )
     ;

tryStatement:
    'try' LT* statementBlock LT* (finallyClause
     |  (catchClause LT* catchClause* (LT* finallyClause)?) )
         -> ^(Try statementBlock catchClause* finallyClause?)
     ;

catchClause:
    'catch' LT* '(' LT* Identifier LT* ('if' expression) ?  LT* ')' LT* statementBlock
         -> ^(CatchClause ^(Ident Identifier) statementBlock expression?)
     ;

finallyClause:
    'finally' LT* statementBlock -> ^(FinallyClause  statementBlock)
     ;

// expressions
expression:
     (a=assignmentExpression->$a)
         (
             LT* ',' LT* b=assignmentExpression
             ->    ^(ExpressionStatement $expression $b)
         )*
     ;

expressionNoIn:
     (a=assignmentExpressionNoIn->$a)
         (
             LT* ',' LT* b=assignmentExpressionNoIn
             ->    ^(ExpressionStatement $expressionNoIn $b)
         )*
     ;




assignmentExpression:
    conditionalExpression
    | leftHandSideExpression LT* assignmentOperator LT* assignmentExpression
        -> ^(Assignment leftHandSideExpression assignmentOperator assignmentExpression)
    ;

assignmentExpressionNoIn:
   conditionalExpressionNoIn
    | leftHandSideExpression LT* assignmentOperator LT* assignmentExpressionNoIn
       -> ^(Assignment leftHandSideExpression assignmentOperator assignmentExpressionNoIn)
    ;

leftHandSideExpression:
    callExpression
    | newExpression
    ;
NEW: 'new';

newExpression:
   memberExpression
    |  NEW LT* newExpression -> ^(New newExpression)
    ;

pexp:
  primaryExpression                                -> ^(Expr primaryExpression)
  | functionexpression                             -> ^(Expr functionexpression)
  | 'new' LT* memberExpression LT* arguments       -> ^(New memberExpression arguments)

  ;


memberExpression:
   (p=pexp ->$p   )
   (
        (LT* memberExpressionSuffix)
        ->
          ^( MemberExpr $memberExpression  memberExpressionSuffix )
    )*
  ;



memberExpressionSuffix:
    indexSuffix
    | propertyReferenceSuffix
    ;

callExpression:
    memberExpression LT* arguments (LT* callExpressionSuffix)*
       ->  ^(Functioncall memberExpression arguments callExpressionSuffix*)
    ;

callExpressionSuffix:
    arguments
    | indexSuffix
    | propertyReferenceSuffix
    ;

arguments:
    '(' (LT* assignmentExpression (LT* ',' LT* assignmentExpression)*)* LT* ')'
       -> ^(Listarguments assignmentExpression* )
    ;

indexSuffix:
    '[' LT* expression LT* ']'  -> ^(Index expression)
    ;


propertyReferenceSuffix:
    '.' LT* Identifier -> ^(Property ^(Ident Identifier ))
    ;

assignmentOperator:
    '=' | '*=' | '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '>>>=' | '&=' | '^=' | '|='
    ;


conditionalExpression:
    (a=logicalORExpression ->^(Expr $a))
        (
            LT* '?' LT* a1=assignmentExpression LT* ':' LT* a2=assignmentExpression
            ->    ^(Ternary $conditionalExpression $a1 $a2)
        )?
    ;


conditionalExpressionNoIn:
    (a=logicalORExpressionNoIn-> ^(Expr $a))
        (
            LT* '?' LT* a1=assignmentExpressionNoIn LT* ':' LT* a2=assignmentExpressionNoIn
            ->    ^(Ternary $conditionalExpressionNoIn $a1 $a2)
        )?
    ;






logicalORExpression:
    (a=logicalANDExpression->$a)
        (
            LT* '||' LT* b=logicalANDExpression
            ->    ^(Expr $logicalORExpression Or $b)
        )*
    ;


logicalORExpressionNoIn:
    (a=logicalANDExpressionNoIn->$a)
        (
            LT* '||' LT* b=logicalANDExpressionNoIn
            ->    ^(Expr $logicalORExpressionNoIn Or $b)
        )*
    ;




logicalANDExpression:
    (a=bitwiseORExpression->$a)
        (
            LT* '&&' LT* b=bitwiseORExpression
            ->    ^(Expr $logicalANDExpression And $b)
        )*
    ;


logicalANDExpressionNoIn:
    (a=bitwiseORExpressionNoIn->$a)
        (
            LT* '&&' LT* b=bitwiseORExpressionNoIn
            ->    ^(Expr $logicalANDExpressionNoIn And $b)
        )*
    ;




bitwiseORExpression:
    (a=bitwiseXORExpression->$a)
        (
            LT* '|' LT* b=bitwiseXORExpression
            ->    ^(Expr $bitwiseORExpression BOr $b)
        )*
    ;


bitwiseORExpressionNoIn:
    (a=bitwiseXORExpressionNoIn->$a)
        (
            LT* '|' LT* b=bitwiseXORExpressionNoIn
            ->    ^(Expr $bitwiseORExpressionNoIn BOr $b)
        )*
    ;



bitwiseXORExpression:
    (a=bitwiseANDExpression->$a)
        (
            LT* '^' LT* b=bitwiseANDExpression
            ->    ^(Expr $bitwiseXORExpression BXor $b)
        )*
    ;


bitwiseXORExpressionNoIn:
    (a=bitwiseANDExpressionNoIn->$a)
        (
            LT* '^' LT* b=bitwiseANDExpressionNoIn
            ->    ^(Expr $bitwiseXORExpressionNoIn BXor $b)
        )*
    ;



bitwiseANDExpression:
    (a=equalityExpression->$a)
        (
            LT* '&' LT* b=equalityExpression
            ->    ^(Expr $bitwiseANDExpression BAnd $b)
        )*
    ;


bitwiseANDExpressionNoIn:
    (a=equalityExpressionNoIn->$a)
        (
            LT* '&' LT* b=equalityExpressionNoIn
            ->    ^(Expr $bitwiseANDExpressionNoIn BAnd $b)
        )*
    ;

EQUSING:    ('==' | '!=' | '===' | '!==') ;


equalityExpression:
    (a=relationalExpression->$a)
        (
            LT* EQUSING LT* b=relationalExpression
            ->    ^(Expr $equalityExpression EQUSING $b)
        )*
    ;


equalityExpressionNoIn:
    (a=relationalExpressionNoIn->$a)
        (
            LT* EQUSING LT* b=relationalExpressionNoIn
            ->    ^(Expr $equalityExpressionNoIn EQUSING $b)
        )*
    ;


COMPSIGNOIN: ('<' | '>' | '<=' | '>=' | 'instanceof' );
```

```
TIN: 'in';

compsig: COMPSIGNOIN | TIN;

relationalExpression:
    (a=shiftExpression->$a)
        (
            LT* compsig LT* b=shiftExpression
            ->    ^(Expr $relationalExpression compsig  $b)
        )*
    ;

relationalExpressionNoIn:
    (a=shiftExpression->$a)
        (
            LT* COMPSIGNOIN LT* b=shiftExpression
            ->    ^(Expr $relationalExpressionNoIn COMPSIGNOIN $b)
        )*
    ;


SHIFTSIG: ('<<' | '>>' | '>>>');
shiftExpression:
    (a=additiveExpression->$a)
        (
            LT* SHIFTSIG LT* b=additiveExpression
            ->    ^(Expr $shiftExpression SHIFTSIG $b)
        )*
    ;

additiveExpression:
    (a=multiplicativeExpression->$a)
    (   LT*  PLUSMINUS LT* b=multiplicativeExpression
        -> ^(Expr $additiveExpression PLUSMINUS $b) // use previous rule result
    )*
    ;

MULSIG:('*' | '/' | '%');

multiplicativeExpression:
    (a=unaryExpression->$a)
    (   LT* MULSIG LT* b=unaryExpression
        -> ^(Expr $multiplicativeExpression MULSIG  $b) // use previous rule result
    )*
    ;

PLUSMINUS: ('+' | '-');

INCDEC:  ('++' | '--');


DELETE:  'delete';
VOID: 'void' ;
TYPEOF: 'typeof';

fragment preunary:
      DELETE
    | VOID
    | TYPEOF
    | INCDEC
    | PLUSMINUS
    |   '~'
    | '!' ;

unaryExpression:
     postfixExpression
    | preunary LT* unaryExpression -> ^(UnaryExpr preunary unaryExpression )
    ;

postfixExpression:
     (l=leftHandSideExpression -> $l)
```

```
        (  INCDEC -> ^(Expr $postfixExpression  INCDEC))?
    ;

THIS: 'this';

primaryExpression:
    THIS -> This
    | '(' LT* expression LT* ')' -> expression
    | Identifier -> ^(Ident Identifier)
    | literal
    | arrayLiteral
    | objectLiteral
    | arrayLiteralCreation
    ;

// arrayLiteral definition.
arrayLiteral:
    '[' LT* assignmentExpression? (LT* ',' (LT* assignmentExpression)?)* LT* ']'
        -> ^(List assignmentExpression*)
    ;
arrayLiteralCreation:
    '[' LT* assignmentExpression LT* forInList LT* ']'
        -> ^(ListCreation assignmentExpression forInList)
    ;
forInList:
 'for' LT* Identifier? LT* '(' LT* forInStatementInitialiserPart LT* 'in'
     LT* expression LT* ')'
         -> ^(ForIn  ^(Expr forInStatementInitialiserPart?)
             ^(Expr expression) ^(Each Identifier?) )
    ;

// objectLiteral definition.
objectLiteral:
    '{' LT* propertyNameAndValue? (LT* ',' LT* propertyNameAndValue)* LT* '}'
        -> ^(ObjectLiteral propertyNameAndValue*)

    ;
propertyNameAndValue:
      Identifier LT* propertyName LT* '(' LT* ')' LT* '{' LT* functionbody '}'
        -> ^(PropertyGet Identifier propertyName functionbody)
    | a=Identifier LT* propertyName LT* '(' LT* b=Identifier  LT* ')'
        LT* '{' LT* functionbody '}'
            -> ^(PropertySet $a propertyName ^(Ident $b) functionbody)
    | propertyName LT* ':' LT* assignmentExpression
            -> ^(PropertyName propertyName assignmentExpression)

    ;


propertyName:
     Identifier -> ^(Ident Identifier)
    | StringLiteral -> ^(String StringLiteral)
    | NumericLiteral -> ^(Number NumericLiteral)
    ;

// primitive literal definition.
NULL: 'null';
TRUE: 'true';
FALSE: 'false';

literal:
      NULL -> ^(Null )
    | TRUE -> ^(Ltrue )
    | FALSE -> ^(Lfalse )
    | StringLiteral -> ^(String StringLiteral)
    | NumericLiteral -> ^(Number NumericLiteral)
    | RegularExpressionLiteral -> ^(RegEx RegularExpressionLiteral)

    ;

// regular expression
```

```
RegularExpressionLiteral:
      { self.areRegularExpressionsEnabled() }?=>  '/' RegularExpressionBody '/' '.'? IdentifierPart*
    ;
fragment RegularExpressionBody:
    RegularExpressionFirstChar RegularExpressionChar* ;

fragment RegularExpressionFirstChar:
     ~ ( LT | '*' | '\\' | '/' )
    | '\\' ~( LT )
    ;

fragment RegularExpressionChar:
     ~ ( LT | '\\' | '/' )
    | '\\' ~( LT )
    ;

// lexer rules.
StringLiteral:
     '"' DoubleStringCharacter* '"'
    | '\'' SingleStringCharacter* '\''
    ;

fragment DoubleStringCharacter:
     ~('"' | '\\' | LT)
    | '\\' EscapeSequence
    ;

fragment SingleStringCharacter:
     ~('\'' | '\\' | LT)
    | '\\' EscapeSequence
    ;

fragment EscapeSequence:
     CharacterEscapeSequence
    | '0'
    | HexEscapeSequence
    | UnicodeEscapeSequence
    ;

fragment CharacterEscapeSequence:
     SingleEscapeCharacter
    | NonEscapeCharacter
    ;

fragment NonEscapeCharacter:
     ~(EscapeCharacter | LT)
    ;

fragment SingleEscapeCharacter:
     '\'' | '"' | '\\' | 'b' | 'f' | 'n' | 'r' | 't' | 'v'
    ;

fragment EscapeCharacter:
    SingleEscapeCharacter
    | DecimalDigit
    | 'x'
    | 'u'
    ;

fragment HexEscapeSequence:
     'x' HexDigit HexDigit
    ;

fragment UnicodeEscapeSequence:
     'u' HexDigit HexDigit HexDigit HexDigit
    ;

NumericLiteral:
     DecimalLiteral
    | HexIntegerLiteral
    | 'NaN'
    ;

fragment HexIntegerLiteral:
     '0' ('x' | 'X') HexDigit+
    ;

fragment HexDigit:
     DecimalDigit | ('a'..'f') | ('A'..'F')
    ;

fragment DecimalLiteral:
     DecimalDigit+ '.' DecimalDigit* ExponentPart?
    | '.'? DecimalDigit+ ExponentPart?
    ;

fragment DecimalDigit:
     ('0'..'9')
    ;

fragment ExponentPart:
     ('e' | 'E') ('+' | '-') ? DecimalDigit+
    ;

Identifier:
     IdentifierStart IdentifierPart*
    ;

fragment IdentifierStart:
     UnicodeLetter
    | '$'
    | '_'
    | '\\' UnicodeEscapeSequence
    ;



fragment IdentifierPart:
     (IdentifierStart) => IdentifierStart
    | UnicodeDigit
    | UnicodeConnectorPunctuation
    ;


fragment UnicodeLetter:
    '\u0041'..'\u005A' | '\u0061'..'\u007A' | '\u00AA'  | '\u00B5'  | '\u00BA'
    | '\u00C0'..'\u00D6'    | '\u00D8'..'\u00F6'    |'\u00F8'..'\u021F' | '\u0222'..'\u0233'
    | '\u0250'..'\u02AD'    | '\u02B0'..'\u02B8'          | '\u02BB'..'\u02C1'
    |'\u02D0'..'\u02D1' | '\u02E0'..'\u02E4'    | '\u02EE'  | '\u037A'  | '\u0386'
    | '\u0388'..'\u038A'    | '\u038C'  |'\u038E'..'\u03A1'
    | '\u03A3'..'\u03CE'    | '\u03D0'..'\u03D7'    | '\u03DA'..'\u03F3'    | '\u0400'..'\u0481'
    |'\u048C'..'\u04C4' | '\u04C7'..'\u04C8'    | '\u04CB'..'\u04CC'    | '\u04D0'..'\u04F5'
    | '\u04F8'..'\u04F9'    |'\u0531'..'\u0556' | '\u0559'  | '\u0561'..'\u0587'
    | '\u05D0'..'\u05EA'    | '\u05F0'..'\u05F2'    |'\u0621'..'\u063A' | '\u0640'..'\u064A'
    | '\u0671'..'\u06D3'    | '\u06D5'  |'\u06E5'..'\u06E6' |'\u06FA'..'\u06FC' | '\u0710'
    | '\u0712'..'\u072C'    | '\u0780'..'\u07A5'    | '\u0905'..'\u0939'    | '\u093D'
    |'\u0950'   | '\u0958'..'\u0961'    | '\u0985'..'\u098C'    | '\u098F'..'\u0990'
    | '\u0993'..'\u09A8'    |'\u09AA'..'\u09B0' | '\u09B2'  | '\u09B6'..'\u09B9'
    | '\u09DC'..'\u09DD'    | '\u09DF'..'\u09E1'    |'\u09F0'..'\u09F1' | '\u0A05'..'\u0A0A'
    | '\u0A0F'..'\u0A10'    | '\u0A13'..'\u0A28'    | '\u0A2A'..'\u0A30'    |'\u0A32'..'\u0A33'
    | '\u0A35'..'\u0A36'    | '\u0A38'..'\u0A39'    | '\u0A59'..'\u0A5C'    | '\u0A5E'
    |'\u0A72'..'\u0A74' | '\u0A85'..'\u0A8B'    | '\u0A8D'  | '\u0A8F'..'\u0A91'
    | '\u0A93'..'\u0AA8'    | '\u0AAA'..'\u0AB0'    | '\u0AB2'..'\u0AB3'    | '\u0AB5'..'\u0AB9'
    | '\u0ABD'  | '\u0AD0'  | '\u0AE0'  | '\u0B05'..'\u0B0C'    |'\u0B0F'..'\u0B10'
    | '\u0B13'..'\u0B28'    | '\u0B2A'..'\u0B30'    | '\u0B32'..'\u0B33'
    | '\u0B36'..'\u0B39'    |'\u0B3D'   | '\u0B5C'..'\u0B5D'    | '\u0B5F'..'\u0B61'
    | '\u0B85'..'\u0B8A'    | '\u0B8E'..'\u0B90'    |'\u0B92'..'\u0B95' | '\u0B99'..'\u0B9A'
    | '\u0B9C'  | '\u0B9E'..'\u0B9F'    | '\u0BA3'..'\u0BA4'    |'\u0BA8'..'\u0BAA'
    | '\u0BAE'..'\u0BB5'    | '\u0BB7'..'\u0BB9'    | '\u0C05'..'\u0C0C'
    | '\u0C0E'..'\u0C10'    |'\u0C12'..'\u0C28' | '\u0C2A'..'\u0C33'
    | '\u0C35'..'\u0C39'    | '\u0C60'..'\u0C61'    | '\u0C85'..'\u0C8C'    |'\u0C8E'..'\u0C90'
    | '\u0C92'..'\u0CA8'    | '\u0CAA'..'\u0CB3'    | '\u0CB5'..'\u0CB9'    | '\u0CDE'
    |'\u0CE0'..'\u0CE1' | '\u0D05'..'\u0D0C'    | '\u0D0E'..'\u0D10'    | '\u0D12'..'\u0D28'
    | '\u0D2A'..'\u0D39'     |'\u0D60'..'\u0D61' | '\u0D85'..'\u0D96'     | '\u0D9A'..'\u0DB1'
    | '\u0DB3'..'\u0DBB'     | '\u0DBD'   |'\u0DC0'..'\u0DC6' | '\u0E01'..'\u0E30'
    | '\u0E32'..'\u0E33'     | '\u0E40'..'\u0E46'     | '\u0E81'..'\u0E82'     | '\u0E84'
    | '\u0E87'..'\u0E88'     | '\u0E8A'   | '\u0E8D'   | '\u0E94'..'\u0E97'
    | '\u0E99'..'\u0E9F'     | '\u0EA1'..'\u0EA3'     |'\u0EA5'    | '\u0EA7'   | '\u0EAA'..'\u0EAB'
    | '\u0EAD'..'\u0EB0'     | '\u0EB2'..'\u0EB3'     | '\u0EBD'..'\u0EC4'     | '\u0EC6'
    | '\u0EDC'..'\u0EDD'     | '\u0F00'   | '\u0F40'..'\u0F6A'     | '\u0F88'..'\u0F8B'
    | '\u1000'..'\u1021'     | '\u1023'..'\u1027'     | '\u1029'..'\u102A'     | '\u1050'..'\u1055'
    | '\u10A0'..'\u10C5'     | '\u10D0'..'\u10F6'     | '\u1100'..'\u1159'     | '\u115F'..'\u11A2'
    | '\u11A8'..'\u11F9'     | '\u1200'..'\u1206'     | '\u1208'..'\u1246'     | '\u1248'
    | '\u124A'..'\u124D'     | '\u1250'..'\u1256'     | '\u1258'   | '\u125A'..'\u125D'
    | '\u1260'..'\u1286'     | '\u1288'   | '\u128A'..'\u128D'     | '\u1290'..'\u12AE'
    | '\u12B0'   | '\u12B2'..'\u12B5'     | '\u12B8'..'\u12BE'     | '\u12C0'   | '\u12C2'..'\u12C5'
    | '\u12C8'..'\u12CE'     | '\u12D0'..'\u12D6'     | '\u12D8'..'\u12EE'     |'\u12F0'..'\u130E'
    | '\u1310'   | '\u1312'..'\u1315'     | '\u1318'..'\u131E'     | '\u1320'..'\u1346'
    |'\u1348'..'\u135A' | '\u13A0'..'\u13B0'     | '\u13B1'..'\u13F4'     | '\u1401'..'\u1676'
    | '\u1681'..'\u169A'     |'\u16A0'..'\u16EA' | '\u1780'..'\u17B3'     | '\u1820'..'\u1877'
    | '\u1880'..'\u18A8'     | '\u1E00'..'\u1E9B'     |'\u1EA0'..'\u1EE0' | '\u1EE1'..'\u1EF9'
    | '\u1F00'..'\u1F15'     | '\u1F18'..'\u1F1D'     | '\u1F20'..'\u1F39'     |'\u1F3A'..'\u1F45'
    | '\u1F48'..'\u1F4D'     | '\u1F50'..'\u1F57'     | '\u1F59'   | '\u1F5B'   | '\u1F5D'
    |'\u1F5F'..'\u1F7D' | '\u1F80'..'\u1FB4'     | '\u1FB6'..'\u1FBC'     | '\u1FBE'
    | '\u1FC2'..'\u1FC4'     | '\u1FC6'..'\u1FCC'     | '\u1FD0'..'\u1FD3'     | '\u1FD6'..'\u1FDB'
    | '\u1FE0'..'\u1FEC'     | '\u1FF2'..'\u1FF4'     | '\u1FF6'..'\u1FFC'     | '\u207F'
    | '\u2102'   | '\u2107'   | '\u210A'..'\u2113'     | '\u2115'   | '\u2119'..'\u211D'
    | '\u2124'   | '\u2126'   | '\u2128'   | '\u212A'..'\u212D'     | '\u212F'..'\u2131'
    | '\u2133'..'\u2139'     | '\u2160'..'\u2183'     | '\u3005'..'\u3007'
    | '\u3021'..'\u3029'     | '\u3031'..'\u3035'
    | '\u3038'..'\u303A'     | '\u3041'..'\u3094'     | '\u309D'..'\u309E'     | '\u30A1'..'\u30FA'
    | '\u30FC'..'\u30FE'     | '\u3105'..'\u312C'     | '\u3131'..'\u318E'     | '\u31A0'..'\u31B7'
    | '\u3400'  | '\u4DB5'   | '\u4E00'   | '\u9FA5'   | '\uA000'..'\uA48C'     | '\uAC00'   | '\uD7A3'
    | '\uF900'..'\uFA2D'     | '\uFB00'..'\uFB06'     | '\uFB13'..'\uFB17'     | '\uFB1D'
    | '\uFB1F'..'\uFB28'     | '\uFB2A'..'\uFB36'     |'\uFB38'..'\uFB3C' | '\uFB3E'
    | '\uFB40'..'\uFB41'     | '\uFB43'..'\uFB44'     | '\uFB46'..'\uFBB1'     |'\uFBD3'..'\uFD3D'
    | '\uFD50'..'\uFD8F'     | '\uFD92'..'\uFDC7'     | '\uFDF0'..'\uFDFB'     | '\uFE70'..'\uFE72'
    |'\uFE74'    | '\uFE76'..'\uFEFC'     | '\uFF21'..'\uFF3A'     | '\uFF41'..'\uFF5A'
    | '\uFF66'..'\uFFBE'     | '\uFFC2'..'\uFFC7'     | '\uFFCA'..'\uFFCF'     | '\uFFD2'..'\uFFD7'
    | '\uFFDA'..'\uFFDC'     ;

fragment UnicodeCombiningMark:
    '\u0300'..'\u034E'  | '\u0360'..'\u0362'     | '\u0483'..'\u0486'     | '\u0591'..'\u05A1'
    | '\u05A3'..'\u05B9'     | '\u05BB'..'\u05BD'     | '\u05BF'   | '\u05C1'..'\u05C2'
    | '\u05C4'   | '\u064B'..'\u0655'     | '\u0670'   | '\u06D6'..'\u06DC'     |'\u06DF'..'\u06E4'
    | '\u06E7'..'\u06E8'     | '\u06EA'..'\u06ED'     | '\u0711'   | '\u0730'..'\u074A'
    |'\u07A6'..'\u07B0' | '\u0901'..'\u0903'     | '\u093C'   | '\u093E'..'\u094D'
    | '\u0951'..'\u0954'     |'\u0962'..'\u0963' | '\u0981'..'\u0983'     | '\u09BC'..'\u09C4'
    | '\u09C7'..'\u09C8'     | '\u09CB'..'\u09CD'     |'\u09D7'    | '\u09E2'..'\u09E3'
    | '\u0A02'   | '\u0A3C'   | '\u0A3E'..'\u0A42'     | '\u0A47'..'\u0A48'     |'\u0A4B'..'\u0A4D'
    | '\u0A70'..'\u0A71'     | '\u0A81'..'\u0A83'     | '\u0ABC'   | '\u0ABE'..'\u0AC5'
    |'\u0AC7'..'\u0AC9' | '\u0ACB'..'\u0ACD'     | '\u0B01'..'\u0B03'     | '\u0B3C'
    | '\u0B3E'..'\u0B43'     |'\u0B47'..'\u0B48' | '\u0B4B'..'\u0B4D'     | '\u0B56'..'\u0B57'
    | '\u0B82'..'\u0B83'     | '\u0BBE'..'\u0BC2'     |'\u0BC6'..'\u0BC8' | '\u0BCA'..'\u0BCD'
    | '\u0BD7'   | '\u0C01'..'\u0C03'     | '\u0C3E'..'\u0C44'     |'\u0C46'..'\u0C48'
    | '\u0C4A'..'\u0C4D'     | '\u0C55'..'\u0C56'     | '\u0C82'..'\u0C83'     | '\u0CBE'..'\u0CC4'
    |'\u0CC6'..'\u0CC8' | '\u0CCA'..'\u0CCD'     | '\u0CD5'..'\u0CD6'     | '\u0D02'..'\u0D03'
    | '\u0D3E'..'\u0D43'     |'\u0D46'..'\u0D48' | '\u0D4A'..'\u0D4D'     | '\u0D57'
    | '\u0D82'..'\u0D83'     | '\u0DCA'   | '\u0DCF'..'\u0DD4'     |'\u0DD6'    | '\u0DD8'..'\u0DDF'
    | '\u0DF2'..'\u0DF3'     | '\u0E31'   | '\u0E34'..'\u0E3A'     | '\u0E47'..'\u0E4E'
    | '\u0EB1'   | '\u0EB4'..'\u0EB9'     | '\u0EBB'..'\u0EBC'     | '\u0EC8'..'\u0ECD'
    | '\u0F18'..'\u0F19'     | '\u0F35'   | '\u0F37'   | '\u0F39'   | '\u0F3E'..'\u0F3F'
    | '\u0F71'..'\u0F84'| '\u0F86'..'\u0F87'     | '\u0F90'..'\u0F97'     |'\u0F99'..'\u0FBC'
    | '\u0FC6'   | '\u102C'..'\u1032'     | '\u1036'..'\u1039'     | '\u1056'..'\u1059'
    |'\u17B4'..'\u17D3' | '\u18A9'   | '\u20D0'..'\u20DC'     | '\u20E1'   | '\u302A'..'\u302F'
    | '\u3099'..'\u309A'     |'\uFB1E'   | '\uFE20'..'\uFE23'     ;

fragment UnicodeDigit:
    '\u0030'..'\u0039'  | '\u0660'..'\u0669'     | '\u06F0'..'\u06F9'     | '\u0966'..'\u096F'
    | '\u09E6'..'\u09EF'     | '\u0A66'..'\u0A6F'     | '\u0AE6'..'\u0AEF'     | '\u0B66'..'\u0B6F'
    | '\u0BE7'..'\u0BEF'     | '\u0C66'..'\u0C6F'     | '\u0CE6'..'\u0CEF'     | '\u0D66'..'\u0D6F'
    | '\u0E50'..'\u0E59'     | '\u0ED0'..'\u0ED9'     | '\u0F20'..'\u0F29'     | '\u1040'..'\u1049'
    | '\u1369'..'\u1371'     | '\u17E0'..'\u17E9'     | '\u1810'..'\u1819'     | '\uFF10'..'\uFF19';

fragment UnicodeConnectorPunctuation:
    '\u005F'    | '\u203F'..'\u2040'    | '\u30FB'  | '\uFE33'..'\uFE34'    | '\uFE4D'..'\uFE4F'
    | '\uFF3F'  | '\uFF65'  ;

Comment:
    '/*' (options {greedy=false;} : .)* '*/' {$channel=HIDDEN;}
    ;

LineComment:
    '//' ~(LT)* {$channel=HIDDEN;}
    ;

LT:
    '\n'        // Line feed.
    | '\r'      // Carriage return.
    | '\u2028'  // Line separator.
    | '\u2029'  // Paragraph separator.
    ;

WhiteSpace:
    ('\t' | '\v' | '\f' | '␣' | '\u00A0')   {$channel=HIDDEN;}
    ;
```

Listing D.1: JavaScript AntLR grammar file.

D.2 Precedence Rules

Operator	Operation
++	Pre- or post-increment
–	Pre- or post-decrement
-	Negate number
+	Convert to number
˜	Invert bits
!	Invert boolean value
delete	Remove a property
typeof	Determine type of operand
void	Return undefined value
*, /, %	Multiply, divide, remainder
+, -	Add, subtract
+	Concatenate strings
<<	Shift left
>>	Shift right with sign extension
>>>Shift right with zero extension	
<, <=, >, >=	Compare in numeric order
<, <=, >, >=	Compare in alphabetic order
instanceof	Test object class
in	Test whether property exists
==	Test for equality
!=	Test for inequality
===	Test for strict equality
!==	Test fo strict inequality
&	Compute bitwise AND
ˆ	Compute bitwise XOR
—	Compute bitwise OR
&&	Compute logical AND
——	Compute logical OR
?:	Ternary operator
=	Assign to a variable or property
*=, /=, %=, +=, -=, &=, ˆ=,—=,<<=, >>=, >>>=	Operate and assign
,	Discard first operand, return the second

Table D.1: `JavaScript` operators' order of precedence, from the higher order to the lower order.)

D.3 Reserved Keywords

```
break, case, catch, continue, debugger,default,delete,do,
else,false,finally,for,function,if,in,instanceof,new,null,
,return,switch,this,throw,true,try,typeof,var,void,while,
with
//Unused reserved word for furture versions of JavaScript
class,const,enum,export,extends,import,super
//Reserved in strict mode
```

```
implements, interface, let, package, private, protected, public, static, yield
//not fully reserved but not allowed as variable, function or parametter names
eval,arguments
//Reserved in ECMAScript 3 but not in ECMAScript 5
abstract, boolean, byte, char, class,const, double, enum, export, extends,
 final , float , goto, implements, import, int, interface, long, native,
package, private, protected, public, short, static , super, synchronizd,
throws, transient , volatile
//Predinied global variables and functions
arguments, Array, Boolean, Date, decodeURI, decodeURIComponent, encodeURI,
encodeURIComponent, Error, eval, EvalError, Function, Infinity, isFinite,
isNan, JSON, Math, NaN, Number, Object, parseFloat, parseInt, RangeError,
ReferenceError, RegExp, String, SyntaxError, TypeError, undefined, URIError.
```

Appendix E
Natural deduction rules

	introduction	elimination
$\wedge$	$\dfrac{\phi \qquad \psi}{\phi \wedge \psi}\,{\scriptstyle \wedge i}$	$\dfrac{\phi \wedge \psi}{\phi}\,{\scriptstyle \wedge e_1}$ $\quad$ $\dfrac{\phi \wedge \psi}{\psi}\,{\scriptstyle \wedge e_2}$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\,{\scriptstyle \vee i_1}$ $\quad$ $\dfrac{\psi}{\phi \vee \psi}\,{\scriptstyle \vee i_2}$	$\dfrac{\phi \vee \psi \quad \boxed{\begin{array}{l}\phi\\ \;\vdots\\ \chi\end{array}} \quad \boxed{\begin{array}{l}\psi\\ \;\vdots\\ \chi\end{array}}}{\chi}\,{\scriptstyle \vee e}$
$\rightarrow$	$\dfrac{\boxed{\begin{array}{l}\phi\\ \;\vdots\\ \chi\end{array}}}{\phi \rightarrow \chi}\,{\scriptstyle \rightarrow i}$	$\dfrac{\phi \qquad \phi \rightarrow \psi}{\psi}\,{\scriptstyle \rightarrow e}$
$\neg$	$\dfrac{\boxed{\begin{array}{l}\phi\\ \;\vdots\\ \bot\end{array}}}{\neg\phi}\,{\scriptstyle \neg i}$	$\dfrac{\phi \qquad \neg\phi}{\bot}\,{\scriptstyle \neg e}$
$\bot$		$\dfrac{\bot}{\phi}\,{\scriptstyle \bot e}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}\,{\scriptstyle \neg\neg e}$

Figure E.1: Natural deduction rules for propositional logic.

Appendix F
Automatic Verification of Protocols

The codes presented in this Chapter are the model of the TCRR protocol and the `VerifyMyVM` protocol detailed in Chapter 7. There are two versions for each protocol, one corresponding to the AVISPA verification software and the other to Scyther. The results of the verification is presented in Chapter 7.

```
const PCR;

hashfunction hash;

protocol tcrr(User,Node,TPM){
  role User {
   fresh n1: Nonce;
   fresh n3: Nonce;
   var n2: Nonce;
   fresh ksession: SessionKey;
   send_1(User,Node,{n1,Node}sk(User));
   read_4(Node,User,n2,{{{PCR,n1,n2,Node}hash}sk(TPM),n2}sk(Node));
   send_10(User,Node,{{n1,n2,n3,ksession,User}pk(Node)}sk(User));
   read_13(Node,User,{n3}ksession);
   claim_u1(User,Niagree);
   claim_u2(User,Secret,ksession);
   claim_u3(User,Nisynch);
  }
  role Node {
   var n1: Nonce;
   fresh n2: Nonce;
   var n3: Nonce;
   var ksession: SessionKey;
   read_1(User,Node,{n1,Node}sk(User));
   send_2(Node,TPM,n1,n2,Node);
   read_3(TPM,Node,{{PCR,n1,n2,Node}hash}sk(TPM));
   send_4(Node,User,n2,{{{PCR,n1,n2,Node}hash}sk(TPM),n2}sk(Node));
   read_10(User,Node,{{n1,n2,n3,ksession,User}pk(Node)}sk(User));
   send_13(Node,User,{n3}ksession);
   claim_n3(Node,Nisynch);
   claim_n1(Node,Niagree);
   claim_n2(Node,Secret,ksession);
  }
  role TPM{
    var n1: Nonce;
    var n2: Nonce;
    read_2(Node,TPM,n1,n2,Node);
    send_3(TPM,Node,{{PCR,n1,n2,Node}hash}sk(TPM));
  }
}
/*const Ursula,Nadia,Eve,Tony : Agent;*/
/*untrusted Eve;*/
/*compromised sk(Eve);*/
/*compromised sAIK(Eve);*/
/*compromised sksubEK(Eve);*/
```

Listing F.1: TCRR protocol modelization & validation under Scyther.

```
 $> time ./scyther.py --max-runs=20 --all-attacks
  verification_scyther.spdl
  Verification results:
 claim id [tpmp,u1], Niagree : No attacks.
 claim id [tpmp,u2], Secret ksession : No attacks.
 claim id [tpmp,u3], Nisynch : No attacks.
 claim id [tpmp,n3], Nisynch : No attacks.
```

```
claim id [tpmp,n1], Niagree : No attacks.
claim id [tpmp,n2], Secret ksession : No attacks.
real        0m6.029s
user        0m5.950s
sys         0m0.060s
```

Listing F.2: Validation of the TCRR protocol with Scyther.

```
role r_user (User,Node: agent, Snd, Rcv: channel(dy)
    , Pkuser, PkAIK, PksubEK , Pknode : public_key
    , PCR : text, H: hash_func)
played_by User
def=
  local State : nat, N1,N2,N3 : text,
    Ksession : symmetric_key
  init
    State := 0
  transition
   1. State = 0 /\ Rcv(start)
   =|>  State':= 1 /\ N1':=new() /\
   Snd({N1'.Node}_inv(Pkuser))
   2. State = 1 /\ Rcv(
  {{H(PCR.N1.N2'.Node)}_inv(PkAIK).N2'}_inv(Pknode))
   =|> State' := 2 /\ N3':=new() /\
   Snd({{N1.N2'.N3'.Ksession}_PksubEK}_inv(Pkuser))
   3. State = 2 /\ Rcv({N3}_Ksession) =|> State':=3
   /\ secret(Ksession,sec,{User,Node})
   /\ request(User,Node,N3,valid)
end role
role r_node (Node,User,TPM : agent,
  SndUser, RcvUser,SndTPM, RcvTPM : channel(dy),
  Pkuser, PkAIK, PksubEK,Pknode : public_key ,
  PCR : text, H: hash_func)
played_by Node
def=
  local  State : nat, N1,N2,N3 : text,
    Ksession : symmetric_key
  init
    State := 1
  transition
   1. State = 1
   /\  RcvUser({N1'.Node'}_inv(Pkuser))
   =|> State' := 2 /\ N2':= new()
   /\ SndTPM(N1'.N2'.Node')
   2. State = 2
   /\ RcvTPM({H(PCR.N1.N2.Node)}_inv(PkAIK)) =|>
   State' := 3
   /\ SndUser(
   {{H(PCR.N1.N2.Node)}_inv(PkAIK).N2}_inv(Pknode))
   3. State = 3 /\ RcvUser(
   {{N1.N2.N3'.Ksession'}_PksubEK}_inv(Pkuser))
   =|> State' := 4
   /\ SndUser({N3'}_Ksession')
   /\ witness(User,Node,N3,valid)
end role
role r_tpm (TPM ,User,Node: agent,
     Snd, Rcv: channel(dy),
     Pkuser, PkAIK, PksubEK,Pknode : public_key,
     PCR : text, H: hash_func)
played_by TPM
def=
  local  State  : nat, N1,N2 : text
  init
    State := 10
  transition
   1. State = 10
   /\  Rcv(N1'.N2'.Node') =|> State' := 11 /\
   Snd({H(PCR.N1'.N2'.Node')}_inv(PkAIK))
end role
role session(User,Node,TPM : agent,
    Pkuser, PkAIK, PksubEK ,Pknode: public_key,
    PCR : text ,H: hash_func
)
def=
  local SndUser,RcvUser : channel (dy),
        SndTPM,RcvTPM : channel (dy)
  composition
     r_user(User,Node,SndUser,RcvUser,Pkuser,
                     PkAIK, PksubEK,Pknode,PCR,H)
     /\ r_node(Node,User,TPM,SndUser,RcvUser,SndTPM,
    RcvTPM,Pkuser, PkAIK, PksubEK,Pknode,PCR,H)
     /\ r_tpm(TPM,User,Node,SndTPM,RcvTPM,Pkuser,
                     PkAIK, PksubEK,Pknode,PCR,H)
end role
role environment() def=
   const user,tpm,node : agent,
   pcr,quote,unbind,extend,soft,zeros:text ,
   pkuser, pkAIK ,pksubEK,pknode,pki : public_key,
   ash: hash_func, sec,valid :protocol_id
  intruder_knowledge = {user,tpm,node,pkuser,zeros,
    pkAIK,pksubEK,pknode ,pcr,has,quote,unbind,
    extend,pki,inv(pki)}
  composition
  session(user,node,tpm,pkuser, pkAIK,
                          pksubEK,pknode,pcr,ash)
/\  session(user,i,tpm,pkuser, pkAIK,
                          pksubEK,pknode,pcr,ash)
/\  session(i,node,tpm,pkuser, pkAIK,
                          pksubEK,pknode,pcr,ash)
/\  session(user,node,i,pkuser, pkAIK,
                          pksubEK,pknode,pcr,ash)
end role
goal
  secrecy_of sec
  authentication_on valid
end goal
environment()
```

Listing F.3: TCRR protocol modelization & validation under AVISPA.

```
 $> ./avispa verification_avispa.hlpsl --ofmc
 % OFMC
 % Version of 2006/02/13
 SUMMARY SAFE
 DETAILS BOUNDED_NUMBER_OF_SESSIONS
 PROTOCOL
   /home/benoit/git/tcrr/avispa/results_verif.if
 GOAL    as_specified
 BACKEND OFMC
 COMMENTS
 STATISTICS
   parseTime: 0.00s
   searchTime: 7.93s
   visitedNodes: 6696 nodes
   depth: 17 plies
```

Listing F.4: Validation of the TCRR protocol with AVISPA.

```


protocol tcrr(User,CA,Node,TPM){
  role CA {
   const PCR: Nonce;
    var n1: Nonce;
   read_1(User,CA,{TPM,Node,CA,n1}sk(User));
   send_2(CA,User,{pk(TPM),pk(Node),TPM,Node,PCR,User,n1}sk(CA));

   claim_u1(CA,Niagree);
   claim_u3(CA,Nisynch);
   }
  role User {
   //User knows pk(CA) prior to the communication
```

```
    fresh n1: Nonce;
    var PCR: Nonce;
    send_1(User,CA,{TPM,Node,CA,n1}sk(User));
    read_2(CA,User,{pk(TPM),pk(Node),TPM,Node,PCR,User,n1}sk(CA));

    claim_u1(User,Niagree);
    claim_u3(User,Nisynch);
  }

}
```

Listing F.5: Validation communication between the user and the CA prior to the TCRR protocol with Scyther.

```
secret ksession: Function;
const pk: Function;
secret sk: Function;
const PCR : Nonce;
inversekeys (pk,sk);
protocol verifvm(User,NodeTPM){
  role User {
    const n1: Nonce;
    send_1(User,NodeTPM,{n1}ksession(User,NodeTPM));
    read_5(NodeTPM,User,
         {n1,{PCR}sk(NodeTPM)}ksession(User,NodeTPM));
    claim_u1(User,Niagree);
    claim_u3(User,Nisynch);
  }
  role NodeTPM{
    var n1: Nonce;
    read_1(User,NodeTPM,{n1}ksession(User,NodeTPM));
    send_5(NodeTPM,User,
         {n1,{PCR}sk(NodeTPM)}ksession(User,NodeTPM));
    claim_n3(NodeTPM,Nisynch);
    claim_n1(NodeTPM,Niagree);
  }
}
const Ursula,Nadia,Eve : Agent;
untrusted Eve;
```

Listing F.6: VerifyMyVM protocol modelization & validation under Scyther.

```
$ time ./scyther.py verification_vm.spdl
 Verification  results:
claim id [verifvm,u1], Niagree  : No attacks.
claim id [verifvm,u3], Nisynch  : No attacks.
claim id [verifvm,n3], Nisynch  : No attacks.
claim id [verifvm,n1], Niagree  : No attacks.
real     0m0.055s
user     0m0.034s
sys      0m0.019s
```

Listing F.7: Validation of the VerifyMyVM protocol with Scyther.

```
role r_user (User,Node: agent, Snd, Rcv: channel(dy),
    PkAIK : public_key, Ksession : symmetric_key,
    PCR : text, H: hash_func)
played_by User
def=
  local State : nat,N1 : text
  init
    State := 0
  transition
   1. State = 0 /\ Rcv(start)
   =|>  State':= 1 /\ N1':=new() /\
   Snd({N1'}_Ksession)
   2. State = 1
    /\ Rcv({{H(PCR.N1)}_inv(PkAIK).N1}_Ksession)
    =|> State':=2
    /\ request(User,Node,N1,valid)
end role
role r_node (Node,User : agent,
 SndUser, RcvUser : channel(dy), PkAIK : public_key,
 Ksession : symmetric_key, PCR : text, H: hash_func)
played_by Node
def=
  local  State : nat, N1 : text
  init
    State := 1
  transition
   1. State = 1
   /\  RcvUser({N1'}_Ksession)
   =|> State' := 2
   /\ SndUser({{H(PCR.N1)}_inv(PkAIK).N1}_Ksession)
   /\ witness(User,Node,N1,valid)
end role
role session(User,Node : agent,
    PkAIK: public_key, Ksession : symmetric_key,
    PCR : text ,H: hash_func
)
def=
  local SndUser,RcvUser : channel (dy)
  composition
     r_user(User,Node,SndUser,RcvUser,
                    PkAIK, Ksession ,PCR,H)
     /\ r_node(Node,User,SndUser,RcvUser,
                    PkAIK, Ksession ,PCR,H)
end role
role environment() def=
   const user,node : agent,
   pcr:text , ksession : symmetric_key,
   pkAIK ,pki : public_key,
   ash: hash_func, valid :protocol_id
  intruder_knowledge = {user,
    pkAIK,pcr,has, pki,inv(pki)}
  composition
    session(user,node,pkAIK,ksession,pcr,ash)
/\  session(user,i,pkAIK,ksession,pcr,ash)
/\  session(i,node,pkAIK,ksession,pcr,ash)
/\  session(user,node,pkAIK,ksession,pcr,ash)
end role
goal
  authentication_on valid
end goal
environment()
```

Listing F.8: VerifyMyVM protocol modelization & validation under AVISPA.

```
$> ./avispa verification_vm_avispa.hlpsl --ofmc
% OFMC
% Version of 2006/02/13
SUMMARY SAFE
DETAILS  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/benoit/git/verifymyVM/avispa/results_verif.if
GOAL      as_specified
BACKEND OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 0.07s
  visitedNodes: 158 nodes
  depth: 6 plies
```

Listing F.9: Validation of the VerifyMyVM protocol with AVISPA.

Appendix G
Table of Obfuscation Transformations

Obfuscation			Quality			Metrics
Target	Operation	Transformation	Potency	Resilience	Cost	
Layout		Scramble Identifiers	medium	one-way	free	
		Change Formatting	low	one-way	free	
		Remove Comments	high	one-way	free	
Control	Computations	Insert Dead or Irrelevant Code	Depends on the quality of the opaque construct and on the nesting depth of its insertion			μ_1, μ_2, μ_3
		Extend Loop Condition				$\mu_1, \mu_2, \mu 3$
		Reducible to non-Reducible				$\mu_1, \mu_2, \mu 3$
		Add Redundant Operands				$\mu_1, \mu_2, \mu 3$
		Parallelize Code	high	strong	costly	μ_1, μ_2
	Aggregation	Inline Method	medium	one-way	free	μ_1
		Outline Statements	medium	strong	free	μ_1
		Interleave Functions	Depends on the quality of the opaque predicate			μ_1, μ_2, μ_5
		Clone Functions				μ_1
		Block loop	low	weak	free	μ_1, μ_2
		Unroll loop	low	weak	cheap	μ_1
		Loop fission	low	weak	free	μ_1, μ_2
	Ordering	Reorder Statements	low	one-way	free	
		Reorder Loops	low	one-way	free	
		Reorder Expression	low	one-way	free	
Data	Storage & Encoding	Change Encoding	Depends on the complexity of the encoding function			μ_1
		Promote Scalar to Object	low	strong	free	
		Change Variable Lifetime	low	strong	free	μ_4
		Split Variable	Depends on the number of variables into which the original variable is split			μ_1
		Convert Static to Procedural Data	Depends on the complexity of the generated function			μ_1, μ_2
	Aggregation	Merge Scalar Variables	low	weak	free	μ_1
		Split Array	*	weak	free	μ_1, μ_2, μ_6
		Merge Arrays	*	weak	free	μ_1, μ_2
		Fold Array	*	weak	cheap	$\mu_1, \mu_2, \mu_6, \mu_3$
		Flatten Array	*	weak	free	
	Ordering	Reorder Functions & Variables	low	one-way	free	
		Reorder Arrays	low	weak	free	

Table G.1: Obfuscation transformations and their qualities.

List of Figures

List of Tables