Università
Ca'Foscari
Venezia

DEPT. OF ENVIRONMENTAL SCIENCES, INFORMATICS AND STATISTICS
Master's degree programme in Informatica – Computer Science
Second Cycle (D.M. 270/2004)

Final thesis

Security Enhancements of Robot Operating Systems

Supervisor:
Chiar.mo Prof.
Agostino Cortesi

Graduand:
Gianluca Caiazza
Matriculation Number
840009

Academic Year:
2015 – 2016

If you reveal your secrets to the wind, you should not blame the wind for revealing them to the trees

— Kahlil Gibran

ABSTRACT

In recent years we observed a grown of cybersecurity threats, especially thanks to the ubiquitous of connected and autonomous devices commonly know as Internet of Things. The notion of "Internet of Things", covers industrial automation, autonomous vehicles, robot-assisted surgery, surveillance platforms, home service automation and many more robotics domains. Considering the sensitive information that are processed by these devices, the possibility of attacks should be considered as a serious security matter. This thesis focuses on the Robot Operating System (ROS), a widely adopted standard robotic middle-ware. We analyse its possible vulnerabilities and the resulting threats that could be posed by attackers. More in details, the present work will provide an in-depth analysis of ROS and SROS - a proposed addition to the ROS API ecosystem to support modern cryptography and security measures - in addition to the development of a static analyser upon SROS for the automatic creation of software-enforcement security profiles. In order to do so, we perform the following tasks: (1) standardize the security logging format; (2) standardize the profile syntax for the policy; (3) provide new tools to introspect recorded security logs. Finally, we discuss some enhancements for SROS following the standard proposed by oneM2M.

ACKNOWLEDGEMENTS

First of all, I would like to thank my supervisor Agostino Cortesi for his guidance and assistance throughout the writing of this thesis. Besides my advisor, I would like to thank Professor Young Im Cho for her valuable advices and her patience. My sincere thanks also goes to Ruffin White, you definitely provided me with the ideas that I needed to choose the right direction and successfully complete my thesis.

I want to thank my whole family, in particular my parents, my brother, my grandparents and my uncle Vincenzo; they motivate me to reach my objectives, and have always supported me spiritually not only throughout the writing of this thesis but also during my whole life.

Finally, I would like to thank my dear friends Roberta and Gaia, who have shared with me the joy of the university life, you have made quite enjoyable even the assignments!

My '*core*' friends Giacomo, Gianmarco and Marco who always mock me - I promise I will return the favour -, and all the others that for matter of space I can't write here. However, I would like to single out '*carbon paper*' Francesco, Maksym and the '*globe-trotter*' Francesco for their presence in my life since forever, thank you guys!

Thank you very much, everyone!

CONTENTS

1 INTRODUCTION 1

1.1 Possible Attacks . . . 3

1.2 Middleware or Wrapper? . . . 6

1.3 Related Work . . . 7

1.4 Validation of the Approach . . . 8

1.4.1 API . . . 8

1.4.2 Communication . . . 9

1.5 Thesis Structure . . . 10

I ROS ANALYSIS 11

2 ROS 12

2.1 Ros Levels . . . 13

2.1.1 Filesystem Level . . . 14

2.1.2 Computation Graph Level . . . 15

2.1.3 Community Level . . . 19

2.2 Client Libraries . . . 19

2.3 Techical Overview . . . 19

2.3.1 Master . . . 20

2.3.2 Parameter Server . . . 20

2.3.3 Node . . . 21

2.4 Communications . . . 22

2.4.1 Establishing a topic connection . . . 23

2.4.2 Establishing a service connection . . . 24

3 ROS 2 26

3.1 Structure of the middleware . . . 28

4 SROS 31

4.1 Computational Graph . . . 31

4.1.1 Public key Infrastructure 32
4.1.2 Access Control 35
4.2 Filesystem 36

II SOFTWARE ENHANCEMENTS 38

5 ACCESS CONTROL 39
5.1 Terminology 40
5.2 Mandatory Access Control 40
5.2.1 SELinux and AppArmor 41

6 SECURITY LOGGING 43
6.1 AppArmor logging 44
6.2 ROS API logging 45

7 POLICY PROFILE SYNTAX & GENERATION 49
7.1 AppArmor Syntax 49
7.1.1 Permission access mode 50
7.1.2 Profile structure 51
7.2 ROS 52
7.3 API partition 55
7.3.1 Master API 58
7.3.2 Parameter API 60
7.3.3 XML-RPC connection 61
7.4 Profile Autogeneration 62

III PROPOSITIONS 70

8 CERTIFICATES 71
8.1 User-Pull Architecture 71
8.1.1 Monolithic 74
8.1.2 Autonomic 76
8.1.3 Chained signatures 78
8.2 Server Pull Architecture 79

9 ONEM2M 82

10 CASE OF STUDIES 85
10.1 Certificate Overhead 85
10.2 Encryption Overhead 87
11 CONCLUSIONS 90

IV APPENDIX 92

A ROS API 93
A.1 Parameter 93
A.2 Slave 93
A.3 Master 94
A.4 Legacy connection 95

LIST OF FIGURES

Figure 1 ROS Levels 13
Figure 2 ROS Concepts 17
Figure 3 ROS Graph 23
Figure 4 ROS Middleware Adapter 29
Figure 5 ROS Middleware Communication 30
Figure 6 SROS Keyserver 34
Figure 7 Globbing syntax 66
Figure 8 User-Pull model 73
Figure 9 Monolithic certificate 74
Figure 10 Autonomic certificate 76
Figure 11 Chained certificate 78
Figure 12 Server-Pull model 81
Figure 13 TLS handshake 86

LIST OF TABLES

Table 1 Names Resolution 18
Table 2 Topic Connection 24
Table 3 Service Connection 25
Table 4 ROS API division 56
Table 5 Slave API 58
Table 6 Master API 60
Table 7 Parameter API 61
Table 8 Transport API 62

ACRONYMS

AA attribute authority

AC Attribute certificate

ACL Access Control List

AES Advanced Encryption Standard

API Application Programming Interface

CA Certificate Authority

CLI Command Line Interface

CRL Certificate Revocation List

DAC Discretionary Access Control

DDS Data Distribution Service

ETSI European Telecommunications Standards Institute

GSM Global System for Mobile Communication

ICT information and communication technology

IDC Identity certificate

IDL Interface Description Language

IOT Internet of Things

M2M Machine-to-Machine

MAC Mandatory Access Control

MCS Multi-Category Security

MLS Multi-Level Security

NSA National Security Agency

OMG Object Management Group

OSRF Open Source Robot Foundation

PKC X.509 public key certificate

RBAC Role Based Access Control

ROS Robot Operating System

RPC Remote Procedure Call

RTPS Real-Time Publish Subscribe

SDN Software-Defined Networking

SPOF Single Point of Failure

TE Type Enforcement

UAV Unmanned Aerial Vehicle

VPN Virtual Private Network

1

INTRODUCTION

In recent years, we observed a growth of cybersecurity threats, especially due to the ubiquitous of connected and autonomous devices commonly defined as Internet of Things (IoT).

Nowadays it isn't uncommon to wear several connected devices that collect all kind of information. Considering the sensitive nature of this information that are processed by these devices, the possibility of attacks should be considered as a serious security matter.

However, it is sufficient to give a glance at the industry market to observe how this is crammed with proprietary closed implementations that almost on daily base hit the headlines about a new security threat that often doesn't receive an opportune fix. There have been some feeble movements for the definition of new standards with the objective of merge the different solutions into a comprehensive one with the goal of enhancing interoperability and, overall, security (e.g. ETSI). Nowadays we believe that the most promising one is *oneM2M* [**14**]. Established through the alliance of seven standards organisations, five global information and communication technology (ICT) in addition to more that 200 companies in the telecoms and IT industries, oneM2M aims to develop a single *horizontal* platform for exchange and share data among all applications. The usage of that platform enables improved functionality-cost-quality tradeoff in addition to enhancements in scalability and security by design of Machine-to-Machine (M2M) communications.

Following the same objectives of oneM2M, this thesis focuses on the Robot Operating System (ROS) [**15**], a widely adopted standard open-source robotic middle-ware. This framework has several philosophical goals that make it a

suitable choice for stakeholders that need to reuse cross-projects software and need to develop their solutions in heterogeneous device networks. The peculiarity of ROS is the *peer-to-peer* topology that connects at runtime a number of processes that potentially can be dislocated in a number of different hosts. However, from our point of view, it's important to analyse how the communications are handled in the network. In fact, everything is managed with a simple `Publish-Subscribe` interface with no encryption method at all. In addition to that, each node (agent) doesn't have a mechanism to identify the other agents in the network.

In the early days of ROS, since all the communications were handled by physical connections in autonomous networks (i.e. CAN-bus), the security aspect was overlooked. However, in these days, we are observing an increase of projects that use long-range radio connections (e.g. GSM) as primary communication mechanism that requires some degree of cybersecurity.

That said, the goal of this thesis is to define software security enhancements for the communication level in ROS computational graph. In detail, we want to develop security mechanisms in ROS's Publish-Subscribe paradigm by means of secure communications and different authorisation solutions. That said, the proposed solution, need to respect the following points:

- Legacy application support: we don't want to tamper the existing applications in order to fit our solution.
- Node authentication: in order to implement authentication, we need to design mechanisms to unambiguously identity the entities in ROS.
- Realtime: although ROS is not a realtime framework, we want to maintain the agility of the system. This means that we need to design nimble solutions and mechanisms.
- Power consumption: we need to remember that we are in a low power device network in which, if possible, we should ensure security without wasting power resources that doesn't introduce too much overhead.

- Security by design: tightly coupled with the previous two constraints, we want to develop solutions that doesn't need byzantine mechanisms for ensure security.

In order to do so, we define upon SROS a static analyser for the automatic generation of software-enforcement security profiles for ROS's agents. In addition, we introduce the concept of authentication by means of x.509 certificates.

The work is organised in three tightly coupled macro-tasks that leads us to the definition of an high level static set of tools for the creation of policy profiles:

- Security Logging: in this task we make logging in SROS more parsable and pervasive among ROS nodes without breaking the legacy ROS tools.
- Policy Profile Syntax: we define a standard for SROS policy profile
- Policy Profile Generation: we define new tools to audit and auto-generate policy profiles from the standardised logs

Thanks to this new approach, we are able to provide to users a mechanism to define security policy profiles without a great deal of effort. Moreover, we give an high level solution that permits to maintain the same security profile structure independently from the back-end ROS implementation.

1.1 POSSIBLE ATTACKS

In this section we give an overview of the threats deriving from an environment modification in ROS applications. By changing from a wired solution to a cellular network, the entrance point of a malicious user exponentially increase. Let's consider the example of a remote 4G control of an unmanned aerial vehicle (UAV) or simply 'drone'. By means of this example we want to discuss the weak points of the current implementation of ROS.

We can distinguish two possible macro-groups of attacks: `physical` and `logical`.

Starting from the physical, we can consider two kind of attacks: `contact` and `no-contact`. The first defines attacks in which the device is shotted down (e.g. with a gun) or simply captured. Instead, the latter albeit in the physical group, defines *cyber-physical* attacks to communication bridges; in detail, in this group we have attacks as: jamming the device, tampering the messages with a signal in same wavelength or simply dossing the communication bridge. However, this kind of attacks, although considered in this analysis phase, are not covered in this thesis. We believe that counter measures against this kind of attacks should be considered in the intra-device security design, which is out of scope of this document.

So, considering logical attacks only, it's easy to understand the importance of ensuring the key properties of computer security:

- `confidentiality`: protect the information from disclosure to unauthorised entities.
- `integrity`: protect information from being modified by unauthorised entities.
- `authenticity`: introduce mechanisms to authenticate the entities in the system.
- `non-repudiation`: an entity of a transaction cannot deny having received a transaction nor can the other entity deny having sent a transaction.
- `availability`: ensure that authorised entity are able to access the information when needed.

The reason why this concepts are so important, is due to the fact that we want to define from scratch a secure system for ROS. In fact, in the current

implementation, all the communication are managed with a simple Publish-Subscribe interface with no encryption or security methods at all. All in all, we need to develop upon the existing infrastructure methods to allow these security properties. Furthermore, by applying these simple rules we develop countermeasures against 'standard' literature attacks as:

- `Eavesdropping` (or interception): the attacker "listens" to the flow of data, accessing information and therefore breaking *confidentiality*. Since in ROS a publisher send everything as plaintext we are not able to avoid and block this kind of attacks.

- `Modification`: the attacker intercepts the data and modifies it, breaking *integrity* of information. The attacker can put himself in the middle of the communication between the remote and the drone and change the content of the message

- `Forging or impersonation`: the attacker introduces new information and lets the destination believe that the information is coming from the honest source. It breaks *authentication*. Similarly to the previous one, by design, ROS allows the presence of multiple publisher and subscriber for the same topic (see **??**) This is tricky because it allows an attacker to steal the device without any knowledge from the administrators.

- `Repetition` (reply-attack): re-send a message that has been intercepted in the past. In this way, an attacker can build a payload that allows him to do everything he wants. In order to fix this kind of attacks, a possible solution is the introduction of an 'approval' time windows for the messages. In this setup we need to enrich the sent message API. In practice, it must contain at least a time stamp and a modern hash function.

1.2 MIDDLEWARE OR WRAPPER?

On the implementation side, we have two alternative solutions. We could achieve a suitable solution by developing a middleware that takes care of the management of the network agents. Thanks to this horizontal approach, we could be independent from the development of ROS and we can build a back-end/front-end system that we could manage independently. However, in order to make the middleware interact with the existing infrastructures, it's needed to tamper some ROS's API functions.
The current implementation of ROS is deeply-rooted in a *master/slave* architecture that potentially could trap us in vicious cycle in which we need to constantly apply our solution at each ROS update (that may also broke in case of a major release). In ROS we have several entities that interact in the computational graph via API, developing a new horizontal middleware means that we need to manually modify all the ROS's API that allows these interactions. In order to overcame this problem, we need to find a way to build upon the existing ROS APIs a software 'layer' that allow us to enhance the way in which ROS entities interact. In informatics this particular layer is called `wrapper`. By means of a wrapper instead of a middleware, we are able to design high level security features leveraging on the already existing API functions. The idea is to build upon each function a new set of operations that allow us to enhance their behaviour by fitting additional security mechanisms as authentication. As such, we maintain the higher approach which means develop a wrapper.

From the analysis of ROS 2 (Chapter 3) we will see that OSRF is going to change virtually everything related to the APIs from ROS 1, and it will introduce the concept of middleware interfaces. Therefore, in the last part of this thesis we introduce other enhancements and solutions to make ROS able to communicate in an oneM2M miscellaneous network leveraging on the new middleware interfaces.

1.3 RELATED WORK

To the best of our knowledge, the present work is the first research focusing on the discussion of a set of tools and techniques for the automatic definition of security policy profiles in a robotic framework. Although several security threats analysis in the industrial robotic applications have been performed recently, they mainly focus on the threats deriving from the modification of network topology from local to remote connection [**20**, **22**, **23**].

There are two possible ways to tackle the communication problem: from the *communication channel* or from the *communication paradigm*.

In the first case Akerberg el al. [**21**], have proposed a security communication framework for the integration in classical wired industrial networks of wireless nodes, with the implementation of end-to-end integrity and authentication measures by utilising the black channel concept.

On the other hand, Wang *et al.* [**25**], analyse Publish-Subscribe communication paradigm in a wide-area network. In this particular setup services handle information across distinct authoritative domains and need to manage a large population of publishers and subscribers. In detail, they discuss about the security issues and requirements that arise, distinguishing among those requirements that can be achieved with current technology and those that require novel solutions.

Similarly to our solution, in [**26**] Dieber *et al.* proposed a security architecture intended for use on top of ROS on the application level. By means of an authorisation server and the usage of x.509 certificates for authentication, they ensure that only valid registered nodes are authorised to execute some operations. Furthermore, with the usage of cryptographic methods they ensure data confidentiality and integrity.

In detail, each node have a x.509 certificate with a unique serial number. Leveraging on these certificates the nodes authenticate themselves and verify if the requesting node is authorised to execute the operation. However, this

statical architecture is based on the assumption that we have manually generated and distributed the certificates and registered the list of nodes in the authorisation server; moreover, they delegate to the user the distribution of the lists of certificates serial number of the nodes that are authorised to query each other.

Additionally, Lera *et al.* [**17**], presented an interesting analysis on which they proposed that ROS communications should be encrypted. Differently to the previous discussions, in this document they directly evaluate the performance of ROS under encryption. In detail, they used 3DES cyphering algorithm and evaluated the performance both from the computing and the communications point of view.

1.4 VALIDATION OF THE APPROACH

We said that we want to define software enhancements for the communication graph level in ROS network; in terms of validation this means that we need to analyse how this new concepts are going to impact in graph security.

In the graph level, we have two mechanisms to communicate: the `communication channel` and the `APIs`. Where the first is simply the physical means to communicate and the latter are the public interface of each node. Let's evaluate these mechanism in sections below.

1.4.1 *API*

With the term API we are addressing the set of public defined methods to communicate between various agents in the graph; we can consider these methods from two point of view: physical and logical.

When we talk about the physical aspect of the API, we are referring to the effects that this functions have on the underling operating system; for example it may be possible to undermine the OS by means of a vulnerability in the

API implementations. Considering this threats, SROS already leverage a good deal by securing the Filesystem Level by means of the AppArmor Linux kernel security modules.

Thanks to the AppArmor's profile, SROS is able to manage the API access rights on the system as well as quarantine ROS processes. In this way we can block, at kernel level, malicious or dysfunctional nodes to access unauthorised area or files in the degree of disruption if they perform something not authorised.

On the other hand, in order to secure API calls from the logical point of view, we need to enhance the behaviour of the agents in the graph. Our objective is to define safe sets of operations and introduce the concepts of identification, authentication and authorisation. In this way, at cost of a supervised initialisation, we are able to logically confine the possible operations of a node to a finite set of operations. In this way we are able to granulise the access to information and resource per agent. So, we move the setup complexity to the statical definition of a suitable hierarchical structure for accessing the device resources (camera, sensors, services, etc) instead of keeping it at runtime with a stateful solution.

1.4.2 *Communication*

This part of the validation follows directly from what we stated in the previous section about logical attacks. In fact, we want to ensure that a bad actor is not capable of performing any of those. The usage of private-public certificates ensures that if we entitled a node to interact in a specific namespace, this will not be able to 'escape' this property (otherwise disruptive countermeasures are applied) enforcing integrity and authentication. In addition, thanks to the usage of encryption, we are able to guarantee confidentiality in the network.

1.5 THESIS STRUCTURE

The thesis is organised in three macro-blocks: *ROS Analysis, Software Enhancements, Propositions*.

In the first part, we give an overview of ROS in Chapter 2, ROS 2 in Chapter 3 and we introduce SROS [**16**], an addition to the ROS API and ecosystem to support modern cryptography and security measures developed by the OSRF, in Chapter 4.

In the second part, we give an introduction to access control and some tools in Chapter 5, then, we explain how we tackled the problem about autogenerating the policy profiles by standardising the logs in Chapter 6 and the profile syntax in Chapter 7, therefore in the same chapter we describe the generation phase of the certificate.

In the last part we discuss in Chapter 8 about privacy management with x.509 certificates and we propose alternative implementations; in Chapter 9 we discuss about the integration of ROS in oneM2M and in Chapter 10 we present some case of study on overhead introduced by encryption and by the usage of certificate in a network.

Finally, Chapter 11 concludes with final remarks and a discussion of future works.

Part I

ROS ANALYSIS

In this part we introduce the ROS framework and we discuss about the next iteration ROS 2. Then, we present SROS, a proposed addition to the ROS API and ecosystem to support modern cryptography and security measures upon which we develop our enhancements

2

ROS

ROS is an open-source, meta-operating system for robots as: manipulators, mobile robots, autonomous cars, social robots, humanoids, UAVs, UWVs and others. Just by looking to the 2016 ROS metrics report [**35**] it's enough to grasp the growth of this framework in both academic and industrial research. It provides the services of an operating system, including: hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management.

As said in the introduction, ROS implements a peer-to-peer network, namely "*graph*", in which the processes (nodes) can communicate at runtime via publish/subscribe [**19**] pattern.

ROS leverages the communication between processes on different styles, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on the parameter server. However, we notice that ROS is not a realtime framework: albeit it is possible to integrate ROS with realtime code there aren't mechanism to ensure natively realtime communications.

The choice of using different communication styles is due to the fact that the OSRF developers wanted to define ROS as a combination of physical level and split the code in different 'boxes'.

As such, we redefine ROS as a distributed framework of processes that enables them to be individually designed and loosely coupled at runtime. So, one of the main advantages of using ROS instead of other competitor solutions, is the ability of grouping processes into Packages and Stacks, which can be easily shared and distributed. This is possible because, we can logically imagine ROS

as a combination of three levels: filesystem level, graph level and community level.

Thanks to this division we are able to make independent decisions about development and implementation, leaving to developers the ability of integrate different kind of code. Below, we define these level and how we can manage ROS nodes, in addition we give a technical overview of the ROS's graph components.

2.1 ROS LEVELS

As said, ROS has three level of concepts: the Filesystem level, the Computation Graph level, and the Community level. In this section we want to summarise this high level concepts that defines all the ROS functionalities.

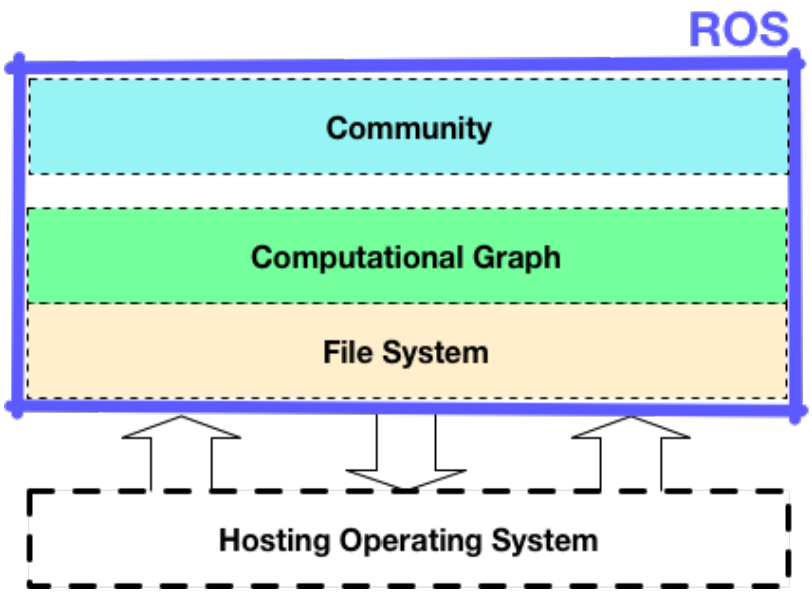

Figure 1: ROS Levels

2.1.1 *Filesystem Level*

We know that ROS is build upon a Unix operation system. This means, that it needs a mechanism to communicate at application level with the OS. As the name suggests, filesystem level covers ROS physical resources that you encounter on 'disk', such as:

- Packages: Packages are the main unit for organising software in ROS. A package may contain ROS runtime processes (nodes), a ROS-dependent library, datasets, configuration files, or anything else that is usefully organised together.
- Metapackages: specialised Packages which only serve to represent a group of related other packages. Most commonly metapackages are used as a backwards compatible place holder for converted rosbuild Stacks.
- Package Manifests: Manifests (package.xml) provide metadata about a package, including its name, version, description, license information, dependencies, and other meta information like exported packages.
- Repositories: A collection of packages which share a common VCS system. Packages which share a VCS share the same version and can be released together using the catkin release automation tool *bloom*. Often these repositories will map to converted rosbuild Stacks. Repositories can also contain only one package.
- Message types: Message descriptions, define the data structures for messages sent in ROS.
- Service types: Service descriptions, define the request and response data structures for services in ROS.

2.1.2 *Computation Graph Level*

From our analysis point of view, the computation graph level is the most important. Here is where need to concentrate our work to provide software security enhancements. As said, the computational graph is a peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph components of ROS are nodes, Master, parameter server, messages, services, and topics, all of which provide data to the 'graph' in different ways. Let's see in detail each component:

- Nodes: Nodes are the basic processes that perform computation. ROS is designed to be modular at a fine-grained scale; usually a robot control system comprises many nodes. For example, one node controls a camera, one node controls the temperature sensors and so on.
- Master: The ROS Master provides name registration and lookup to the rest of the Computation Graph. In practice, the master is a DNS server for nodes. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.
- Parameter Server: The Parameter Server allows data to be stored by key in a central location. Even thou we need to consider this as an independent component, currently it's part of the Master.
- Messages: Nodes communicate with each other by means of messages. A message is simply a data structure, comprising typed fields.
- Topics: Messages are routed via a transport system with publish/subscribe mechanism. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message. A node that is interested in a certain kind of data will subscribe to the appropriate topic. It's important to notice that there may be multiple concurrent publishers and subscribers for a single topic, and

a single node may publish and/or subscribe to multiple topics. It's important to notice that publishers and subscribers are not aware of each other's existence.

- Services: The topics publish/subscribe model is a very flexible communication paradigm, but its `many-to-many`, for *one-way* transport it's more appropriate a request/reply interactions. This kind of communications are done via services, which are defined by a pair of message structures: one for the request and one for the reply. A providing node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.

So, we know that the master acts as nameservice; as such, it stores topics and services registration information as well as nodes addresses. Each node need to communicate with the master, in order to receive information about other registered nodes and make connections as needed. The Master will also make callbacks to these nodes when this registration information changes, which allows nodes to dynamically create connections as new nodes are lunched.

Once a node receive the address of another one from the master, it can establish a direct connection. As example, if a node wants to subscribe to a topic, it needs to send a request connection to the node that publish that topic, and will establish that connection over an agreed upon connection protocol. The most common protocol used in a ROS is called `TCPROS`, which uses standard TCP/IP sockets.

This kind of architecture allows for decoupled operations, however we need to introduce the primary means by which each node can connect to another, i.e. names. Names are used for all the ROS's components: nodes, topics, services, and parameters all have names.

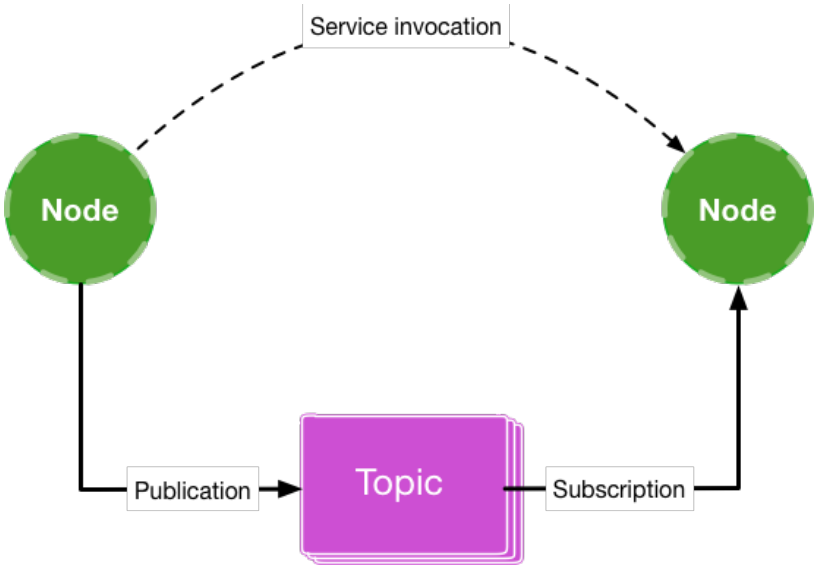

Figure 2: ROS Concepts

In detail, ROS provide a hierarchical naming structure that is used for all resources in a ROS computation graph. These names are very powerful in ROS and central to how larger and more complicated systems are composed in ROS, so it is critical to understand how these names work and how you can manipulate them.

Leveraging on Names, we can provide encapsulation. Each resource is defined within a `namespace`, which it may share with many other resources. In general, resources can create other resources within their namespace and they can access resources within or above their own namespace. This encapsulation isolates different portions of the system from accidentally picking the wrong named resource. This is the reason why we can reuse code from other implementation; from a developer point of view, it is enough to write the nodes that work together as if they are all in the top-level namespace

When these Nodes are integrated into a larger system, they can be pushed down into a namespace that defines their collection of code. For example, one could take a Stanford demo and a Willow Garage demo and merge them into a new demo with 'stanford' and 'wg' subgraphs. If both demos had a Node named 'camera', they would not conflict. Then, Tools (e.g. graph visualisation)

Node	Relative	Global	Private
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar

Table 1: Names Resolution

as well as parameters that need to be visible to the entire graph can be created by top-level Nodes.

We previously introduced the concept of 'pushing down' a resource into a namespace; however, in order to do so, we need to define namespace scopes. In fact, in ROS we can distinguish four types of Graph Resource Names: *base, relative, global, and private*.

Let's see below how they are defined:

- base
- relative/name
- /global/name
- ~private/name

By default, resolution is done relative to the node's namespace.

We need to be careful with the usage of the names that starts with "/"; these are global names and should be avoided as much as possible since they limit code portability of the implementation. On the other hand, names that start with a "~" (private) convert the node's name into a namespace. For example, the node *sensor* in namespace */drone/* has the private namespace */drone/sensor*. We exploit private names for passing parameters to a specific node via the parameter server. See Table 1 for other name resolution example.

2.1.3 *Community Level*

Finally, we define the community level. This is a 'pretended' level that is not physically present in ROS, but it represents ROS's developer community that share code structure and knowledge for the definition of more complex community based systems. In detail it's made of: ROS wiki, mailing lists, software repositories, etc.

2.2 CLIENT LIBRARIES

We said that ROS is a meta-operating system, but what does it means? Actually ROS is an application framework build upon a Unix operating system (Section 2.1.1). As such, we need a library to communicate with the ROS entities.

There are several libraries for ROS that range a remarkable number of programming languages, although current focus is on providing robust C++ (roscpp) and Python (rospy) support.

By means of these libraries developers can write ROS nodes, publish and subscribe to topics, write and call services, and use the Parameter Server.

By observing the ROS's repositories, we can notice that roscpp is the most widely used ROS client library and it's mostly deployed for high performance solutions. On the other hand, rospy favours implementation speed (i.e. developer time) over runtime performance, so we can quickly prototype and test algorithms within ROS.

2.3 TECHICAL OVERVIEW

After this general overview of ROS, we dive into a technical analysis of ROS's graph components. In detail, we discuss about: the master, parameter server, nodes and topic/service connections.

The reason why XMLRPC has been chosen instead of other HTTP/HTTPS protocols is mainly because it lightweight;

In fact, it doesn't require a stateful connection and its supported by a wide range of programming languages.

2.3.1 *Master*

The master is probably the most important component of the current implementation of ROS; implemented via XMLRPC - a stateless HTTP-based protocol - it exposes several functions to the other agents (aka nodes), as registration APIs for register as publishers, as subscribers, and service providers. Furthermore, it includes the mechanisms that allows the discoverability of agents in the network. All in all, we can picture the master as a *trusted* entity in the network that is queried by the nodes via its address (URI). In fact, each entity in the graph is qualified by a URI that corresponds to the `host:port` XML-RPC address of the entity in the server.

2.3.2 *Parameter Server*

Like the Master API, the Parameter Server is implemented via XMLRPC as well. The use of XMLRPC enables an easy integration with the ROS client libraries and also provides greater type flexibility when storing and retrieving data. The Parameter Server can store basic XML-RPC scalars, lists, and base64-encoded binary data. It can also store dictionaries (i.e. structs), but these have a special meaning; in fact, dictionary-of-dictionary is the structure used for the representation of the namespace; in detail, each dictionary represents a level in the naming hierarchy (aka namespace), and the other dictionary represents the internal structure of that namespace 'branch'.

Let's consider an example: we want to set the value of a variable *foo* in the namespace */node/bar/*. In terms of internal structure in the parameter server the value `/node/bar` would be a dictionary `{foo:1}` and the value `/node/` would be a dictionary `{bar:{foo:1}}`.

2.3.3 *Node*

A general node (aka process) has three types of APIs that expose it to the other agents:

1. Slave API: this kind of APIs are used by the node to receiving callbacks from the Master, and for negotiating connections with other nodes.
2. Transport protocol API: namely TCPROS and UDPROS. These methods are used by nodes as communications primitives. As example, if a node wants to establish a topic connection with another, it use these protocols to negotiate a connection. The difference between the two protocols is, trivially, the underling mechanisms: TCP - persistent, stateful TCP/IP socket connections - in TCPROS and UDP in UDPROS.
3. A command-line API: lastly, thanks to the XMLRPC back-end, each node supports command-line remapping arguments, which enables names within a node to be configured at runtime and low-level XMLRPC calls.

As the other entities of the graph, also every node has a URI, which corresponds to the `host:port`, and a XMLRPC embedded server; but this is not used to transport topic or service data, but it only provide Slave APIs to negotiate connections with other nodes and also communicate with the Master APIs.

2.4 COMMUNICATIONS

We want to focus on the way in which communications are handled in ROS. When we talk about networks, in general, TCP is widely used because it provides a simple reliable communication stream: TCP packets always arrive in order, and lost packets are resent until they arrive. However, while these features are great for reliable wired networks, they become less efficient when the underlying network is a lossy WiFi or GSM connection. In this situation, UDP is more appropriate.
So, with no surprise, also ROS doesn't commit to a single transport protocol but embrace both TCP and UDP. As previously said, when a node needs to connect to another one, it negotiates a connection using the appropriate transport via XMLRPC. The result of the negotiation is that the two nodes are directly connected, with messages streaming from publisher to subscriber via ROS's proprietary implementations of TCPROS and UDPROS. This means that each transport has its own protocol for how the message data is exchanged. For example, using TCPROS, the negotiation would involve the publisher giving the subscriber the IP address and port on which to call connect. The subscriber then creates a TCP/IP socket to the specified address and port. The nodes exchange a Connection Header that includes information like the MD5 sum of the message type and the name of the topic, and then the publisher begins sending serialised message data directly over the socket.

We can summarise the whole negotiating process as a two step communication between the nodes and the master. In the first part, the subscriber node queries the master (DNS server) for the publisher node address via XMLRPC, and then a negotiation between the nodes starts.

This process leaves some vulnerabilities that we need to handle. First of all, it may happen that a subscriber avoids to query the master because it already knows the address of the publisher; as such, we are not able to build a stateful solution based on the master only, but we need to implement additional

mechanisms to secure node-to-node communications. Furthermore, we need to sanitise also TCP/UDP communication and not limit our analysis to the XMLRPC channel. Following this overview, in the sections below we will see how the connection are established for topics and services.

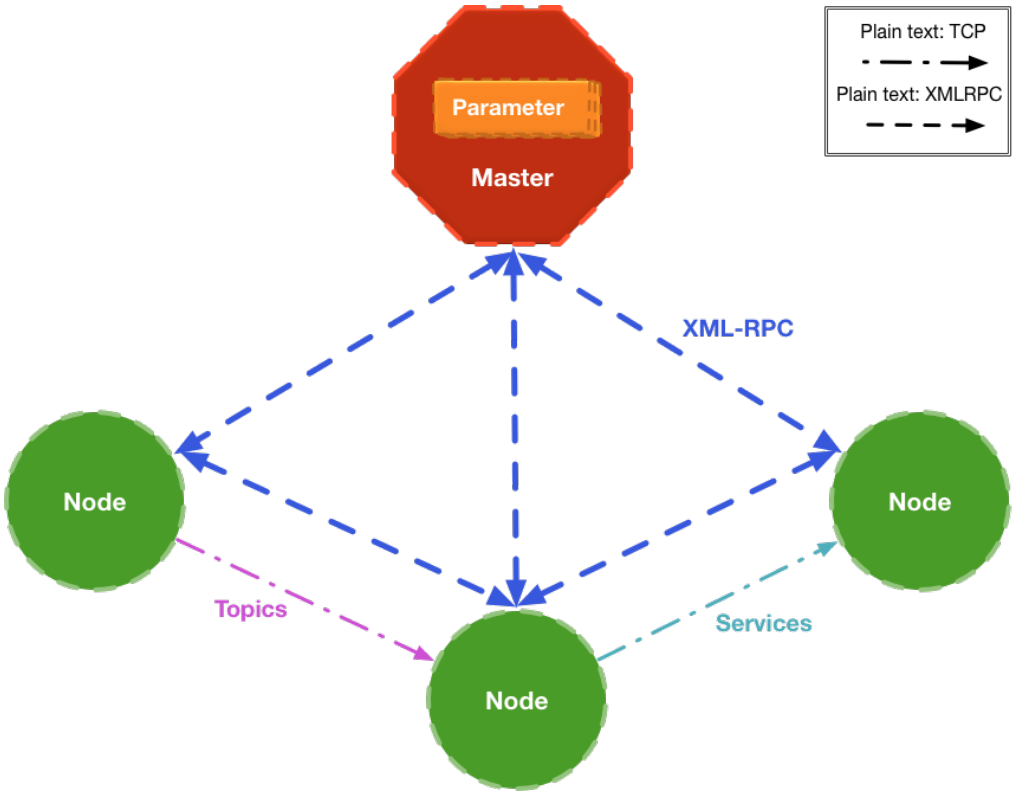

Figure 3: ROS Graph

2.4.1 *Establishing a topic connection*

In the next table, the sequence by which two nodes begin exchanging messages in case of topic communication is depicted:

Operation	Protocol
- Subscriber starts: reads its command-line remapping arguments to resolve which topic name it will use. - Publisher starts: reads its command-line remapping arguments to resolve which topic name it will use.	Remapping Arguments
+ Subscriber registers with the Master + Publisher registers with the Master. + Master informs Subscriber of new Publisher for the wanted topic. + Subscriber contacts Publisher to request a topic connection and negotiate the transport protocol. + Publisher sends Subscriber the settings for the selected transport protocol.	XMLRPC
- Subscriber connects to Publisher using the selected transport protocol.	TCPROS/UDPROS

Table 2: Topic Connection

2.4.2 *Establishing a service connection*

Services can be viewed as a simplified version of topics. The difference is that whereas topics can have many publishers, there can only be a single service provider. However, the master accepts as service provider the most recent node to register with the master. An advantage of using services instead of topics is that service client doesn't have to be a ROS node but it can also be an ad-hoc external application (i.e. a script).

Table 3 depicts the operation on services, according to the corresponding protocols.

Operation	**Protocol**
- Service registers with Master	XMLRPC
+ Service client looks up service on the Master + Service client creates TCP/IP to the service + Service client and service exchange a Connection Header + Service client sends serialised request message + Service replies with serialised response message.	TCPROS/UDPROS

Table 3: Service Connection

3

ROS 2

ROS is in use since 2007. At the moment a new robot specifying system is under construction, namely ROS 2 [36]. The goal of OSRF developers is to design a new set of APIs that will improve - considering the experience gained from the communication graph Section 2.1.3 - the first-generation of APIs. Hopefully, the final result will be a new ROS that still embraces the key concepts (distributed processing, anonymous publish/subscribe messaging, RPC with feedback, etc) but will support new kind of projects.

The changes described below could be integrated into the existing core ROS code; although considered, this solution has been rejected since these intrusive changes could probably brake legacy application support and amongst other things been a burden in the evolution of the project. In fact, the idea is to slowly abandon ROS 1.0 applications and start a new types with ROS 2.

Furthermore there are other straightforward reasons for a fresh start; in fact, if we analyse the main points on which the ROS 1.0 has been design in the first place, we can easily see the limits:

- single robot project
- workstation-class computational resources on board
- no real-time requirements
- excellent network connectivity (wired)
- applications in research prototyping
- maximum flexibility, with nothing prescribed or proscribed

From the first iteration of ROS, we observed a huge evolution of IoT/robotics devices. This translate in new challenges and requests from developers and industry. By crawling on the developers forum or by mining the repositories it is easy to spot the new trend of developers:

- Teams of multiple robots: although physically possible with the current implementation of ROS it is still tricky to build a multi-robots upon the single-master structure
- Small embedded platforms: the limits of a workstation-class computational resources on board is simply huge. In the IoT era the focus needs to be put on bare-metal micro controllers and/or micro computers with limited performance or power constraints (i.e. raspberry pi)
- Support for Real-time systems
- Non-ideal networks: design new solutions to increase the number of network types in which ROS can be deploied
- Production environments: develop tools to seamlessly transport research prototype to real-world application with ease
- Prescribed patterns for building and structuring systems: create new tools for features such as life cycle management and static configurations for deployment in the market

In addition to the devices evolution, nowadays we can count also to a bigger number of open-source framework projects. Back in 2007, the OSRF developers had to develop from scratch the whole systems for discovery, message definition, serialisation, and transport. However, with the introduction of software interfaces it's possible to build a modular solution that allow the interoperability of external software for auxiliary functions that are non-robotics-specific. In practice, they plan to demand to other projects as: Zeroconf, Protocol Buffers, ZeroMQ, Redis, WebSockets and in general other DDS, the task of develop

auxiliary features.

This is a huge advantage for the software security enhancements that we are proposing in this thesis. Furthermore, by means of these external frameworks, OSRF developers are able to achieve some remarkable benefits. First of all, they need to maintain less code that is directly developed by other 'groups'; in addition, they can exploit an indirect advantage by horizontally included software framework that is the possibility of using features that has been introduced in the external framework and also build ad-hoc solutions for each project independently. For example, if a device resources are limited and the project doesn't need a certain feature, it's possible to get rid of the unnecessary software layer (downgrade to a 'core' version of ROS) and lightening the solution.

3.1 STRUCTURE OF THE MIDDLEWARE

In this section we want to go a little more in detail about the structure of a middleware in ROS 2; let's start from the fact that OSRF developers wants to maintain the same core 'concepts' of the current implementation in ROS 2. This means that the client communication libraries will maintain the same publish/subscribe mechanisms used by the custom ROS protocols TCPROS and UDPROS.

However, in order to develop such mechanisms, despite the usage of different middleware solutions (e.g. DDS), we need to introduce a middleware API abstraction layer. This middleware interface defines the API between the ROS client library and any specific implementation.

In detail, each implementation of the interface will have a thin *adapter* which maps the generic middleware interface to the specific API of the middleware implementation. However, for the sake of simplicity, apart from the scheme

below, we will consider the adapter and the actual middleware implementation as a whole.

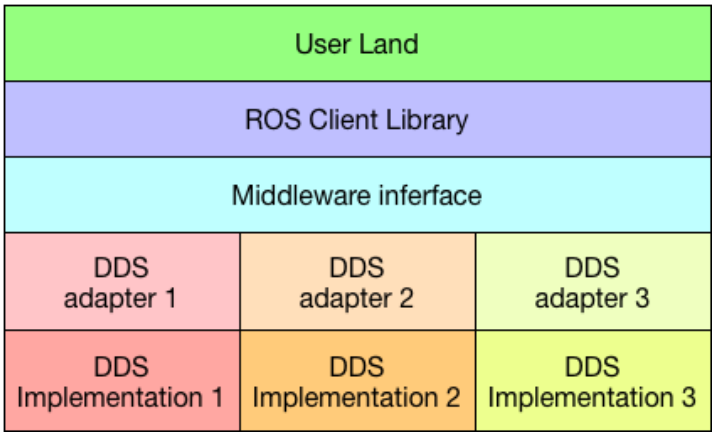

Figure 4: ROS Middleware Adapter

As such, the final solution will not expose any DDS implementation specifics to the user.

Since the goal of this solution is to provide to users a high level independent abstraction of the system, we need to define an abstraction that allow us to interface back and forward with the selected middleware infrastructure.

As we can see in the pictures below, the ROS client library 'above' the middleware interface works only on ROS data structures. In fact, since we will continue to use ROS message files to define the structure of these data objects and ROS library for user land code, it's needed to develop mechanisms to convert the ROS data objects provided from the client library into the suitable data format of the DDS. In reverse, the DDS data format needs to be converted in ROS data object and returned to the ROS library. These operations are all handled by the middleware interface.

Although logically complex, this solution allows us to independently develop ROS software that potentially works with all the possible combination of 'back-ends' and allows OSRF to update the system without braking applications.

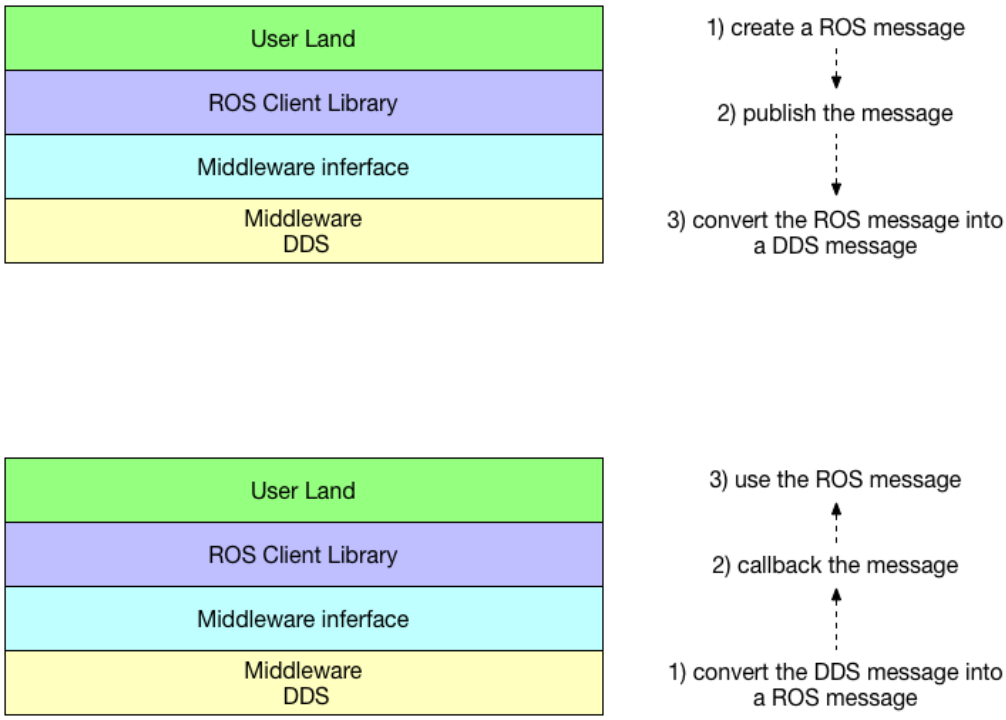

Figure 5: ROS Middleware Communication

This mechanisms needs to be implemented in both publishers and subscribers; they must agree on one specific middleware implementation library. The used middleware implementation library will only use the corresponding type support from each message package and ignore any additionally available type supports. Only middleware implementations can be selected for which the corresponding type support generators have been available when building the message packages.

4

SROS

SROS is a set of security enhancements for ROS, presented at the 2016 ROSCon conference by White and Quigley [**16**]. SROS aims to secure ROS API and ecosystem by means of native TLS/SSL support for all IP/socket level communication. In addition, with the usage of x.509 certificates, White and Quigley defined chains of trust, namespace node restrictions and permitted roles, as well as user-space tooling to auto generate node key pairs, audit ROS networks, and construct/train access control policies. Furthermore, they defined AppArmor profile library templates, that allow users to harden or quarantine ROS based processes running at linux OS kernel level.

Let's see below in detail how SROS tackles the security problems from both the ROS Computational Graph level and the Filesystem level.

4.1 COMPUTATIONAL GRAPH

As said in Chapter 2, ROS nodes communicate in the graph through the usage of XMLRPC APIs and transport libraries such as TCPROS and UDPROS; however, in these infrastructures all network traffic is transmitted in clear text. Additionally, as said in Chapter 1, in ROS there are no means to verify that the payload information was unaltered in transit.

This makes ROS particularly prone to sniffing, man-in-the-middle attacks, and other vulnerability as described in Section 1.1.

Considering that, SROS propose to leverage on the cryptographic protocol SSL the communications on the *computational graph* in the form of additions to ROS library. In detail, SSL is a widespread protocol used in applications such

as https web browsing, email, instant messaging, etc, that aims primarily to provide privacy and data integrity between two communicating entities and, in general, it ensures privacy, integrity and identity; in detail, all the communications are private and secured by symmetric cryptography, furthermore the communicating entities can authenticated by means of public-key certificates and, since the messages are signed, also prevent alteration of the data during transmission.

These addition to ROS libraries allows users with preexisting code a seamlessly use of security features. In fact, these additions don't affect, apart some connections overhead, the legacy applications.

However, we could achieve similar security results with the usage of Virtual Private Network (VPN) or Software-Defined Networking (SDN).

There are several scenarios in which these solutions are applied: corporate policy, nested networking complexity, or simply for deployment and maintenance limitations.

Nevertheless, creating a solid VPN solution can be very tricky. Mistakes made in these areas can give attackers a high bandwidth connection directly into your internal network [**18**]. Instead with SROS, we can ensure that our ROS application's network traffic remain secure even amongst these challenging constraints.

4.1.1 *Public key Infrastructure*

We said that by means of SSL we ensure security from network threats, even from attacks inside a local network, simply encrypting all ROS communications. However, with the usage of a Public Key Infrastructure (PKI) we can provide an additional layer of security with the introduction of *authenticity*.

The application of x.509 certificate for authentication is not new in the field of Web server applications [**9**] and following roughly the same structure, White

and Quigley [**16**] defined a Public-Key infrastructure for ROS with the definition of an ad-hoc certificate authority (CA) and by enhancing the libraries API calls as necessary.

We know that in PKI, CAs are responsible of issuing new certificate and manage the expiration and revocation by means of the certificate revocation list (CRL); in SROS this task are handled by the *keyserver*.

Physically speaking, the keyserver is not a node itself, nor is it tied to any particular ROS node and thus we can run the keyserver independently from the ROS instance. This means that we can decouple the certificate generation from ROS and achieve a better security resource management; in fact, by means of the keyserver, we want to generate the authentication certificates for the entities on the ROS graph, which is why the possibility of tampering it represents a huge threats for the system. Since these certificates are signed, any attempt to mutate or elevate a node's own permissions will void the CA's signature, and thus fail during the TLS handshake for incoming and outgoing connections; however, if a malicious user is able to compromise the keyserver, then it's able to overstep the policy profile security layer and freely generate certificates.

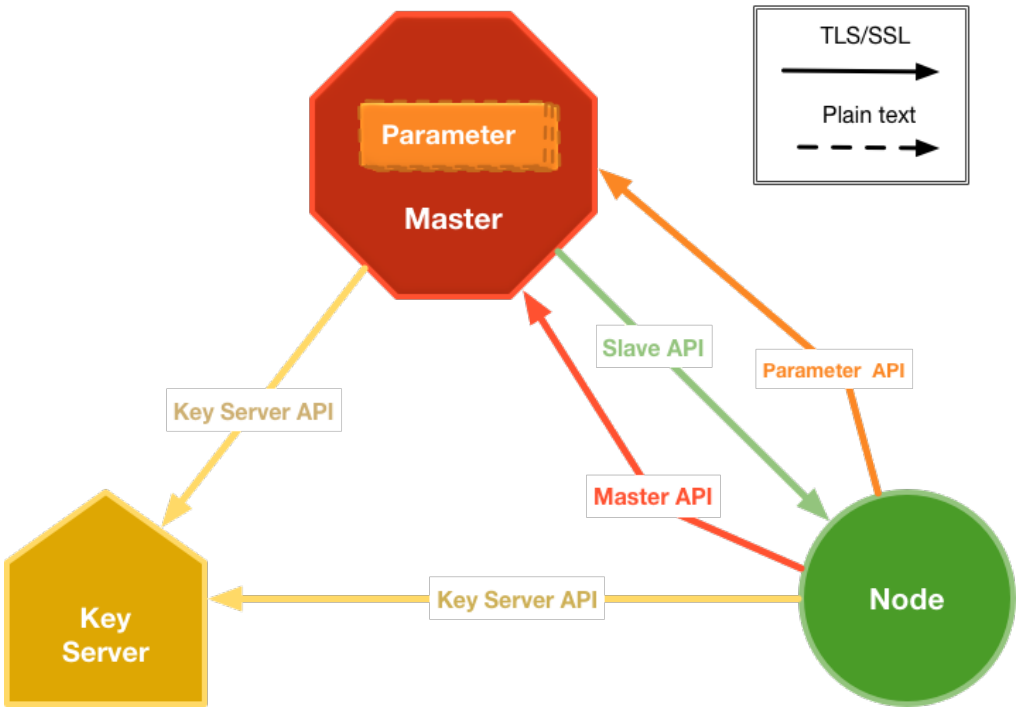

Figure 6: SROS Keyserver

However, how could we summarise access information on ROS graph and embed them into certificates?

We know that ROS uses namespaces to define locations and aspects of ROS's computational graph, such as message topics, services, parameters, and the resolvable names of nodes themselves. By leveraging on this, we can generate an access control list (ACL) of the operations allowed for a node by simply auditing the graph network through security ROS logs and events. Then, we can embed the resulting ACL as policy definitions in X.509 certificate extensions. However, we believe that usage of these logs only limits the way in which SROS works and therefore is one of the problem faced in this thesis (see Chapter 6).

Let's briefly see how SROS sets up the certificates and generate the whole infrastructure for a general ROS application: when the system start, the keyserver

load the designated configuration file. Obviously, since this is the first time that we run our application, there are no certificate yet; then, it define its own certificate and the ROS '*core*' nodes certificates (e.g. rosmaster, rosout, etc). From now on, if a node start under SROS - since it execute the wrapped libraries -, it checks the existence of its own certificate; if it hasn't one, it will attempt to connect via TLS to the keyserver and request - by providing the node's own namespace - its public certificate and private key. Thereafter, when it needs to establish a connection to another node it will provide its own certificate and they will establish a secure connection.

4.1.2 *Access Control*

Until now, we analysed how the SROS authentication phase is defined, however we overlook on how the ACL is generated and physically embedded in the certificates.

First of all, we need to analyse how the policy profiles are built. As previously said, in order to generate the profiles, we need to audit the execution of a '*training*' code, that exercises the entire scope of the application and generates SROS standard actions and events logs. Leveraging on them, SROS master defines a *yaml* configuration file that is parsed by the keyserver at load time; this file contains for each node a list of topics, services and parameters that the node has accessed in the training phase.

The advantage of using an auxiliary policy configuration file is the possibility of define, in a human readable syntax, the access permissions. In this way, a user can manually edit the profiles and granulise the access to resources by tighten a rule or by adding some other ones.

Once we have achieved a suitable profile setup, we can execute the keyserver that will generate the certificates. We want to embed the profiles as certificate

extension; by following the recommendations of the PKIX standard we can use an object identifier (OID) for each object in the certificate.

In detail, we define an OID as a string of numbers identifying a unique object such as a certificate extension or a company's certificate practice statement. By means of this standard OIDs, we can achieve interoperability between organisations on a shared network or, as in our case, a common platform.

In SROS, White and Quigley defined the OID as an 11 digits number that identifies an operation. In detail, we use the first 10 digits to qualify the operation and the 11th, depending on the value, specifies if the operation is allowed or denied for the object's value expression. This expression itself is a regex operation that defines the namespace scope of that action.

The usage of a regex expression is particularly useful in ROS because it allows us to define a globbing style to scope paths and, in addition, is human readable for auditing purposes.

However, this way of generating profiles presents some complexities, since flawlessly defining a policy profile that encapsulates all of the expected exchanges in a ROS graph would be prone to errors and a daunting task manually. Then, our goal is to provide some advanced mechanisms to ease the generation and security of the policy for ROS.

4.2 FILESYSTEM

SROS implements passive security solutions for ROS node from the Filesystem level too. These solutions tackle the security flaws from the operating system point of view. Considering that ROS packages come from varying sources, and vetting each dependency in this high-level stack for vulnerabilities can be impractical and costly.

The easiest solution is to leverage security from the OS's features, by the application of an access control mechanism to resources. We analyse this solution in Chapter 5. We can already mention the existence of third-party solutions as AppArmor.

This is a proactively software layer that implements mandatory access control (MAC) directly in Linux kernel, it's well documented and user-friendly, making it a suitable choice for ROS.

By means of AppArmor we can protect as well quarantine ROS processes. This helps us to ensure that malicious or dysfunctional nodes are restricted in the degree of disruption if they perform unauthorised operation. With the usage of access control in ROS nodes, we are able to mitigate zero-day-exploits that enable attackers to gain shell access or invoke local privilege escalation, as well as quarantine unknown malfunctions during run-time such as accidentally overwriting another node's log files.

For example, one may wish to reserve a dedicated communication channel to a specific node type, and would like to prevent any other nodes from clobbering the same channel. In this case we could amend the policy profile for specific node executable as to be only permitted node capable of reading and writing to the reserved device.

In order to use AppArmor, SROS provides a profile library for it composed of modular primitives to quickly build custom profiles for ROS nodes; using the SROS profile library, users can focus on defining MAC pertinent to the application as: prescribing nodes access to busses, access resource permission, operating system resource usage, etc.

Part II

SOFTWARE ENHANCEMENTS

In this part, we present our analysis and definition of the enhancements. Currently SROS provided users a method to auto-generate policy profiles for access control within the ROS computation graph and API via learning by demonstration. This enables users to quickly audit and record all access events of a ROS system by running it through its paces and compressing these exchanges into a concise human readable policy profile. This access control policy profile should not be confused with that of SROS's AppArmor profile library or the profiles they provide using features from Linux security modules. Those help users provide Mandatory Accesses Control (MAC) for ROS nodes on the runtime processes level of the hosting operating system. The policy profiles we are discussing below are the ones related to enabling access control for ROS nodes in the ROS graph network level.

These policy profiles are eventually used by other SROS tools, such as the keyserver, that then considers these when generating and signing PKI elements, such as the X.509 certificates that are given to ROS nodes, as well as the policy extensions that are embedded in those certificates. Those certificates extensions are then interpreted by connecting peers when deserting whether the other peer has sufficient permissions for the requested ROS API action.

However, at this stage, sroscore is solely responsible for both logging the security related access control events and auto-generating/amending the current policy profile. We believe this is suboptimal and that generating/amending access control policy profile should be done using the security related events that SROS logs.

5

ACCESS CONTROL

Access control is one of the most critical point in today's Information Technology [3]. With the term "Access Control", we address the way in which users can granularly access resources in a system (file, directory, etc). Access control can take many forms: we can determine whether a user has rights to use a resource, or what in the system may be constrained, i.e. when and how the resource may be used. For example, a user (or group of users) might only be permitted access to a certain file, while simultaneously being denied access to some network feature.

During the years, several solutions have been proposed and studied. However, in this chapter we will evaluate only two types of access control [1] [2]: *Discretionary Access Control* (DAC) and *Mandatory Access Control* (MAC).

Nowadays, most of the operating systems use DAC. This technique refers to the owner the task of sets access permissions on a resource. However, this approach presents several problems. One problem is due to the fact that we are demanding to a user the task of setting the correct setup of permissions; whether he sets them wrong, he exposes the resource to unwanted operations/interactions. Furthermore, by design, we have an indirect threat: the superuser. In fact, it's possible to compromise the entire system hijacking a process that runs with superuser privileges.

Instead, with MAC, access permissions are represented in form of global security policy to which all the users of the system are conditioned. As such, is mandatory that the superuser - the one that manages the policies - account is only used in maintaining the global security policy; furthermore, all the day-by-day tasks are made by accounts without the permission to modify the policy.

Therefore, we exploit the possibility of attacking the system through hijacking processes, even though, we still need to protect the superuser account from other types of attack.

5.1 TERMINOLOGY

In order to better understand the concept behind access control, we need to introduce some terminology. In fact, any access control model can be stated formally using the notions of *users, subjects, objects, operations*, *permissions* and the relationships between these entities [5]. The term *user* simply refers to people who interface with the system. An instance of a user's dialog with a system is called *session*. A process acting on behalf of a user is referred to as a *subject*. Furthermore, an *object* is any resource accessible on the system. An *operation* is any active process invoked by a subject and, lastly, *permissions* are authorisations to perform some action on the system (often represented as a combination of objects and operations) [5].

5.2 MANDATORY ACCESS CONTROL

The major drawback of this technique is the complexity of the configuration and maintenance of the policy. In fact, the creation of a global profile requires a detailed knowledge of the behaviour of the system and limits the set of possible usages for new users.

In order to ease the configuration and maintenance of MAC policy by system administrators, during the years, several solutions have been proposed. In particular, we consider `SELinux` and `AppArmor`. It is important to notice that, even though we are under MAC, the linux DAC still apply.

5.2.1 *SELinux and AppArmor*

Security-Enhanced Linux (SELinux) is a Linux feature that provides a variety of security policies for Linux kernel developed by NSA [4]. SELinux evaluates all the operations - that are not blocked by DAC - against its own security policy before allowing it. In particular, it considers all the actions as subjects, in contrast with the Linux DAC model, in which objects are files or directories. In SELinux we consider as objects also other processes and other system resources in both kernel space and user land. This allow us to implement more granular and precise rules for all the entities of the system; however, it also introduces complexity in the setup phase.

In the other hand, we have AppArmor, a proactively software layer that protects the operating system and applications from external or internal threats, by enforcing good behaviour and preventing even unknown application flaws from being exploited.

It is important to notice that both of them are based on a default deny principle; this means that, whenever a rule is not specified, the default behaviour will be to deny the operation. However, there are slightly differences in the way in which this principle is applied. In SELinux we need to specify the whole system permissions beforehand, otherwise all the not defined applications will be blocked. Instead, in AppArmor, we define what a single application can and cannot do, without an implementation of the complete security paradigm. Practically, it means that AppArmor has a default deny approach to the single immunised application but a default permit approach to the rest of the system: once you decide an application is protected by AppArmor, the default deny principle applies to it, while anything that is not immunised is simply not considered by AppArmor, following the default permit approach.

We need also to notice the flexibility that we can achieve with these tools. In fact, it is possible to implement *Mandatory Access Control* (MAC), *Multi-Level*

Security (MLS), *Multi-Category Security* (MCS), *Role Based Access Control* (RBAC) and *Type Enforcement* (TE). Even though they are comparable, it is important to notice that SELinux, that operates by *label*, is safer than AppArmor (that works by *paths*). In practice, SELinux match for each application a label that operates globally in the system per user. On the other hand, AppArmor works on the absolute path of the application binary - a new instances in a different path is not secured - for all the users on the system.

All in all, considering the pros and cons of both the presented solutions, we are able to move on and build upon ROS an hybrid solution that takes into account what we discussed in this chapter.

6

SECURITY LOGGING

In this chapter we discuss about the design of logging in SROS. In particular, we standardise the security logging format by adding more structure that would render it more parsable and informative. In addition, we afford the users more information when debugging/auditing profile policies, but also SROS developers a more defined logging feedback to build tools around such as: interactive GUIs and CLI tools. In order to do so, we adopt some of the conventions from AppArmor and SELinux. As we previously said in the background, it's important to notice that currently SROS security events are logged by the srosmaster process. In the next chapters we discuss also of a solution that allows us to decentralise this into the srosclient level to enable all nodes the ability to log security events from evaluating connecting peer certificates, not some lookup from a local policy file. This would afford users more granular logs from individual nodes when debugging/auditing policy profiles.

As discussed in the previous chapter, we are aware that AppArmor is the most user-friendly solution and then the best way to start.

Let's recall from the Novell website [**6**] the definition of AppArmor:

AppArmor is designed to provide easy-to-use application security for both servers and workstations. Novell AppArmor is an access control system that lets you specify per program which files the program may read, write, and execute. AppArmor secures applications by enforcing good application behaviour without relying on attack signatures, so it can prevent attacks even if they are exploiting previously unknown vulnerabilities.

As such, we prefer to focus on AppArmor instead of the more complex SELinux and analyse it's logging format; then, we give an overview of ROS API and the definition of the new logging format.

6.1 APPARMOR LOGGING

In AppArmor once a profile is defined and loaded, we can operate with it in two different mode [7]:

- complain: in complain mode, violations of AppArmor profile, such as the profiled program accessing files not permitted by the profile, are permitted but also logged. As such, we are executing the application without restricting or confining it. Then, the resulting log can be post-processed with the AppArmor tool `aa-logprof` to transform log events into improved profiles;
- enforce: in enforce mode, on the other hand, violations of AppArmor profile rules are logged and not permitted;

In order to profile a program, we need to introduce an other tool `aa-genprof`; this simply generates a basic profile and asks the user to refine this profile by executing the application in *audit-mode*. This particular mode, although not listed in the official modes listed above, is an internal mode of AppArmor that logs all the events of the application; this logs are then used by the tool in order to generate the policy. Lastly, for the record, we mention the auxiliary *disable* mode that simply disables the selected policy profile.

In detail, when a process tries to to access a file it does not have permission to access, in audit/complain mode we log messages similar to:

```
audit(1386511672.612:238): apparmor="DENIED" operation="exec"
parent=7589 profile="/tmp/sh" name="/bin/uname" pid=7605
comm="sh" requested_mask="x" denied_mask="x" fsuid=0 ouid=0
```

```
audit(1386511672.613:239): apparmor="DENIED" operation="open"
parent=7589 profile="/tmp/sh" name="/bin/uname" pid=7605
comm="sh" requested_mask="r" denied_mask="r" fsuid=0 ouid=0
```

As we can see, permissions request by a process are described in the `operation` and `requested_mask/denied_mask` under POSIX syntax. In addition to, the "name" and process ID of the running program, as well as the profile name including any `hat` - local profile with some additional restrictions and an implicit rule allowing to be used to transition to them - that may be active, separated by "//" [**8**]. As said before, in case we are executing the application in audit mode we also have messages about allowed operation as:

```
audit(1386512577.017:275): apparmor="ALLOWED" operation="open"
parent=8012 profile="/usr/bin/du" name="/etc/apparmor.d/tunables/"
pid=8049 comm="du" requested_mask="r" denied_mask="r" fsuid=1000 ouid=0

audit(1386512577.017:276): apparmor="ALLOWED" operation="open"
parent=8012 profile="/usr/bin/du" name="/etc/apparmor.d/tunables/"
pid=8049 comm="du" requested_mask="r" denied_mask="r" fsuid=1000 ouid=0
```

These are mandatory in order to define with aa-genprof the profile of the application. Now, we have a better understanding of the way in which AppArmor operates and we can move on with the ROS API analysis.

6.2 ROS API LOGGING

As we saw in the previous section, in order to log the operations a suitable choice is select a core set of attributes that appoint unambiguously a certain application in the system. In ROS we are able to achieve this result exploiting the concept of `namespace` that has been introduces during ROS analysis **??**. In detail, any resource in the system is appointed by a unique name. Furthermore, considering that we don't want to revolutionise the way in which ROS developer interact with the existing log infrastructure and we want to insert our

solution upon the existing structure, we need to maintain the following logic:

[node][verbosity] datetime: logging string

In detail, each entry can be dissected in two macro-blocks:

- Context:
  - node name: logging node of origin
  - verbosity: info, debug, warning, error
  - date time: *yyyy-MM-dd HH:mm:ss,fff*
- Info: logging string

Below we can see an example of the bootstrapping log of ROS:

```
[rosmaster.main] [INFO] 2017-01-25 18:47:43,225: initialization complete,
waiting for shutdown
[rosmaster.main] [INFO] 2017-01-25 18:47:43,225: Starting ROS Master Node
[xmlrpc] [INFO] 2017-01-25 18:47:43,226: XML-RPC server binding to
0.0.0.0:11311
[xmlrpc] [INFO] 2017-01-25 18:47:43,226: Started XML-RPC server
[http://GLaDOS:11311/]
[xmlrpc] [INFO] 2017-01-25 18:47:43,227: xml rpc node: starting XML-RPC
server
[rosmaster.master][INFO] 2017-01-25 18:47:43,227: Master initialized:
port[11311], uri[http://GLaDOS:11311/]
.........
.......
.....
```

Leveraging on that, we propose a standard that maintains the compatibility with the legacy tools of ROS. As such we define each component as follow:

- `node`: denotes the name of the agent that is generating the entry;

- `verbosity`: we exploit this variable and use it for representing the mode of the execution:
  - DEBUG: the entry has been added in audit mode.
  - WARN: the entry has been added in complain mode.
  - ERR: the entry has been added in enforce mode. In this case we need to log as an error since in enforce mode the rules block the execution of the operation and report the action.
- `datetime`: is the current data in usual form *yyyy-MM-dd HH:mm:ss,fff*
- `logging string`: represent the log message

Lastly, we use the verbosity level INFO for logging the current *mode status* of the system. As example, if we start SROS in audit mode, we record in the log the entry:

```
[rosmaster.master][INFO] 2017-12-31 12:34:56,789: sros="STATUS"
operation="runtime_mode" mode="audit"
```

Although we want to keep the usage of the tool as a high level as possible, in this phase we need to log low level. In practice, this means that instead of using a "mask" for the operation permissions, we can directly log the API call and specify the nature of the entry. As such, we can log every API call with a triple: *(node, operation, path)*. Where the node is the user that is calling the API, the operation is simply the name of the function and path appoint the resource that the operation affect. As example, we can consider the following API calls - in audit mode -, from a `talker` for the registration as new Publisher of the topic 'chatter'; and a `listener` that wants to connect to the topic:

```
[rosmaster.master][DEBUG] 2017-12-31 12:34:56,795: sros="AUDIT"
operation="registerPublisher" node="/talker" resource="topic"
path="/chatter"
```

```
[rosmaster.master][DEBUG] 2017-12-31 12:34:56,850: sros="AUDIT"
operation="registerSubscriber" node="/listener" resource="topic"
path="/chatter"
```

7

POLICY PROFILE SYNTAX & GENERATION

In this chapter we discuss about the standard syntax for the profile policy file itself. SROS already leverages a good deal from AppArmor functions such as it's globbing syntax. However, rather than the yaml format SROS uses now for policies, we provide a more applicable syntax intended to encode policy rules, definitions, and relationships. In the following two points, we proceed with the analysis of the profile syntax of AppArmor and then we give an overview of the ROS APIs and we propose a new standard for the abstraction of them. Thereafter we discuss about the APIs division mechanism and the definition of tools for introspect the recorded security logs and simplify the user steps to compress and auto-generate policy profiles from those events. Similar to AppArmor's CLI tools, we define `sros-genprof` and `sros-logprof` tool that read security events from SROS logs and guide users on auditing and defining what access control rules to include and what scope they should be applied. This alleviate the need to have a train mode for srosmaster, as any policy file modifications would be done offline by a separate CLI tool and not a ROS node. This also would improve security, as we could then avoid the AppArmor policy expectations that sroscore would have needed to write to the policy file used by the keyserver.

7.1 APPARMOR SYNTAX

In order to analyse AppArmor Syntax we need to consider two main concepts: *permissions definition* and *profile structure*.

7.1.1 *Permission access mode*

In AppArmor we define file permission access with the usage of several modes. This modes are made of a combinations of the following:

- read (r): allows the program to have read access to the resource.
- write (w): allows the program to have write access to the resource. We need this permission if we want to unlinked (remove) a file.
- discrete profile execute (px): this mode requires that a discrete security profile is defined for a resource executed at AppArmor domain transition. However, If there is no profile the access is denied. We need to notice also the variation `Px` of this command that executes the command in a scrubbed (clean) environment.
- unconstrained execute (ux): allows the program to execute the resource without any AppArmor profile applied to the executed resource. Notice that it requires listing execute mode as well. As seen above, we have a variation `Ux` of the command that execute the command in a scrubbed (clean) environment.
- inherit execute (ix): prevents the normal AppArmor domain transition when the profiled program executes the named program. Instead, the executed resource inherits the current profile.
- allow executable mapping (m): this mode allows a file to be mapped into memory. In addition, it makes the pages executable. It is used on some architectures to provide nonexecutable data pages, which can complicate exploit attempts.
- link (l): it mediates access to hard links. When a link is created, the target file must have the same access permissions as the link created

7.1.2 *Profile structure*

Thanks to what we learned in the previous section, we can now analyse how a profile is defined. We consider as example the toy profile for `flush`:

```
# profile to confine postfix/flush
#include <tunables/global>

/usr/lib/postfix/flush
{
  #include <abstractions/base>
  ...
  capability setgid,
  ...
  /usr/lib/postfix/flush                      rix,
  /{var/spool/postfix/,}deferred              r,
  ...
  /{var/spool/postfix/,}flush                 rwl,
  ...
  /{var/spool/postfix/,}incoming              r,
  ...
  /{var/spool/postfix/,}public/qmgr           w,
  /etc/mtab                                   r,
  /etc/postfix/main.cf                        r,
  /etc/postfix/virtual.db                     r,
  @{HOME}/.forward                            r,
  /proc/stat                                  r,
  /proc/sys/kernel/ngroups_max                r,
  /var/spool/postfix/pid/unix.flush           rw,
}
```

Let's see in detail all the components: in the first row, we have a comment naming the program that is confined; although not mandatory, it helps when an administrator needs to take note on a profile by hand. After that, we found an include statement that loads a file containing variable definitions that ease

the creation of the profile. As we can see, we have already introduced the concepts of *comment* and *include*. In practice these entities, are diversified by the space in between the hash and the first letter of the row.

After the definition of all the necessary includes, we can proceed with the capability rules; as a rule of thumb, we start the macro-block of a program profile with the absolute path to the program (line 4). Then we can define, into curly braces, the includes of other profiles as well as the paths and capability entries (i.e. at line 6 we have an include that pulls in components of AppArmor profile and below a capability entry that enable all the POSIX.1e draft capabilities).

Above, we said that AppArmor supports globbing syntax; thanks to that feature, we can create entries which match regexp formulas. In order to be acceptable a regexp formula needs to be enclosed into curly braces (see line 11 to 17). In addition, we introduce the concept of a variable (line 21). The convenience of a variable is that we can modify it's value without changing the value for each entry of the profile.

7.2 ROS

Before we dive into ROS logging format, we need to briefly recall the main entities of the computational graph:

- Master
- Parameter Server
- Nodes
- Topics
- Services

In the current implementation the parameter server, topics and services are embedded in the master node. As such, we can leverage a good deal working

on master level only, keeping the node logging extension as a future integration. In fact, in order to communicate with each other, each node needs to connect to the master through API calls. It's still a suboptimal solution, since we are not logging direct connection between the nodes, but the usage of the profiles it's enough to guarantee that a node performs only the authorised operations.

In AppArmor's profile we define the capabilities of a program with an absolute path and a permission mask. However, a similar system is not applicable for ROS entities; in fact, we can give the same name to a node, a topic, a parameter and a service simultaneously. So, we need a way to unambiguously specify which resource we are addressing. A trivial solution is directly specify the type of the resource as first parameter in the entry.

By analysing the different APIs on the master, we can notice that we can identify three different group of resource that needs to be handled: `parameters`, `topics`, `services`. These macro-groups, allow us to define an high-level concept of the API calls; our final goal is to provide an easy-to-use tool for the users and separate them from the low-level implementation that can mutate without undermine the defined security profile. However, we give a more detailed explanation of the APIs partition in the following chapter.

At this point, we are able to unambiguously address a resource in the graph. The next step, is defining the list of the possible permission modes. We can leverage a good deal from the previously defined AppArmor modes and define for each resource group a couple of mask:

- Parameter (*param*):
  - `read` (r)
  - `write` (w)
- Topic (*topic*):

  - `subscribe` (s)
  - `publish` (p)

- Service (*service*):
  - `call` (c)
  - `execute` (x)

In addition to them, we can introduce also an extra symbols `deny` to granularly manage the access to the resource. In fact, a drawback of using a limited number of modes in comparison with the number of possible low-level APIs, is the granularity of the permissions that needs to be integrated. However, we analyse in detail this point in the next chapter.

At this point we can proceed with the definition of the profile syntax, the idea is to leverage our standard from AppArmor. We know that in ROS we are able to address unambiguously an agent in the graph via `namespace`. On the other hand, a resource needs a prefix. All in all, we can follow the definition of AppArmor profiles and produce the following solution:

```
/namespace
{
#include role
resource /scope masks
}
```

As we can see, we introduce the concept of *roles*. With this variable we can determine the role of the agent in the graph: `master` or `slave`. We further explain this concept and the resulting policy consequences in the following chapter. As such, we have all the necessarily information for the definition of a profile.

Let's recall the example that we previously introduced of a `talker` that wants to register itself as a new Publisher of the topic 'chatter' and a `listener` that wants to connect to it.

One of the resulting profile could be:

```
/talker
{
#include <ros/slave>
topic /chatter p,
}

/listener
{
#include <ros/slave>
topic /chatter s,
}
```

As said, this is only one of the possible solution, in fact we are permitting a bigger set of operations to the nodes that can be further reduced. However, this is default - and easiest - solution for the final user that potentially can further granulise the resources access as needed with the usage of the command deny.

7.3 API PARTITION

In this section we dive into the low-level definition of ROS. Our goal is to logically divide two-fold the available APIs. In the first division we identify the level in the graph where the functions work, whether `master` or `slave`; then, we match the functions with the respectively resource group (parameter, topic, service). However, in addition to master and slave, we need to introduce a third logical level `parameter`.

This logical level should not be confused with the homonym group element 'parameter' that qualify the resource; instead in this case we are addressing the *parameter server* API calls. We need this additional level because even though parameter server is currently physically located in the master, it is intended as a separate entity in the network. All in all, we can divide the APIs in this three groups as depicted in Table 4 (see Appendix A for in depth functions definition).

Parameter	Slave	Master
setParam	getBusStats	registerService
getParam	getBusInfo	unregisterService
hasParam	getMasterUri	registerSubscriber
deleteParam	shutdown	unregisterSubscriber
	getPid	registerPublisher
getParamNames	getSubscriptions	unregisterPublisher
SearchParam	getPublications	
	paramUpdate	lookupNode
subscribeParam	publisherUpdate	getPublishedTopics
unsubscribeParam	requestTopic	getTopicTypes
		getSystemState
		getUri
		lookupService

Table 4: ROS API division

This logical division gives us an high level overview of ROS's communication pattern. Based on the position of the functions we can easily identify the entities that need to communicate in the graph: master and slaves. As such,

we define the include statement *role*, previously introduced in profile syntax definition, namely:

- #include <ros/slave>
- #include <ros/master>

These include statements ensure to an agent the ability of inherit a set of 'role' functions. In particular, only a `master` has the ability to access the slave APIs. The only exception, is the function `requestTopic` that gives the ability to a 'subscriber' to negotiate a communication protocol with a 'publisher' and needs to be used by both master and slaves.

However we still need some mechanisms that allow us to granulise the information disclosure and limits the operations that a agent can perform using this API.

As we will see below, there are several APIs that share the same security *issue* of requestTopic. As such, we introduce some mechanisms to improve our control on the resource consumption. Our goal is to enhance the server-side API. In detail, we need to define some constraints that we will apply before the function's invocation and on the function's response. Depending on the level of severity that we want to achieve, we perform on the functions the following checks and sanitisation:

- `Called ID`: the caller ID name must respect the scope (either the certificate subject name or the alternatives).
- `Resource Check`: the request must respect the resource scope (the operation must be authorised in the policy profile of the agent).
- `Response Sanitisation`: if needed we 'trim' the response in order to not disclose sensitive information in graph-scale analysis API (e.g. getSystemState).

As example, let's consider a *slave* call of the function `requestTopic`. We need to perform this function only if the agent (slave) have the right permissions to

connect to the topic. In practice, this means that a slave that invoke the function can query only topics that has been authorised in his policy profile. We further analyse this restrictions in the following section. See in Table 5 the recap of the Slave API division and permissions.

Slave API	Permission	Caller ID	Resource Check	Sanitised
getBusStats	master			
getBusInfo	master			
getMasterUri	master			
shutdown	master			
getPid	master			
getSubscriptions	master			
getPublications	master			
paramUpdate	master			
publisherUpdate	master			
requestTopic	master $\cup$ slave	✓	✓	✓

Table 5: Slave API

7.3.1 *Master API*

As we know from ROS analysis, the master manage all the connections between the nodes. In fact, when a node needs to perform any operation, either topic or service related, it needs to query the master.

In the following we want to discuss the set of operations that are associated with the masks described in Section 7.2.

First of all, it's useful to give a general idea of topic and service in ROS: we can picture the `publisher` of a topic as a radio broadcaster and the `subscriber`

as the passive program listener. On the other hand, we can describe services as a remote customer service; in practice, we have a technician that `execute` the service and a client that `call` for assistance, all based on request/reply interaction. Since we want to add a security level in the transmissions, as a rule of thumb, the radio broadcaster and the technician need a mechanism to check - before performing the task - if the caller/spectator have the right permission to enquire the resource. We already achieved a suitable solution with the usage of the embedded access control lists (ACLs) in the certificates. However, this simplistic check of agent's permission is not enough for some of the master's operations. So, we need to analyse the previously introduced concept of sanitisation on the API response.

The problem arise from the fact that we can distinguish two group of APIs, a first that perform an operation and a second that return an information (e.g. *requestTopic*) . As said, the first group of APIs is already secured with policy profiles, but the latter needs some additional mechanisms. Our goal is to enhance the behaviour of these APIs by changing the returned value based on the requesting agent; the idea is to intercept with a wrapper the response and modify it as needed.

Let's consider the example of *getSystemState*: this particularly powerful operation, returns a list representation of the system state (i.e. publishers, subscribers, and services). Considering that we don't want to disclosure information to unauthorised agents, we need to modify the response of this API in accordance to the caller agent. This means, that instead of receiving the complete state of the network, the agent is able to 'see' only the agents that are present in his policy. The same applies for the remaining functions that roughly follow the same idea.

See Table 6 for an overview of the master operations.

Master API	Permission	Caller ID	Resource Check	Sanitised
registerService	execute	✓	✓	
unregisterService	execute	✓	✓	
registerSubscriber	subscribe	✓	✓	
unregisterSubscriber	subscribe	✓	✓	
registerPublisher	publish	✓	✓	
unregisterPublisher	publish	✓	✓	
lookupNode	pub ∪ sub	✓	✓	✓
getPublishedTopics	subscribe	✓	✓	✓
getTopicTypes	subscribe	✓	✓	✓
getSystemState	slave	✓	✓	✓
getUri	slave	✓	✓	
lookupService	call	✓	✓	✓

Table 6: Master API

7.3.2 *Parameter API*

In parameter server, we store all the parameters that are declared in the ROS graph. Contrary to the APIs that we analysed so far, parameter API are the more similar to the resource addressed in AppArmor. This means, that we can abstract these API call as file operations: set value, delete variable, search etc. So, we can straightforwardly define the access control on them following the idea of a discrete access control (see Chapter 5) with the usage of the mask permissions `read` and `write`.

As in the previous groups we have an overlapping area of permissions (*getParamNames* and *SearchParam*). These functions allow a broad search in the

server. In particular the first one get a list of all parameter names stored on the server, instead the latter search for parameter key through all the namespaces. These represents a huge threats for critical application because it expose all the variables that are stored on the server. The proposed solution, follow directly the one proposed for the other similar APIs, namely a sanitisation on the response based on the agent security policy. See Table 7 for the overview of the functions permissions.

Parameter API	Permission	Caller ID	Resource Check	Sanitised
setParam	write	✓	✓	
getParam	write	✓	✓	
hasParam	read	✓	✓	
deleteParam	read	✓	✓	
getParamNames	read ∪ write	✓	✓	✓
SearchParam	read ∪ write	✓	✓	✓
subscribeParam	read	✓	✓	
unsubscribeParam	read	✓	✓	

Table 7: Parameter API

7.3.3 *XML-RPC connection*

Lastly, we discuss about the communication APIs used by ROS in the graph. As said, ROS base its communication protocol on XML-RPC, a remote procedure call (RPC) protocol which uses XML to encode the calls and HTTP as transport mechanism.

Since our goal is to provide a secure solution and management for ROS, we can't overlook these transport APIs. In fact, if necessary, a possible attacker can use this primitive and establish a direct connection simply avoiding the

master calls and the controls that we have defined. So, the idea is to enhance this APIs too; in practice, we need to add to this operations the previously defined controls for topic and service. At the end of the day, it means that we add the mechanisms of the caller ID match and the certificate resource check. All in all, in this way we can block all the entities that are not able to provide a correct certificate to establish the connection. We can see the overview of the APIs in the following Table 8.

Topic Transport	Permission	Caller ID	Resource Check	Sanitised
connect_topic	publish	✓	✓	
accept_topic	subscribe	✓	✓	
Service Transport	**Permission**	**Caller ID**	**Resource Check**	**Sanitised**
connect_service	execute	✓	✓	
accept_service	call	✓	✓	

Table 8: Transport API

7.4 PROFILE AUTOGENERATION

Finally, in this section we analyse how we can transform the defined SROS logs into certificate extension's policy profile. In order to do so, we want to define some tools to automate the creation and refinement of profiles. Our goal is to provide an easy to use software, that allow users to create suitable certificates without a great deal of effort. As previously done, at first we give an overview of the AppArmor's tools and techniques; then, we define the counterpart tools and techniques that we design for ROS.

Let's start from the idea of profile creation, in AppArmor there are two ways to approach this problem: *stand-alone profiling* and *systematic profiling*.

The first, as the name suggest, is a method suitable for profiling small client applications like mail clients; the latter, instead, is used for profiling more programs at once and for profiling applications that run for long times such as network server applications.

These approaches are characterised by a set of CLI tools that allow the user to define profiles by demonstration. First of all, we introduce a tool called *aa-genprof*. This utility, by means of the auxiliary tool *aa-autodep* creates an approximate profile of the application on the base of the invoked libraries (via the command `ldd`); then, it set the provisional profile into complain mode, reloads AppArmor, marks the log with an informative status message and prompts the user to an interactive interface for the execution of the program and guide the exercise of its functionality.

Once the user have exercised the application functions as possible, it can stop the learning phase and apply, in complain mode, the resulting profile to the system.

In fact, thanks to the audit mode we have logged all the files and directories to which the program requires permissions in order to function properly, therefore we can build a first version of the profile.

However, as we could imagine, the learning phase is not enough to explore all the application's functionality, as example, it may happen that after some time we need to access new resources because the behaviour of a program drift in other execution branches not explored before.

In order to overcame this problems, AppArmor defines an additional tool *aa-logprof* that review the output found in the logs and generate new entries in AppArmor security profiles. In detail, when we execute aa-logprof, it first scan the log files from the last aa-genprof '*status*' entry and if there are new security events, that are not covered by the existing profile set, it gives suggestions for modifying the profile.

By leveraging on the AppArmor approach, we discuss about the definition of a set of CLI tools for SROS profile generation; one of our goal, as said, is to alleviate the need of fully concentrate the generating and management of the profile in the keyserver. As such, we define two external tools for the generation and enhancement of profiles, namely: *sros-genprof* and *sros-logprof*.

These tools follows directly from the AppArmor counterpart; however, in spite of the two different profile technique, in SROS, we want to implement only systematic profiling.

In fact, contrary to AppArmor, that allow the existence of no-profiled applications, in ROS we can't allow the existence of nodes without certificates, otherwise we are not able to sustain the SSL/authentication infrastructure.

Let's review how the tools works; we could consider the same example of a general ROS application given in Section 4.1.1: at first we need to run our application with sros-genprof, that will start the master - in place of the command *sroscore* - in audit mode and logs all the actions and events of the computational graph. Once we are satisfied with the degree of exercise we can stop the learning/audit mode and start the creation of the profile. Below, we can see in detail how sros-genprof work.

```
$ sros-genprof
Connecting to master.....

Writing log entries to /home/user/.ros/log/
Creating SROS profiles in /home/user/.ros/sros.d.

Please start your applications in
other windows to exercise the graph functionality now.

Once completed, select the "Scan" button below in
order to scan the system logs for SROS events.

For each SROS event, you will be given the
```

```
opportunity to choose whether the access should be
allowed or denied.

[(S)can system log for SubDomain events] / (F)inish
```

The creation of the roles follow directly from the tables pictured in the previous section; this means that if in the log a node publish a topic `foo`, in the policy profile we insert an entry in which we specify that the node can *publish* that topic: `topic /chatter/foo p`.

However, it may happen that we have several similar resources in the same namespace or we want to extend a rule to other similar scope; for example, we may have topics `foo` and `foo1` in chatter namespace. If the node publish both topics the system advice the user to use globbing syntax, which means that we can define regular expressions with the following syntax:

- * : represent any number and any characters in the current namespace
- ** : represent any number and any characters including the definition of sub-namespaces
- ? : represent a single character or number
- [abc] : represent the the single character `a`, `b`, or `c`
- [a-c] : represent all the character in between `a` and `c`
- {ab,cd} : expands the string to match the expressions `ab` and `cd`

As such, when the log entry of `foo1` is analysed, the program prompt us some choices. The first is the safest one that simply adds a new entry for the topic, then we have a formula that defines the previous published topics with the addition of `foo1`; afterwards, we have two broadly formulas that consider all the topics with the prefix `foo` and as suffix a single or any character/number.

```
Profile:     ros/talker
Access mode: publish
```

```
Topic:       /chatter/foo1


 [1 - topic /chatter/foo1 p,]
  2 - topic /chatter/foo{ ,1} p,
  3 - topic /chatter/foo? p,
  4 - topic /chatter/foo* p,

(A)llow / [(D)eny] / (I)gnore / Audi(t) / Abo(r)t / (F)inish
```

At this point, a user can also manually audit the formulas and, in addition, can define `deny` rules for other topics in the same namespace. For example we could have a set of topics in the namespace *chatter* that share the same prefix `foo` but we want to define a rule that match all these topics but exclude `foo2`. One possible solution could be the insertion of a deny rule as:

```
....
deny topic /chatter/foo2 p,
topic /chatter/foo* p,
....
```

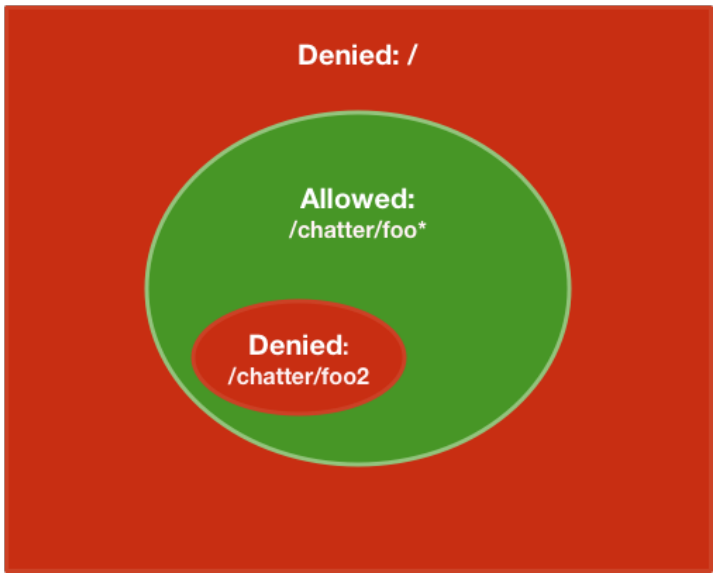

Figure 7: Globbing syntax

Obviously, we can apply globbing syntax across the entire scope; for example, if we want to access all the resource named as *temp_sensor* in the graph we can straightforwardly define the rule: `resource /**/temp_sensor mask`.

That said, until now we discussed about the definition of profiles for a new ROS application, however we need to consider a second scenario in which we have already defined a set of certificates but we need to enhance the policy profiles to allow new operations or limit the agents permissions.

A trivial solution to this problem could be to reuse the previously defined sros-genproof and regenerate from scratch the certificate using the new applications in the training phase; having said that, for long-lasting applications, or simply complicated deployment, in which we are not able to measure the size of the graph or all the possible execution 'paths', we are not able to give in one shot the whole structure, therefore, we build our knowledge during time.

However, this kind of approach leads to the generation of a lot of logs that make the profiles generation an incredibly time consuming task; in fact, if we always consider the entire log when we need to update the profiles, we spend a lot of time just recomputing the same entries. In order to overcame this problem, we implement the same ruse used by AppArmor developer of leaving a mark in the log that point out from which line of the log we need to start in the next 'scan'.

So, following the same idea of sros-genproof we define *sros-logprof*; as the AppArmor counterpart tool, sros-logprof scan the log files produced in complain mode by the master and if new security events that are not covered by the existing rules are founded, it gives suggestions for modifying the profile. Let's see an example of the execution below: in this case, we consider the graph previously defined, in which we have two topics in `/chatter` namely `/foo` and `/foo1` and we want to insert a new one *foobar*:

```
$ sros-logprof
Reading log entries from /home/user/.ros/log/
Updating SROS profiles in /home/user/.ros/sros.d.
```

```
Complain-mode changes:

Profile:     ros/talker
Access mode: publish
Topic:       /chatter/foobar


 [1 - topic /chatter/foobar p,]
  2 - topic /chatter/foo{ ,1,bar} p,
  3 - topic /chatter/foo* p,

(A)llow / [(D)eny] / (I)gnore / Audi(t) / Abo(r)t / (F)inish
Adding topic /chatter/foobar p, to profile.

.......

= Changed Local Profiles =

The following local profiles were changed. Would you like to save them?

 [1 - ros/talker]
  2 - ros/listener
  3 - ros/rosmaster
(S)ave Changes / Save Selec(t)ed Profile
[(V)iew Changes] View Changes b/w / (C)lean profiles / Abo(r)t
Writing updated profile for ros/listener.
Writing updated profile for ros/rosmaster.
Writing updated profile for ros/talker.
```

As we can see, the tool scan the log until it found a new entry that doesn't match any other rules; however, since there are other topics with the same namespace and prefix name it tries to insert the new one as a combination of the three topics rules or simply with an independent entry as the previous one.

Once we are satisfied with the resulting policy, we forward the updated profiles to the keyserver that issues the new certificates as already defined in SROS Chapter 4.

All in all, we can summarise the workflow for autogenerate the profiles for a ROS application as follow:

1. if we don't have previous certificates: start *sros-genproof* to put sroscore in complain mode, otherwise go to point 5
2. run through the app paces such that all the events are logged
3. follow the suggests for policy amendments
4. new policy are defined and saved - new certificate are issued
5. put sroscore in complain mode and run applications
6. SROS violations are logged
7. start *sros-logprof* to audit and amend the rules
8. repeat steps 4-7 until we are pleased with the results
9. final profiles are set to enforce mode

Part III

PROPOSITIONS

In this part we discuss about propositions to improve the current security managements and integration of ROS in heterogeneous networks.

8

CERTIFICATES

In this chapter we discuss about different approaches for the definition of the relations between the certificates and policy profiles. As previously discussed in Chapter 4, SROS statically defines the policy profiles in the extension field of each x.509 certificate. In addition to the current implementation, we want to discuss about other statical solutions and introduce also dynamic ones. First of all, we need to define the two different operational models for distribution and usage of the certificates [9]:

- user-pull architecture
- server-pull architecture

These models are respectively user-based and host-based. Both of them are made of the same components already defined in SROS: nodes and key-server/attribute server.

In the current SROS solution, the certificate authorities (CAs) are simultaneously key-server and attribute server; their purpose is to issue the certificates that are used by the nodes in the graph. However, as we will see later, this behaviour isn't always safe. Furthermore, it's important to notice that all the techniques that we are going to define need the policy profile already defined via sros-genproof and sros-logproof. Let's see below in detail the two architectures and some applications.

8.1 USER-PULL ARCHITECTURE

As the name suggests, in this mode the user pulls the attributes certificate bundle from the server and stores it locally. Then it uses the certificate for

the authentication phase with the other agents in the graph. This family of solutions exploits the problem of the authentication leveraging on the integrity services offered by the certificates by design. In fact, the certificates are issued by the CAs which are `trusted` entities in the system. However, in addition to the current implementation, in which we store all the information inside of the x.509 certificate, we want to introduce additional types of certificate that are defined in the x.509 standard: *Identity certificates* (ID) and *Attributes certificates* (AC)[**11**].

While X.509 public-key certificates (PKC-ID) bind an identity and a public key, an attributes certificate (AC) doesn't contain a public key but may contain attributes that specify group membership, role, security clearance, or other authorisation information associated with the AC holder. The reason why we should use AC for authentication information in place of PKC extension is manly for two reasons:

- authorisation often does not have the same lifetime as the identity and the public key. So, when we store authorisation information in a PKC extension, the general result is that we are shortening the PKC useful lifetime.
- the CA that issues the PKC is not usually authoritative for the authorisation information. In fact, it represents a threat for the system that should be avoided.

Indeed one of the advantages of using AC instead of bundle information on PKC is the support for short-lived as well as long-lived certificates. Typical short-lived validity periods might be measured in hours, as opposed to months for PKCs. Short validity periods allow ACs to be useful without the usage of the certificate revocation list (CRL) that adds overhead in the certificate verification.

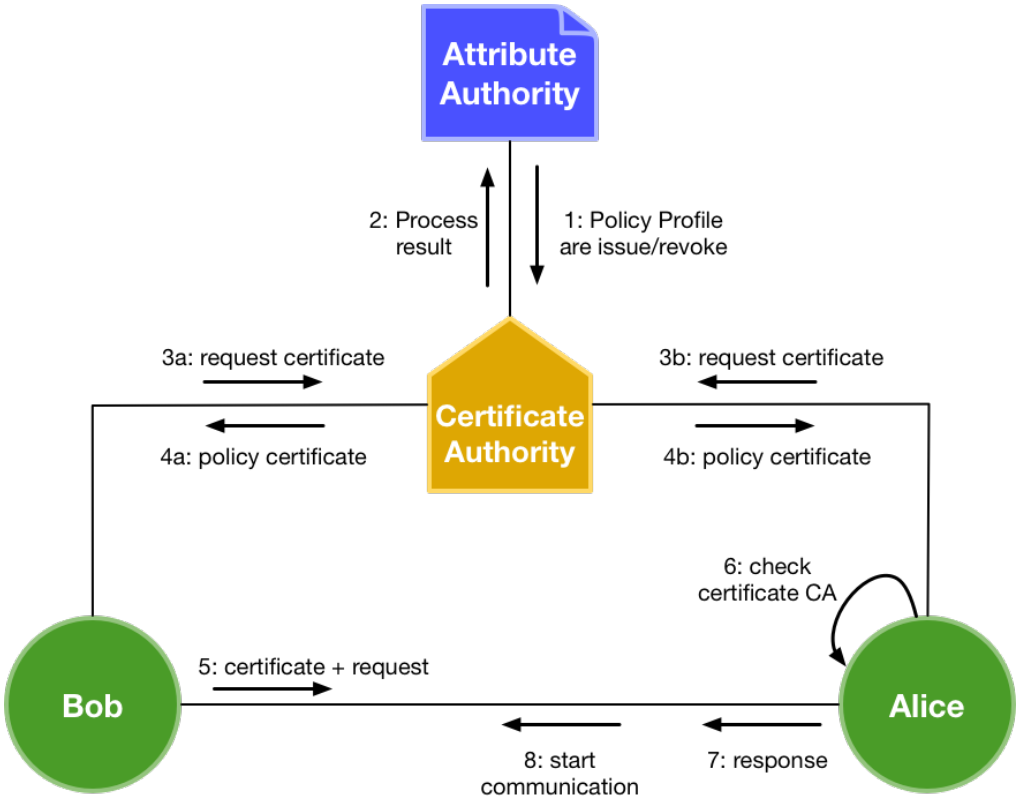

Figure 8: User-Pull model

However, it's important to notice that AC needs PKC to work. In fact, authorisation information needs to be bound to an identity. In order to do so, the attribute certificates binds attribute information to the ID certificate's subject. There are several ways in which we can bound the authorisation with the identity certificate, below we presents three different approachs [**10**]: *monolithic, autonomic* and chained signatures.

8.1.1 *Monolithic*

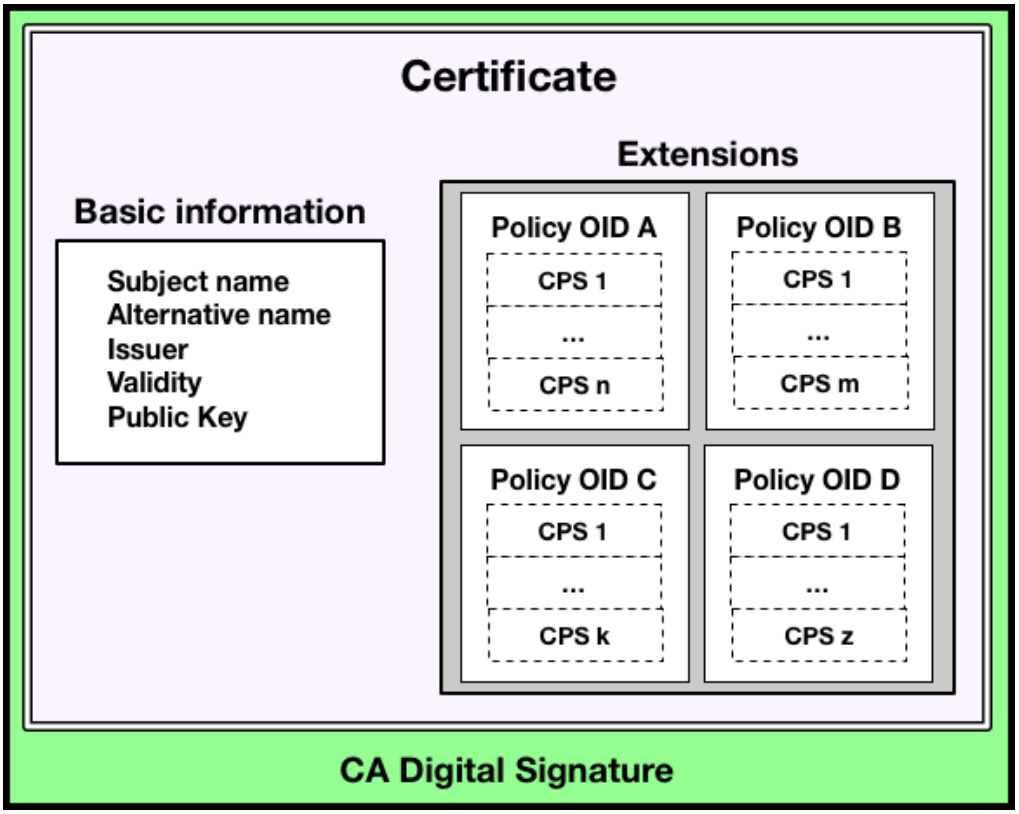

Figure 9: Monolithic certificate

This is the easiest solution in which we consider only one certificate authority in charge of both identity certificate and attribute. The resulting certificate is the one used by SROS at the moment. In fact, in the monolithic approach we create a certificate that holds both the identity and attributes information; trivially, this is implemented by using x.509 and the extension fields.

The resulting certificate *tightly coupled* the identity information and the attributes with a single signature. This means that in order to change an information in an existing certificate we need to revoke the previous and issue a new updated one. However, the management of this particular solution is simplified in comparison of the other, since we need to trust only one CA.

Considering that all the certificate information are verified by the only CA's signature and therefore there is only one CRL that needs to be checked.

However, as previously said, this approach has several drawbacks. First, multiple CA's are not supported. We can't revoke a certificate if we aren't the issuer CA and there is the possibility of issuing multiple certificate with different attributes for the same agent. Secondly, due to the design of the solution we are not able to maintain different life-time for multiple attributes; in fact, all the attributes share the lifetime of the PKC.

All in all, monolithic approach is the most favourable solution when we are limited in terms of resources or we are looking for a statical solution, even though we sacrifice flexibility in maintenance.

8.1.2 *Autonomic*

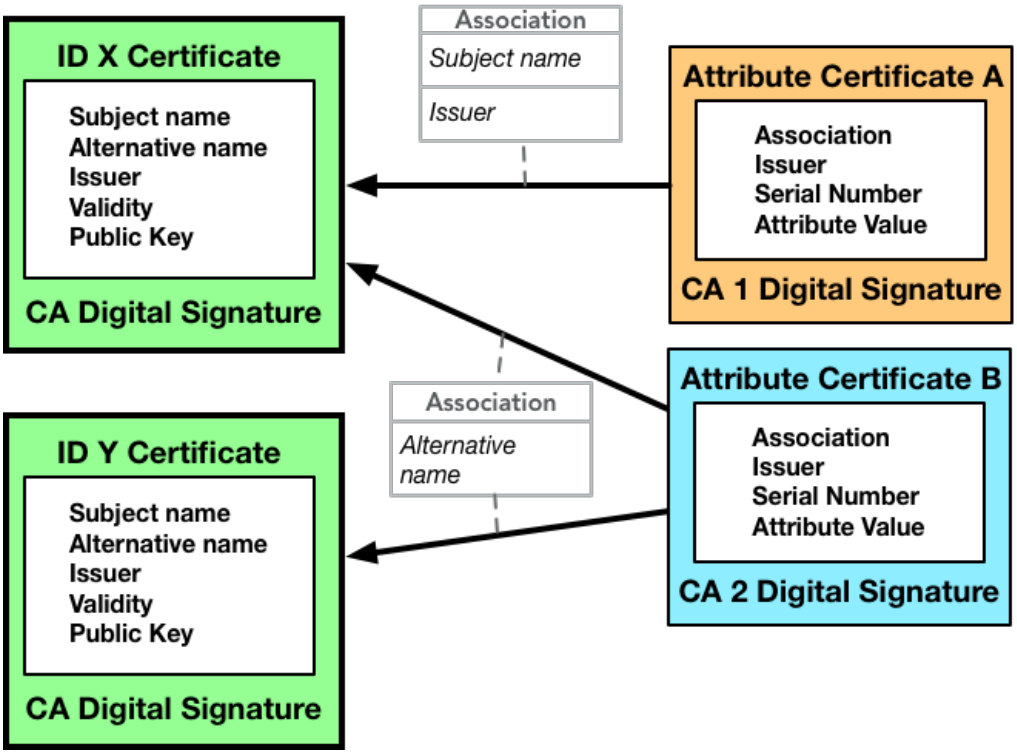

Figure 10: Autonomic certificate

In this case we introduce the concepts of multiple CAs and we differentiate between identity and attribute certificates. With this approach we want to define a *loosely coupled* binding between the ID certificate and the AC. This particular solution allows the existence of multiple ID certificates per agent provided that there is a injective function from the certificates to the agent; this means that we will never have more than one agent that corresponds to an ID certificate. As such we can bind each AC with a different set of information from the ID as: subject's name, public-key, certificate serial number, etc.

Depending on the chosen set of information that has been selected we can modify the certificate issuing a new one and still maintain the correlation between the AC and ID as long as the binder information has not been changed.

As example, if we insert in the ID certificate the unique serial number of the agent and we bound the attribute certificate based on that, we can change the other information such as lifetime, serial number, subject's name, while the link between the certificates holds.

All in all, as we can see with autonomic signature we achieve the higher reusability of the ID certificate and offer to agents a more flexible mechanism for obtain attribute certificates. However, since we moved from a static solution to a dynamic one, we should be extra careful about the new threats as the choice of the information set. In fact, even though we have an injective function from the certificates to the agent there aren't constraint on the information that are stored in the certificates. It means that if we accidentally choose a common set of information, it may happen that the same attribute certificate can be used by unauthorised agents that share the set of information with the authorised agent.

8.1.3 *Chained signatures*

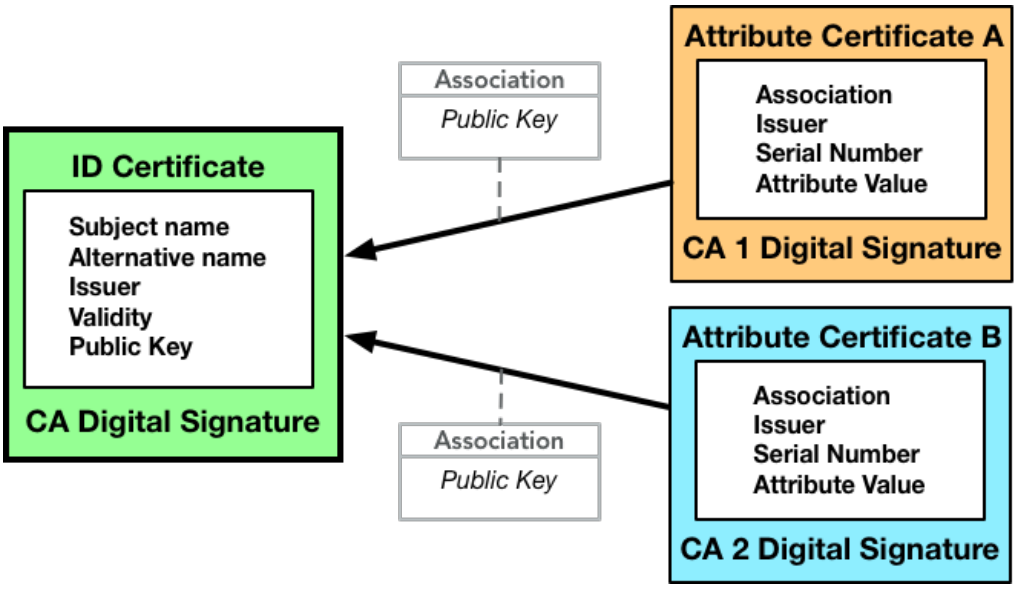

Figure 11: Chained certificate

Lastly, with chained signatures we want to exploit the previously introduced concepts of multiple CAs and different types of certificates by adding a *tightly coupled* binding between the two types of certificate.

With this technique we want to take the security features guaranteed by the monolithic solution and part of the flexibility of the autonomic approach and create a new hybrid solution. As in autonomic an agent can have multiple ID certificates issued by multiple CAs. However, instead of binding the attribute certificate with an arbitrary set of information, we bound on the digital signature of the corresponding ID certificate. In fact, if the information in the referenced ID certificate are changed (a new certificate is issue), the digital signature *should* change as well. It's crucial that the signature changes its value. That said when we work with cryptography signature this is can't be taken for granted; we need to be careful on the choice of the encryption algorithm and avoid the ones that should be considered discontinued (e.g. MD5). In fact,

with such algorithms it's possible to forge two (or more) certificates with different contents but with the same digital signature [12]. This makes it possible for an attacker to bypass the mechanism that links the AC to a unique ID and perform unauthorised operations with another ID certificate.

On the other hand, under the assumption that we are using a suitable signature algorithm, when we issue a new ID certificate (with a different signature) the link between the two certificates is broken and then the attribute certificate becomes automatically useless. One of the advantage of chained signatures is that we don't need to aggregate all the attributes according to the shortest lifetime of the certificate as in monolithic. Furthermore, we introduce a mechanism that allows us to share with other agents only the necessarily information. In fact, with a monolithic certificate all the policy of the agent are available in the PKC, instead with this solution we can share only the necessarily attribute certificate enforcing a new privacy feature and dynamic management of profiles.

8.2 SERVER PULL ARCHITECTURE

As we saw above, in user model we delegate to the agent the task of authenticate himself with the other agents via certificate. Instead, in server pull model the general idea is to demand the authentication phase to the attribute authority. Our goal is to define a dynamic solution in which apart from the AA no one needs to know the attribute information.

However, as in user pull model, we still need a method to unambiguously identify an agent in the graph; we can achieve this straightforwardly with the usage of the already defined ID certificates.

But instead of issuing attribute certificates and bound them to the ID, we store the attribute policy roles locally in the attribute authority. This particular solution allows us to implement in the AA whenever access control we want (e.g. MAC, DAC, MLS, MCS, RBAC), without modifications to the client (agents) APIs. We design a high level API that permits the agents to retrieve

the authorisation response from the AA regardless of the chosen access control method.

Let's see in detail how this high level API works: as example we consider the handshaking for resource access between two agents *Alice* and *Bob*. In this example, Bob wants to establish a connection to Alice and access one of the Alice's API. As we previously saw in user pull model, Alice needs to verify Bob's permissions policy and decide if allow or deny the operation. In user pull model, it was enough to check the right extension policy rules in Bob's PK certificate and proceed in accordance with that. Instead, in server pull, we query for a response the attribute authority.

Let's see in detail how this API works between Alice and the AA:

- Bob sends his request and public key (ID) to Alice;
- Alice needs to know if Bob can access the resource; so, she authenticate herself (via ID) with the trusted AA;
- once the TLS tunnel has been established, she submit through the API call Bob's ID certificate and the requested resource;
- the AA evaluates the request and replies to Alice with a response *true* or *false*, respectively if Bob is allowed to perform the operation or not.

There are several advantages on this model: first of all, thanks to the usage of AA instead of static certificate we can achieve a dynamic flexible solution that can evolve and change during run-time without additional setup. Secondly, we can maintain the secrecy of the sensitive information about the policy topology inside the attribute authority without compromising the network topology in critical applications.

Still, the solution introduces the problem of the *single point of failure* (SPOF). In practice, considering that all the agents need to query the attribute authority in order to receive a response about the permissions, if it fails the entire system will stop working. In fact, from the attacker point of view it will be enough to tamper the AA to compromise the entire system. In addition, we introduce by

design an overhead in the handshaking process; in fact, what was a straightforward local check of the profile in user model, became a remote request to the external attribute authority. However, this problem can be slightly mitigated by resuming previous sessions via `Session Ticket` as introduced in TLS 1.3 [13].

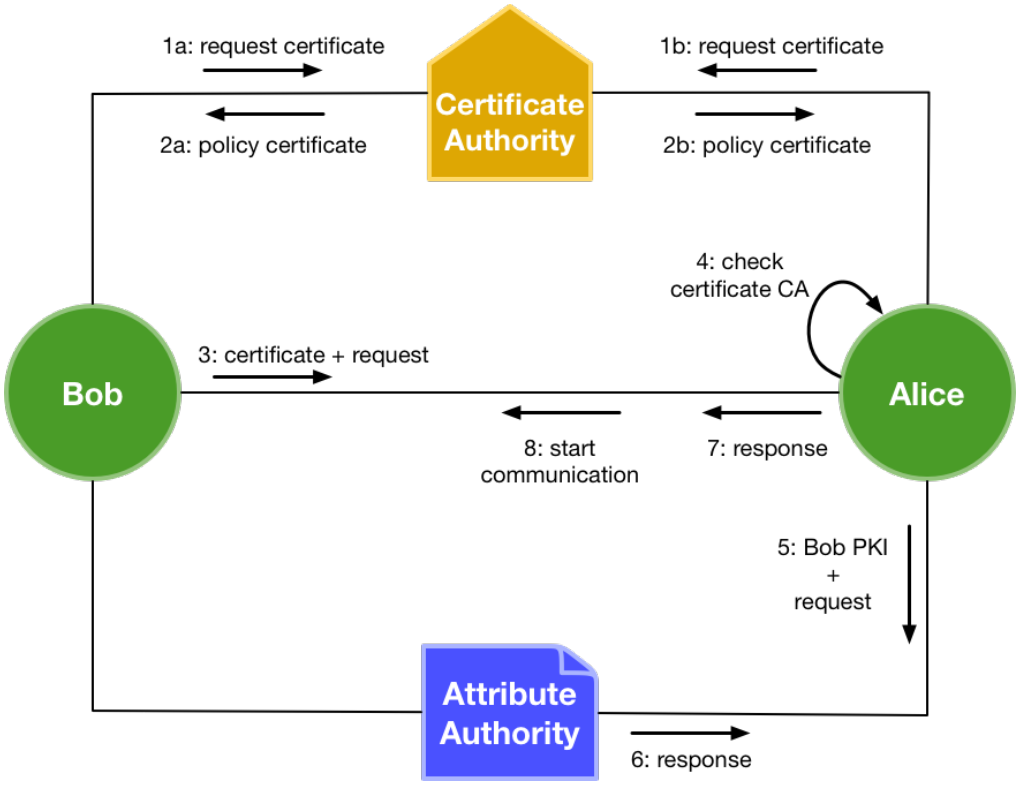

Figure 12: Server-Pull model

9

ONEM2M

OneM2M is the leading global standardisation body for M2M and IoT [14]. Established through the alliance of several standard organisations and industrial companies, it has the goal of creating a framework for facilitating the unification of different technology providing an interworking platform for devices and enabling reuse of existing software and hardware solutions.

What makes this solution appealing for ROS is that oneM2M doesn't want to standardise the whole environment across networks, applications and devices, but to provide a standardised interface to interoperate openly and securely with each other.

Similarly to the goal of ROS 2, oneM2M wants to unite under the same umbrella several different solutions with the concepts of *logical abstraction* and *semantics enablements*.

By means of these concepts, we are able to develop independent solutions for a broad number of devices. The oneM2M idea is to develop an 'universal interface' using semantics rules to match the existing devices API. Then, with the usage of a wrapper on the API calls they are able to integrate the device with their functions. In this way, similarly to our proposed solution, they don't need to tamper the devices in order to push new security features but they are applied at high level by the communication framework.

Leveraging on oneM2M, ROS could be deployed in any network. As said in Chapter 3, the usage of external solutions could enhance ROS with the introduction of all the back-end components and security features.

Furthermore, by means of the oneM2M framework, ROS developer could seamlessly develop solutions that could be used securely in complex system

as smart cities. A similar solution has been presented by Datta *et al.* in [**24**], in which they discussed the integration of their IoT framework into oneM2M standard architecture. In particular, they discussed about the challenges on the urban infrastructure in terms of public safety, resource management, co-operative mobility management and more. By putting particular emphasis on the increasingly use of the next generation information and communication technologies (ICT) commonly defined IoT.

Apart from the interaction with an heterogeneous network of devices, the integration of oneM2M standards could bring as dowry in ROS the security architecture components. The architecture consists of three different macro-layer:

- Security Functions layer: this layer contains a set of security functions that are exposed at reference point to the nodes. These are six different categories of security functions: Identification, Authentication, Authorisation, Security Association, Sensitive Data Handling and Security Administration.
- Security Environment Abstraction Layer: this layer implements various security capabilities such as key derivation, data encryption/decryption, signature generation/verification, security credential read/write from/to the Secure Environments, and so on. The security functions in the Security Functions Layer invoke these functions in order to do the operations related to the Secure Environments. In addition this layer also provides physical access to the lower Secure Environments layer.
- Secure Environment layer: this layer contains the secure environments that provide various security services related to sensitive data storage and sensitive function execution.

Leveraging on them, we could alleviate the need of develop in-house solution for storing sensitive information and for securing critical components

as certificate/attribute authorities in SPOF scenarios. Furthermore, oneM2M natively support solutions for the creation of RBAC access control and verification using end-to-end security of functions that could significantly mitigate the effort in manually sanitisation of ROS libraries API calls. Moreover, it also supports an advance type of dynamic authorisation for allowing the dynamical issue of temporary permissions and token structures for the creation of one-time authorisations.

All in all, we believe that the integration of the SROS features in oneM2M with the additional tools discussed in this thesis for the autogeneration of security profiles, could benefits the ROS growth and community more than the single integration of different DDS as discussed in Chapter 3. Furthermore it would broad the number of possible deployments of the ROS robots in real-world scenario, without the concern of pushing security update to the different robots software components during the robot life-span.

10

CASE OF STUDIES

The usage of encryption mechanisms and TLS/SSL connections in general IoT networks and in ROS include computation and resource costs. In this kind of applications in which we expect that the involved entities have limited computational resources, costly operations like public key encryption or digital signatures have a significant impact in the agility of the system. In this chapter, we will discuss with the usage of real-word examples, how the introduction of TLS and encryption affects overall the system performance.

10.1 CERTIFICATE OVERHEAD

As mentioned, information in an unencrypted channel are insecure, untrustworthy and easily interceptable.

We said that we can leverage a good deal for securing the network communication by means of the TLS protocol. Obviously, communicating over an encrypted channel introduces additional computational costs: first, we have the TLS asymmetric handshake, then, once a shared secret is established we have the symmetric encryption communication.

In general the most costly operation in a TLS connection is the initialisation via asymmetric handshake. However, instead of always using full handshake as depicted in Figure 13, we could use alternative techniques as the *session resumption* or *false start*. The first allows to resume a previous session thanks to the cached keys and connection information; the latter, speeds up the initial connection process by giving the sender a 'head' start on the receiver. In detail, a normal TCP connection forces one party to wait for a response from the

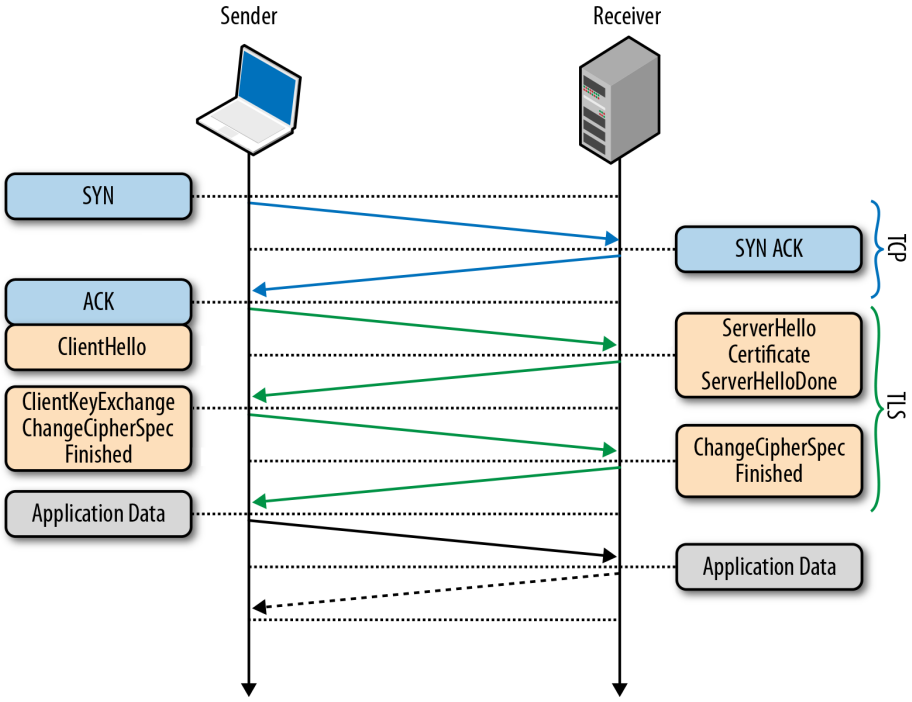

Figure 13: TLS handshake

other. Instead with this technique, the sender starts the transmission when the handshake is only partially complete; by the time the handshake is finished, data has already been transferred.

Once we have established a connection, since our applications are expected to work in real time, we need to measure how much overhead affects the system performance. This means that we need to evaluate how the encrypted connections performs in comparison of unencrypted ones. By simply comparing the two types of connection in web based applications we can observe, in mean, a difference of 5 ms between the two approaches in terms of response times, with a peak increase in CPU time of about 2% [**29**]. For example, in 2010 Google switched to using HTTPS in Gmail without deploying any additional machines or special hardware on their front-end machines, resulting with a CPU load increase of only 1% [**27**, **28**]. We can conclude that the usage of TLS doesn't effect overall the performance of the devices.

However, we need to contextualise, since we are sill considering workstation class computational resources. In terms of embedded devices in which we have limited resources, or simply energy constraint, we need to deploy ad-hoc SSL and TLS implements as MatrixSSL [**30**] that allow the usage of these techniques for low memory footprint IoT devices that require low overhead per connection.

By means of these techniques and tools, we can overcome the problem of certificate overhead [**31**], still we need to acknowledge the main problem deriving from the usage of encrypted communications: the cypher encryption and decryption cost. We tackle these problems, by analysing in the next section the computational and additional data overhead.

10.2 ENCRYPTION OVERHEAD

In this section we discuss about the performance of different encryption algorithms in ROS presented by Lera *et al.* in [**17**] and by Dieber *et al.* in [**26**].

TLS implements several cryptographic algorithms to secure connections. It's important to notice that most of these algorithms have dedicated instructions added in hardware in modern CPUs, that help them run substantially faster than software only ciphers implementation. So, depending on the chosen algorithm and key-length we can easily observe some changes in the performance.

In both the presented papers, the authors tested the performance when the format of data published by the node have been changed from plain text to an encrypted strings. However, they have chosen different approaches to the problem: Lera *et al.* decided to stress the performance of a ROS network in the worst possible scenario with 3DES algorithm; instead, Dieber *et al.* performed their test using Advanced Encryption Standard (AES) cypher.

As said before, the choice of the suitable algorithm and key-length should not be underestimate. For example, it's well known that 3DES performs worst

than AES [32]. As such, there are some tools available for benchmarking the different algorithms as Crypto++ [33] and useful guideline for the definition of a suitable key length [34].

With no surprise, from the data size point of view, the chosen algorithms without padding grows linearly in data traffic with respect of the number of messages. For example, in [17], the authors delivered 10 messages in plain mode and encrypted mode generating respectively, on average, 346 bytes in the first case and 420 bytes in the latter. The only negligible overhead that it's included is due to the encryption operation and some embedded information.

Instead, from the computational point of view things are more complicated. This concept is more clearer in [17] than in [26], since they performed a more in deep analysis of encryption messages. In detail, the usage of 3DES encryption algorithm that perform a triple data encryption is transposed in a quadratically grown of operational 'cost'. The majority of this cost is due to the encrypt/decrypt operations that are carried out for each message by the CPU.

This hypothesis is confirmed when we analyse the three different experiment that have been executed by the authors:

- *Hello world + timestamp*: this is the most basic operation, in which they encrypt, sent and decrypt small messages made of a fixed string and a timestamp. Since the messages are small, the cost of each encryption/decryption is limited and therefore, we can't observe a huge impact to the total performance (~1% of CPU)

- *Custom text*: in this case they send three different strings of growing dimensions, namely: 256kb, 512kb, 1024kb. Here we observe that the execution time remains similar to plain text execution, but there is an increase of ~60% of CPU usage with a pick of 98% for the bigger strings due to the encryption/decryption operations.

- *Video flow*: considering the previous CPU usage, in this final example they verify if the encryption/decryption operations introduce some delays in

the consumption of a 'real-time' resource as a security video. In this setup they achieve 14.56 frames per second in the publisher, and 14.54 frames per second in the listener.

All in all, we can conclude that the insertion of cypher capabilities in ROS doesn't effect the performance of the network but only the devices power consumption. Although we need to recall that the example proposed by Lera *et al.* considers the usage of a not optimised algorithm and that the performance may vary accordingly to the chosen algorithm and key-length.

11

CONCLUSIONS

We have presented a static mechanism to automatically generate and amend policy profiles for ROS, build upon SROS.

In order to determine a suitable solution we analysed the ROS framework and we discussed about the possible threats deriving from the environmental change from local networks to internet-scale connections. We introduced the logical infrastructure changes that OSRF wants to make on ROS 2, and we focused on the problems that are tackled by SROS and its weak points.

Once we had an overview of the threats and weak points of the whole infrastructure, we proceeded to the definition of our work that has been presented in three-macro blocks. In the first part we defined a new standard for ROS logs, that allows us to reconstruct all the library functions call-back that are not covered by the 'basic' log system, by means of a more accurate and defined structure. Then, we proposed a standard syntax for the definition of the policy profiles based on the well-known AppArmor syntax. In addition we logically divided the API calls by resource type and we proposed some security enhancements for access control management. Finally, we discussed about the definition of a set of tools for the automatic generation of access control profiles.

From our analysis it emerges that one of the biggest threats of ROS is in the communication mechanism between nodes and the master. As we saw in Section 7.3 the ROS APIs needs particular care in checking the permissions of the resource caller and sanitisation of the functions returned values.

Contrary to all the related works, to the best of our knowledge this is the first research that focuses on the automatic definition of embedded policy profiles

in a trusted network and actively prevents, at application level, the disclosure of sensitive information and blocks unauthorised agents by applying a-priori access control model in addition to library functions security enhancements.

Still, there are a number of issues that are part of our plans for future work as: mitigate the disclosure of sensitive information (i.e an agent profile) - as described in Chapter 8 -, hardening the slave APIs node-to-node and extend the log analysis to node level. As a further work we would like to decouple the cryptographic operations from the authenticity mechanisms by means of middleware implementations as oneM2M or other DDS. Moreover, conduct a further analysis in the definition of a real-time system with the addition of cryptographic mechanisms via 'Real-Time Publish Subscribe' (RTPS) protocol for ROS 2.

All in all, we believe that we have given a solid base for the definition of the future security mechanisms for robotic operating system that could be easily and securely integrated in big-scale deployments without suffering software limitations. Furthermore, in our opinion the definition of high level solutions as the one that has been proposed in this thesis is critical for spread security solutions by reducing the tradeoff between security and usability.

Part IV

APPENDIX

A

ROS API

Below a low-level overview of the API call divided by block:

A.1 PARAMETER

- `setParam`: set the selected parameter with the provided value
- `getParam`: retrieve parameter value from server
- `hasParam`: check if parameter is stored on server
- `deleteParam`: simply delete the selected parameter
- `getParamNames`: get list of all parameter names stored on this server
- `searchParam`: search for parameter key on parameter server. In detail , search starts in caller's namespace and proceeds upwards through parent namespaces until Parameter Server finds a matching key
- `subscribeParam`: retrieve parameter value from server and subscribe to updates to that param
- `unsubscribeParam`: unsubscribe from a parameter value on the server

A.2 SLAVE

- `getBusStats`: Retrieve transport/topic statistics
- `getBusInfo`: retrieve transport/topic connection information
- `getMasterUri`: Get the URI of the master node

- `shutdown`: Stop this server
- `getPid`: Get the PID of this server
- `getSubscriptions`: Retrieve a list of topics that this node subscribes to.
- `getPublications`: Retrieve a list of topics that this node publishes.
- `paramUpdate`: Callback from master of current publisher list for specified topic.
- `publisherUpdate`: callback from master of current publisher list for specified topic
- `requestTopic`: Publisher node API method called by a subscriber node.

  Request that source allocate a channel for communication. Subscriber provides a list of desired protocols for communication. Publisher returns the selected protocol along with any additional params required for establishing connection. For example, for a TCP/IP-based connection, the source node may return a port number of TCP/IP server.

A.3 MASTER

- `registerService`: register the caller as a provider of the specified service.
- `unregisterService`: register the caller as a provider of the specified service.
- `registerSubscriber`: subscribe the caller to the specified topic. In addition to receiving a list of current publishers, the subscriber will also receive notifications of new publishers via the publisherUpdate API.
- `unregisterSubscriber`: unregister the caller as a publisher of the topic.
- `registerPublisher`: register the caller as a publisher the topic.
- `unregisterPublisher`: unregister the caller as a publisher of the topic.

- `lookupNode`: Get the XML-RPC URI of the node with the associated name/-caller_id. This API is for looking information about publishers and subscribers. Use lookupService instead to lookup ROS-RPC URIs.
- `getPublishedTopics`: Get list of topics that can be subscribed to. This does not return topics that have no publishers. See LgetSystemState() to get more comprehensive list.
- `getTopicTypes`: Retrieve list topic names and their types.
- `getSystemState`: Retrieve list representation of system state (i.e. publishers, subscribers, and services).
- `getUri`: Get the XML-RPC URI of this server
- `lookupService`: lookup all provider of a particular service.

A.4 LEGACY CONNECTION

- `connect_topic`: Connect subscriber to topic.
- `accept_topic`: API response from the transport policy
- `connect_service`: Connect subscriber a service
- `accept_service`: transport policy API response

BIBLIOGRAPHY

[1] Mick Bauer - *"A Paranoid Penguin - An Introduction to Novell AppArmor"*, Linux Journal, 2006.

[2] Mick Bauer - *"Paranoid Penguin - Introduction to SELinux"*, Linux Journal, 2007.

[3] D. F. Ferraiolo, D. R. Kuhn - *"Role-Based Access Controls"*, 15th National Computer Security Conference, 1992

[4] National Security Agency (NSA)
`resource online:` *https://www.nsa.gov/what-we-do/research/selinux/*

[5] D. F. Ferraiolo, D. R. Kuhn, R. Chandramouli - *"Role-Based Access Controls"*,
Book, ARTECH HOUSE, 2007.

[6] Novell - AppArmor.
`resource online:` *https://www.novell.com/documentation/apparmor*

[7] Novell - AppArmor Manual.
`resource online:` *https://www.novell.com/documentation/apparmor/apparmor201_sp10_admin/data/sec_aa_whatimm_tools.html*

[8] Ubuntu - Man page.
`resource online:` *http://manpages.ubuntu.com/manpages/wily/man7/apparmor.7.html*

[9] Joon S. Park, Ravi Sandhu - *"Smart Certificates Extending X.509 for Secure Attribute Services on the Web"*
Proceedings of the of 22nd National Information Systems Security Conference (NISSC), 1999, pp. 337- 348

[10] Joon S. Park , Ravi Sandhu - *"Binding Identities and Attributes Using Digitally Signed Certificates"*
Proceeding ACSAC '00 Proceedings of the 16th Annual Computer Security Applications Conference, Page 120, 2000

[11] S. Farell, R. Housley, S. Turner - *"An Internet Attribute Certificate Profile for Authorization*,
Internet Engineering Task Force (IETF), 2010

[12] A. Lenstra , X. Wang, B. de Weger - *"Colliding X.509 Certificates"*,
Report EPFL, 2005

[13] The Transport Layer Security (TLS) Protocol Version 1.3
`resource online:` *https://tools.ietf.org/html/draft-ietf-tls-tls13-18*

[14] OneM2M - *"The interoperability enabler for the entire m2m and IoT ecosystem"*,
white paper, 2015

[15] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, A. Ng - *"ROS: an open-source Robot Operating System"*,
ICRA Workshop on Open Source Software, 2009

[16] M. Quigley, R. White, H. I. Christensen - *"SROS: Securing ROS over the wire, in the graph, and through the kernel"*
ROSCon, 2016

[17] F. J. R. Lera, J. Balsa, F. Casado, C. Fernandez, F. M. Rico, V. Matellan - *"Cybersecurity in Autonomous Systems: Evaluating the performance of hardening ROS"*
XVII Workshop en Agentes Físicos, 2016

[18] K. Lam, D. LeBlanc, B. Smith - *"Assessing Network Security"*
Book, Microsoft Press, 2004

[19] P. T. Eugster, P. A. Felber, R. Guerraoui, A. M. Kermarrec - *"The many faces of publish/subscribe"*,

Journal, ACM Computing Surveys (CSUR), Volume 35 Issue 2,Pages 114-13, 2003

[20] M. Cheminod, L. Durante, A. Valenzano - *"Review of security issues in industrial networks"*,
IEEE Transactions on Industrial Informatics, Volume: 9, Issue 1, 2013

[21] J. Akerberg, M. Gidlund, T. Lennvall, J. Neander, M. Bjorkman - *"Efficient integration of secure and safety critical industrial wireless sensor networks"*,
EURASIP Journal on Wireless Communications and Networking, no. 1, pp. 1–13, 2011.

[22] E. Byres, P. E. Dr, D. Hoffman - *"The myths and facts behind cyber security risks for industrial control systems"*,
In Proc. of VDE Kongress, 2004.

[23] D. Dzung, M. Naedele, T. von Hoff, M. Crevatin - *"Security for industrial communication systems"*,
Proceedings of the IEEE, vol. 93, no. 6, pp. 1152–1177, 2005.

[24] S. K. Datta, R. P. F. da Costa, C. Bonnet, J. Härri - *"oneM2M Architecture Based IoT Framework for Mobile Crowd Sensing in Smart Cities "*
Networks and Communications (EuCNC), 2016

[25] C. Wang, A. Carzaniga, D. Evans, A. Wolf - *"Security issues and requirements for internet-scale publish-subscribe systems"*,
System Sciences, 2002. HICSS. Proceedings of the 35th Annual Hawaii International Conference on, 2002, pp. 3940–3947.

[26] B. Dieber, S. Kacianka, S. Rass, P. Schartner - *"Application-level Security for ROS-based Applications"*,
Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on

[27] Brian Jackson - *"Analyzing HTTPS Performance Overhead"*
Article, 2017, `resource online:` *https://www.keycdn.com/blog/https-performance-overhead/*

[28] Adam Langley - *"Overclocking SSL*
Article, 2010, `resource online:` *https://www.imperialviolet.org/2010/06/25/overclocking-ssl.html*

[29] Sascha Wenninger - *"Why You Shouldn't be Afraid of SSL Performance"*
Article, 2013, `resource online:` *https://blogs.sap.com/2013/06/23/whos-afraid-of-ssl/*

[30] Compact Embedded SSL/TLS stack - MatrixSSL
`resource online:` *http://www.matrixssl.org/*

[31] M. Thompson, W. Johnston, S. Mudumbai, G. Hoo, K. Jackson, A. Essiari - *"Certificate-based Access Control for Widely Distributed Resources"*
SSYM'99 Proceedings of the 8th conference on USENIX Security Symposium - Volume 8 Pages 17-17, 1999

[32] G. Singh, Supriya - *"A Study of Encryption Algorithms (RSA, DES, 3DES and AES) for Information Security"*
International Journal of Computer Applications (0975 - 8887) Volume 67– No.19, 2013

[33] Crypto++ Library
`resource online:` *https://www.cryptopp.com/*

[34] D. Giry - "Bluecrypt – cryptographic key length recommendation",
`resource online:` *http://www.keylength.com/*, October 2016,

[35] Tully Foote - *"Community Metrics Report"* `resource online:` *http://download.ros.org/downloads/metrics/metrics-report-2016-07.pdf*, 2016

[36] D. Hood, W. Woodall - *"ROS 2 Update"*
ROSCon, 2016