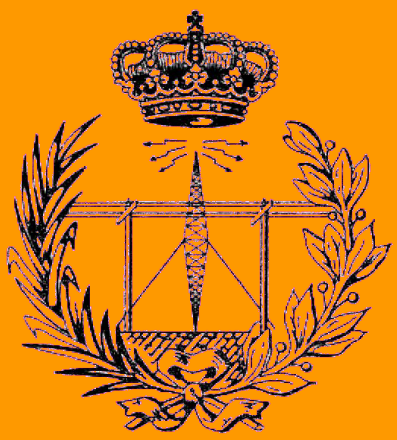
Universidad Pública de Navarra

Escuela Técnica Superior de Ingenieros Industriales y de Telecomunicación

Securing communications in Surgery Robots



Grado en Ingeniería

en Tecnologías de Telecomunicación

Trabajo Fin de Grado

Beatriz V. Fernández Muro

Paolo Ernesto Prinetto (Politecnico di Torino)
Giuseppe Airò Farulla (Politecnico di Torino)
Luis Javier Serrano Arriezu (Universidad Pública de Navarra)

Turín, Julio de 2018

Acknowledgments

Completing a thesis is one of the most importants works in the professional career, and I would not have been able to complete this without the aid and support of countless people over the past recent months.

First and foremost, I would like to thank my advisers from my destiny university Politenico di Torino, Prof. Paolo Prinetto and from my home university, Luis Javier Serrano Arriezu for the oportunity of working in a topic that I am really interested in, for their support and their always generously feedback during all stages of the development of this thesis.

I am very much indebted to my adviser Giuseppe Airò Farulla of the DAUIN Department at Politecnico di Torino. He consistently allowed this paper to be my own work, but steered me in the right direction whenever he thought I needed it, providing me advices, help and fast feedback whenever I ran into a trouble spot or had a question about my research or writing. Giuseppe also provided me contact with other students and the Cyber-Security group who I am also thankful to because of their help, advices and for calming me down in my first days and steps.

Moreover, I would like to thank my friends, family and relatives from Spain, Bulgaria, Portugal, Italy, Hungary and France, all of whom cannot be listed here, for accompanying me on this unforgettable experience. In particular I would like to mention Veni, Leonor, Rodrigo and Abel for their warm friendship, patient and for cheering me in my worst moments.

I want to express my gratitude to my best friend, Leire, who has helped me, advised me and motivate me even with almost a thousand kilometers of distance between us.

I dedicate this thesis to my beloved parents and my sister, Pedro Ángel, Mª Pilar and Ángela, whose love, company, efforts and understanding are incomparable. They are my role models of honesty, strength, and persistence.

Abstract

This thesis describes the analysis and implementation of methods for applying cyber-security to the exchange messages between tele-surgery robots. Several scenarios are developed while different security aspects to the communication between the nodes of Robot Operating System (ROS) are added. Those features concern the integration of Virtual Private Networks and devices specially tailored to cyber-security applications in a virtual scenario which represents a real one, applying the proper configuration on them in order to stablish encryption and authentication to the message exchange between the different robots.

The final goal of this analysis is to provide a solution to the cyber-security shortage in tele-surgery systems since its lack implies that someone's life could be affected.

Keywords

Cyber-security, ROS, SEcube, Tele-surgery, robots

Resumen

En el presente Trabajo Fin de Grado se estudia el análisis e implementación de métodos para aportar ciber-seguridad al intercambio de mensajes entre robots de tele-cirugía.

A lo largo del trabajo se desarrollan varios escenarios en los que se van añadiendo distintos aspectos de seguridad a la comunicación entre nodos de Sistema Operativo Robótico (ROS). Estos aspectos versan sobre la integración de Redes Privadas Virtuales (VPN) y dispositivos especialmente diseñados para aplicar ciber-seguridad en un escenario virtual representativo del real, configurádolos para establecer encriptación y autenticación en el intercambio de mensajes entre los distintos robots.

Con este análisis se pretende aportar un avance en el ámbito de la tele-cirugía ya que la seguridad tiene una alta relevancia, sin ella la vida de una persona puede verse afectada.

Palabras clave

Ciber-seguridad, ROS, SEcube, tele-cirugía, robots

Contents

List of Figures

List of Tables

1 Introduction

The topic of this thesis is related to secure tele-surgery, surgical procedures carried out remotely thanks to advances in robotic and computer technology.
This thesis focuses on ROS (Robotic Operating System) [10] architecture, security and applications given it is the most preferred software architecture of robotic applications, including surgical ones such as Raven II [11] or da Vinci Reseach Kit [12] robots.

1.1 Motivation

There is a wide range of medical robotics applications such as Robotic Assistance, Surgical robots, Pharmabotics, Companion Robots, Disinfectant Robots in healthcare, etc, [13] and applications related to them are increasing nowadays. Their use is expected to be very common in the future which signifies robots' architecture and communications have to be secure by reason of, as main goal, the patient who is being operated could not be harm by any external agent.
The analysis and research on tele-surgery - which implies ROS as well - remain in continuous improvement and, being focused on terms of security, it is well known that ROS developers did not focus on the cyber security of the system, quoting Brian Gerkey at the ROS-I conference in Stuttgart: *"If you claim that you've found a security hole in ROS, you're lying: there is no security"*. Securing ROS is important in tele-medicin applications such as surgery robots, due to the possibility of suffering an attack which could damage the robot and even harm people as the surgeon or the patient.

1.2 Objectives

The principal purpose of this thesis is to provide security to the whole communication channel and the messages exchanged between robots, particularly in the area of Tele-surgery.
The first objective is to create a simple scenario that represents a tele-surgery network, designing and developing it using a virtualized environment. The second one is to implement strategies of how to secure the environment after having a deep study and analysis

of them.

1.3 Structure

This thesis is organized as follows:

Section 2 introduces the state of art of Tele-medicin and provides a description of the history of surgery robots with a focus on possible attacks and countermeasures.

Section 3 describes the methodology that has been used as well as the scenarios developed, showing the vulnerabilities of each one and implementing security improvements until a final - secure - scenario.

Section 4 and Section 5 give the results achieved and some conclusions derived from them.

Section 6 presents the ideas of future work on this topic and possible improvements.

All the code developed for this thesis is presented and explained in the Appendices at the end of the thesis.

2 State of art

2.1 Brief history of Robotics in medicin

Since the XX century, robotics has been applied to different fields, not only to the industry field but also to research. Ideas of its use in medicin did not take long to appear, PUMA - **P**rogrammable **U**niversal **M**achine for **A**ssembly, or **P**rogrammable **U**niversal **M**anipulation **A**rm - was created by *Victor Scheinman* as an industrial robotic arm but PUMA 560, Figure 1, was used for the first time in robotic stereotactic brain biopsy in 1985. Three years later, PROBOT was used to perform transurethral prostate surgery and in 1992, ROBODOC was used to prepare (more precisely and quickly than the human surgeon) a cavity in the femur for hip replacement in human patients.

Figure 1: PUMA 560: robotic surgical arm [1].

In the decade of 1990 the da Vinci Surgical System and Zeus Robotic Surgical systems (Figure 2) were designed for minimally invasive surgery. Many surgeries were carried out by the Zeus system that time but finally the da Vinci Surgical System (Figure 3) became the most used and most known for its use in general surgery, urology and gynaecology [14].

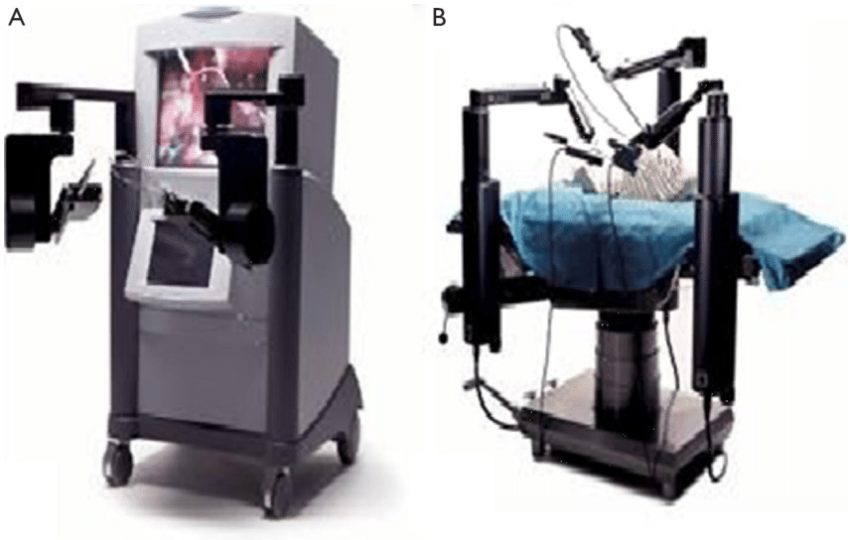


Figure 2: Zeus Robotic Surgical System.

(A) Surgeon console (B) Robotic arms. [1]

Figure 3: Da Vinci Surgical System.

From left to right: surgeon console, patient cart, vision cart. [1]

Da Vinci system is based on a "Master-Slave" paradigm. The surgeon uses a console which controls the arms using joysticks and foot pedals as Figure 4 shows. These joysticks copy the movement of the hands, scaling it and applying a filter of possible tremors. This console also contains a 3-Dimensional screen allowing the surgeon to see the surgery without needing special glasses. [2]

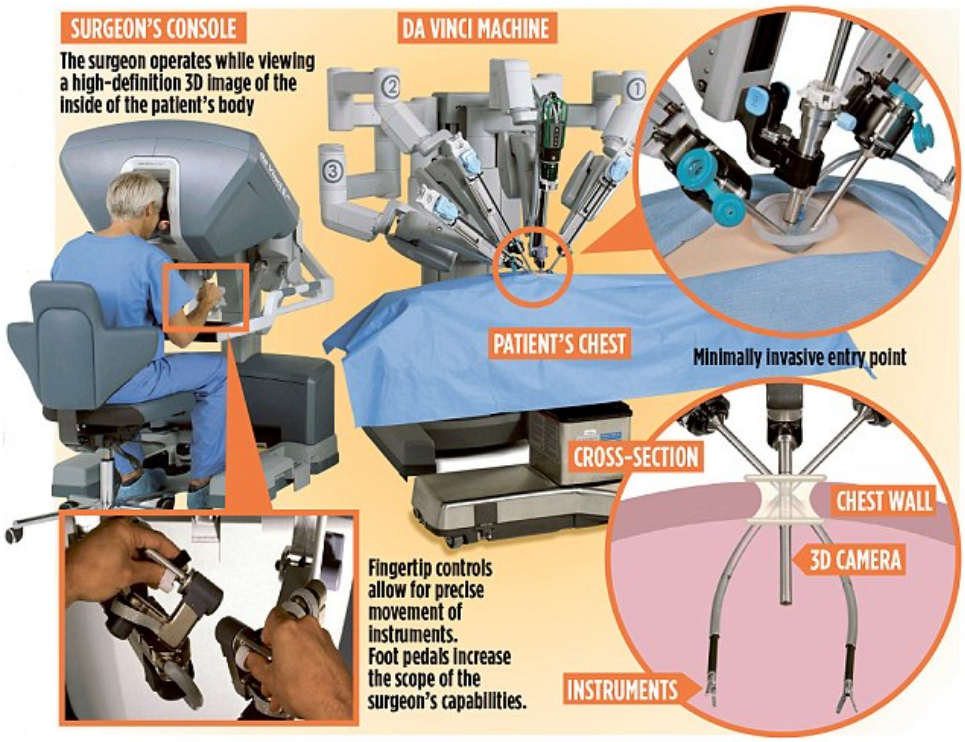


Figure 4: Parts of the Da Vinci surgery robot [2].

Some significant advantages of using the Da Vinci system are its 7 Degree of Freedom (DoF) and that a surgeon controls the surgery from a seated position separated from the patient. This method is known as telesurgery and can theoretically be performed from any distance. 7 DoF, which means 7 independent parameters that define the movement, are typically required to mimic the movement of human forelimb.

Figure 5 represents the number of operations using da Vinci surgical robot system from 2011 to 2018. It can clearly be seen that there has been a significant increase in the amount of them. Approximately 350,000 surgical procedures were performed in 2011, 753,000 and 877,000 procedures were performed in 2016 and 2017 respectively, and they are expected to increase around 11% to 15% in 2018 [3].

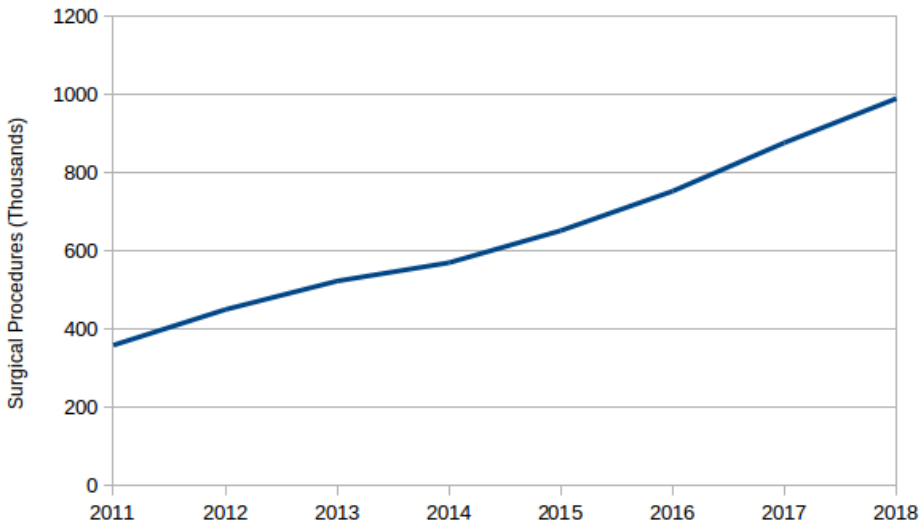


Figure 5: Number of world-wide procedures performed with da Vinci Surgical Robot System per year. Table with the exact values is available in A [3–5].

Owing to the rising trend in Figure 5, it could be claimed that the demand for operations is growing really fast, which implies that improvement on the methodology has to be developed. Some experts believe that the next step in tele-surgery progress will be the combination of surgical robotics and artificial intelligence.

There is also a problem with da Vinci Surgical System: the machine is very expensive, it is not portable and it uses a proprietary software as well. However, it has been developed an affordable and Linux-based operating system that lets users modify code, which is called Raven II. The mentioned system consists of two robot arms, a camera, and an interface for the surgeon.

In tele-surgery the lag time is really important and many researchers have studied about the possible impact of its smaller or bigger value. In [6] and [15] it is explained that surgeons can adapt to tele-surgical latencies but the maximum latency recommended is about 300ms. Figure 6 shows the effects of the delay depending of which is the value of the time-lag.

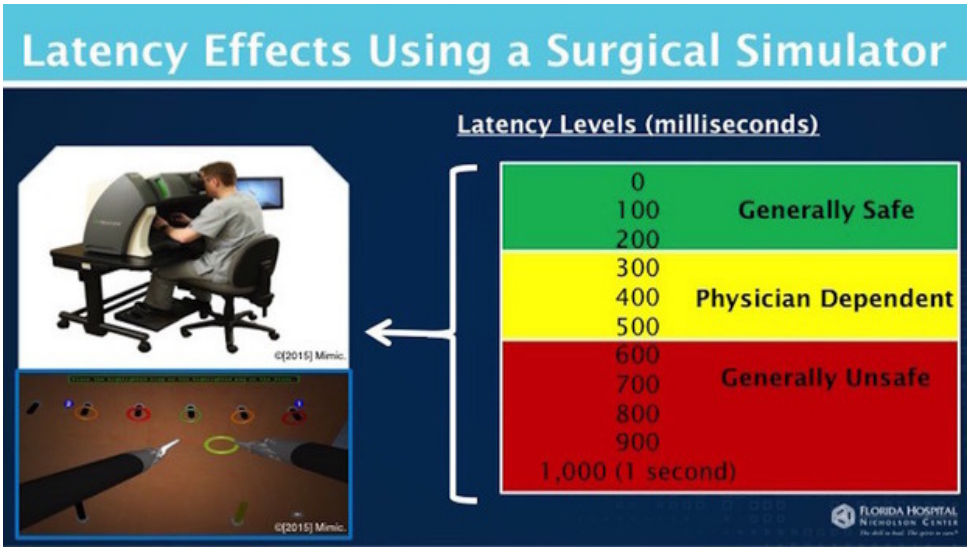


Figure 6: Effects of lag time on remote tele-surgery [6].

This values refers to possible problems caused by video lacency which could make the surgeon to be confused and to not make his/her best.

2.2 Resources

Owing to the increasing use of this method in surgeries it has become necessary to secure the systems, which means to secure ROS. To know how to secure the system, what Robot Operating System is, its vulnerabilities and some possible hardware and software to develop security have been investigated.

2.2.1 Robot Operating System

As defined in ROS Wiki [7] *"ROS is an open-source, meta-operating system for your robot. It provides the services you would expect from an operating system, including hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. (...) The primary goal of ROS is to support code reuse in robotics research and development"*.
The ROS framework is implemented in different programming languages such as C++, Python and Lisp and research is being carried out in order to implement JAVA and Lua. In terms of operating systems in which ROS could be used, it only runs on Unix-based

platforms; although a port to Microsoft Windows is possible, it has not been tested yet. The network of ROS is *Computation Graph*, a peer-to-peer network of processes that are processing data together [7]. The basic Computation Graph concepts of ROS are:

- `nodes`: written with the use of a ROS client library, such as roscpp or rospy. The communication with one another uses streaming topics, Remote Prodecure Call services, and the Parameter Server.
- `master`: provides name registration and lookup to the rest of the Computation Graph, without it, nodes would not be able to find each other, exchange messages, or invoke services.
- `parameter server`: allows data to be stored.
- `messages`: a data structure, comprising typed fields. Messages are routed via a transport system with publish/subscribe semantics: a node sends out a message by publishing it to a given topic.
- `topics`: an identifier of message's content.
- `services`: provide request-reply interactions between nodes.
- `bags`: files format for restoring ROS message data.

As a summary of what is mentioned above, Figure 7 shows a diagram of how the structure of a pair of nodes which are communication in a topic is.

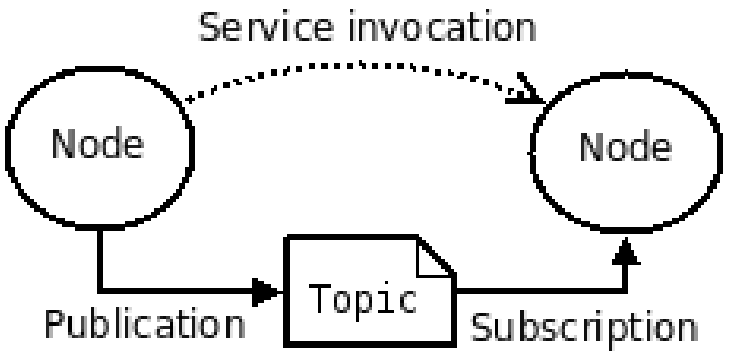


Figure 7: ROS basic features schema: relation between nodes, topics and services [7].

2.2.1.1 ROS communication

Information about the availability of topics, services and the negotiation of connection transport are implemented using XML Remote Procedure Call (XML-RPC). It is the backend of the ROS system APIS which provides the communication between the nodes [16]. There are three different API listings, Master API, Parameter Server API and Slave API that are available in [17], [18] and [19]. Some of the most relevant functions for our study are:

- `getSystemState` - Master API: shows a list representation of system state, for example publishers, subscribers, and services.
- `lookupNode` - Master API: provides the XML-RPC URI of the node with the associated name, or caller identifier, called within the function. This API is used for looking information about publishers and subscribers.
- `publisherUpdate` - Slave API: after a node is subscribed to a specific topic, this functions will also provide information about new publishers in that topic.

2.2.1.2 ROS vulnerabilities and possible attacks on tele-surgery robots

The system has known vulnerabilities due to the fact that the security in ROS was not taken into account:

1. the messages which are sent between ROS core and a node are non-encrypted
2. no authentication is required
3. there is no confidentiality

The lack of encryption, confidentiality and authentication lets an attacker carry out unauthorized Publishing (Injections), unauthorized Data Access, and even Denial of Service (DoS) attacks on specific ROS nodes [20] :

- `Unauthorized publishing:` a node is able to publish data to an arbitrary topic without prior authorization, what makes it possible to inject data or commands into an application in order to disturb its operation and even false sensor data.

- `Unauthorized data access:` every node in ROS may subscribe to every topic within the application. After that, it will receive any data that is published for this topic. This attack is especially hard to discover since a node itself may have no outgoing ROS communication.

- `Denial of Service:` there is no control of which node should publish any particular data, which makes it easy to launch an attack that publishes a large number of fake data. The attacker can also prevent the subscriber from receiving the real data.

  It is possible for an attacker to deal with unauthorized publishing and unauthorized data access attacks at the same time, for instance, a malicious node may act as a subscriber to a publisher and as a publisher to a subscriber - a Man In The Middle Attack - and transparently record and manipulate the data flow between those two nodes, represented in Figure 8 as a change of the data exchange - 0 is modified to 1. In this situation, the attacker is receiving data from the topic (receives data=0) which means unauthorized data access and, after that, he is publishing the data on a topic that he has no authorization, unauthorized publishing.

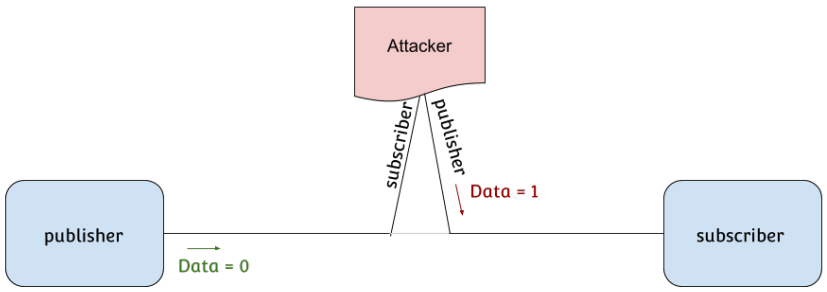


Figure 8: Schema of network when a Man In The Middle attack is being carried out.

The attacker acts as publisher or subscriber depending on what node is it attacking.

Authors in [21] classify attackers to medical robotic infrastructures into two groups, considering the attacker's role within the system:

(I) Network observer that eavesdrops on information exchange between a surgeon and a

robot. Based on the collected information he/she starts inserting false messages into the network, while still allowing both the benign parties to communicate directly.

(II) Network intermediary that assumes a role of an intermediary between a robot and a surgeon, such as MITM (Figure 8). The benign parties don't communicate directly.

Figure 9 represents a schema of the process to develop an attack and know the relevant information of the ROS network, such as topic names and addresses of the nodes. Typically, the first step is to find the URI of the ROS master - which is the IP and port of the ROS master. There are different possibilities to do so, the first one is to read it from the ROS MASTER URI environment variable - the one that client nodes use to locate the master node - but even if that is not possible, yet an attacker can scan the network using Address Resolution Protocol (ARP) in order to identify the hosts in the network. Once the hosts are known, the master can be found by trying the ROS default port on each host until it is found. If the master is configured to use a different network port - with respect to the default ROS port 11311 -, a full portscan on all nodes must be performed.
Next requirement is to get the URI of the subscriber. To do so, first a call to the procedure getSystemState within the XML Remote Procedure Call, previously introduced in section 2.2.1.1, is carried out. The master will return information about the registered publishers, subscribers and services so the node name and topic name is known.
Now, after extracting the name of the subscriber which should receive the (faked) messages, the URI of the subscriber could be got by sending a lookupNode XMLRPC message to the master with the name of the node as parameter. With this URI, the attacker is able to send the XMLRPC message publisherUpdate to the subscriber, containing the topic and a list of the new publishers to the topic - which contains the (fake) publisher created by the attacker.

While the original publisher is still active and visible in the ROS graph, its data will not reach the subscriber anymore. Note that this attack is reversible, thus, after being done, the attacker can just send another publisherUpdate containing the original publishers [20].

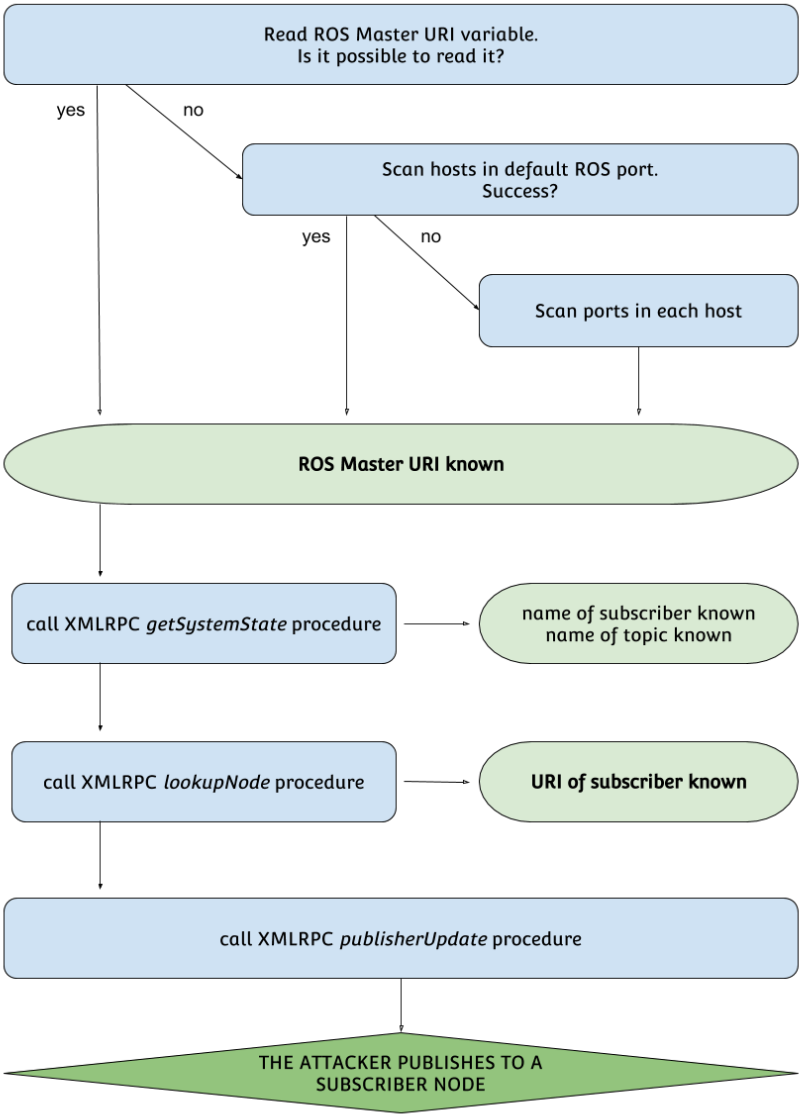


Figure 9: Schema of steps that an attacker has to follow in order to know the relevant information of the ROS network to execute an attack.

According to the impact of the attack on the surgery, it is also possible to distinguish three different categories [21]:

(A) `Intention modification`: the attacker modify the messages while packets are in-flight, the surgeon has no control over them. The effects of this attack could be for example:

- unusual robot movements

- robot becoming randomly engaged or disengaged
- unusual delays in movements

Considering the attacker's role as a network intermediary, it could be diferentiated four subgroups of attacks:

(a) Reordering → the attacker does not forward the messages to the robot in the same order as he/she receives it, the attacker sends them in a random order (and with some delay) what produces that the robot skip the messages received out of order, playing out a jerky robot motion.

(b) Packet loss → the attacker drop some packets which makes the motion become delayed and jerky.

(c) Packet delay → the attacker does not forward the messages in the exact moment he/she receives them, but sends them after some amount of time.

(d) Content modification → the attacker modifies surgeon's packets on-the-fly before forwarding them to the robot.

(B) `Intention manipulation:` the attacker only modifies feedback messages originating from a robot, such as video feed or haptic feedback. As the feedback is assumed to be valid, the attack could end up becoming harmful to a patient.

(C) `Hijacking attacks:` the robot ignores the intentions of a surgeon completely, and, instead, performs some potentially harmful actions. These attacks cause a temporary or a permanent takeover of the robot.

2.2.2 SEcube™

The exploits explained in the previous section occurs for any robotic system with telecommunication. Taking Brian Gerkey's quote in the ROS-I conference in Stuttgart *"ROS assumes a secure network! If that assumption is not true, arbitrarily bad things can happen"* into consideration, it could be assumed that the only way to secure the system is securing the network where the system is developed.

The first strategy is to provide security to the whole communication channel between all the parts of a system, providing a layer of security. By adding encryption and authentication to data streams - although the use of them will increase the use of memory, the amount will be acceptable in most cases [21] -, the attacker will have to carry out more complex attacks because it will become hampered.

Blu5 Group [1] develops added value "bricks" for others to build integrated trusted systems. An example of interest in the topic of our concern is **SE**cube™ that, in rough lines, provides:

- secure key storage,
- encryption and decryption of data stream
- authentication of data streams

SEcube™ - Secure Environment cube - Open Security platform is an open hardware and software platform for security and application developments, a system on Chip embedded environment, a unique security environment where each function can be optimised, executed, and verified on its proper hardware device [8].
Three key security elements in a single package are implemented: a fast floating-point Cortex-M4 CPU, a high-performance FPGA, and an EAL5+ certified SmartCard as represented in Figure 10.

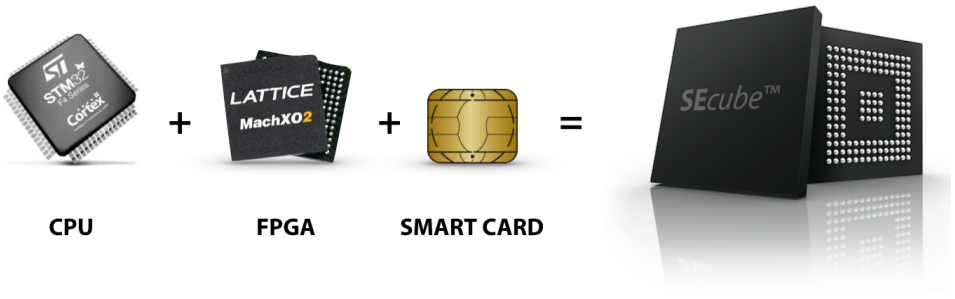


Figure 10: **SE**cube™ components.

[1] http://www.blu5group.com/

Besides the already described **SE**cube™ Chip, there are two other hardware devices related to **SE**cube: the Development Board, named **SE**cube™ DevKit (represented in Figures 11 and 12) and the USB Stick, named **USE**cube™ Stick (represented in Figures 13 and 14). The SEcube™ DevKit is an open development board designed to support developers to integrate the **SE**cube™ Chip in their hardware and software projects whereas **USE**cube™ Stick is an USB form factor based on the **SE**cube™ Chip, which deploys the **SE**cube™ functionalities through a USB 2.0 High-Speed interface [22].

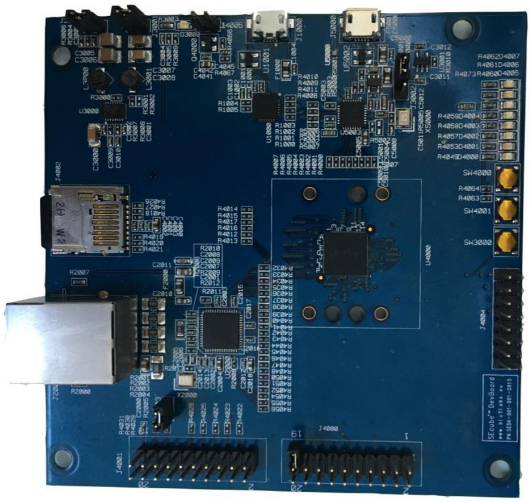


Figure 11: The **SE**cube™ Dev Kit Board.

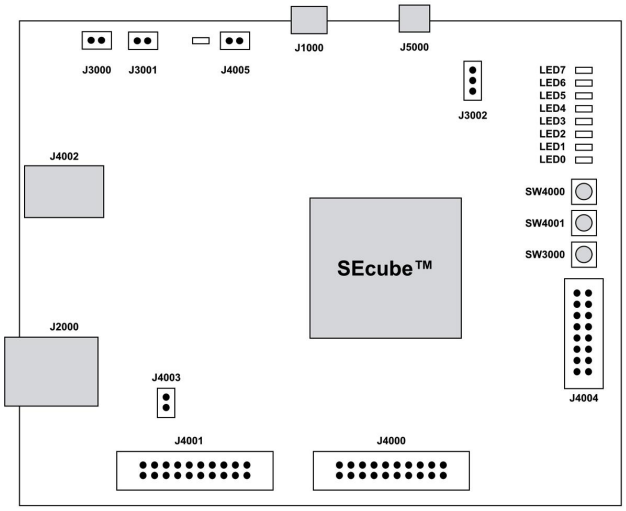


Figure 12: **SE**cube™ Dev Kit Board: interfaces and peripherals.

Figure 13: The **USE**cube™ Stick.

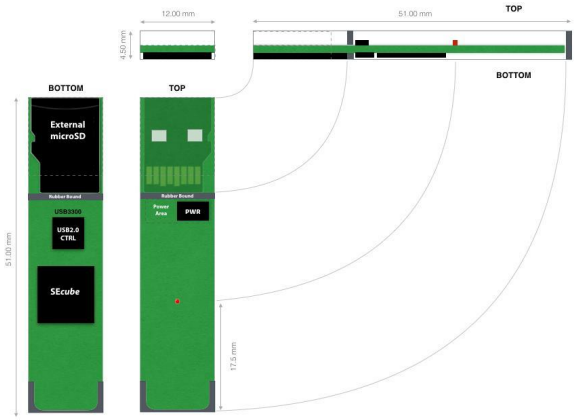


Figure 14: **USE**cube Internal structural details.

The architecture of the firmware - represented in Figure 15 - is based on Abstraction Layers (L0, L1, L2, L3 and Applications) in which each element (but the lowest one) represents a "service" for the upper level and relies on "services" provided by lower levels [8].

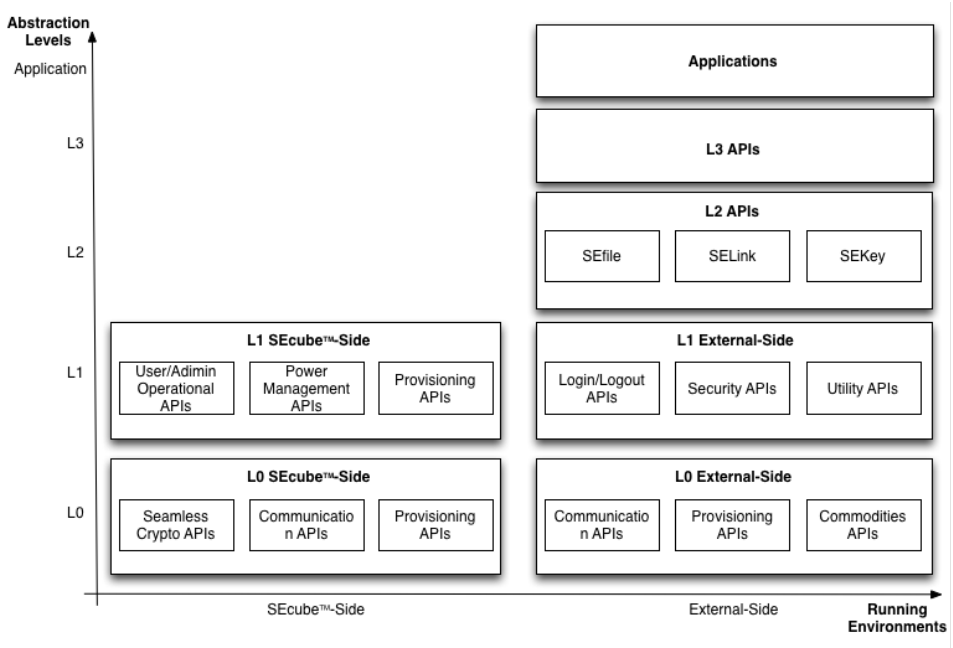


Figure 15: SEcube^{TM}: Software architecture.

What makes **SE**cube^{TM} interesting, in terms of security, is that developers are provided with an easy-to-use, high-security abstraction layer - all the digital and organisational security processes are integrated in a comprehensive, flexible and seamless way, for instance, mathematical and cryptographic elements, like keys and algorithms, are replaced by simpler concepts: groups and policies -, but security experts can verify, change, or write from scratch the whole system starting from the elementary low-level blocks.
The APIs are organised in modular libraries and abstraction layers and developers are open to create their solution starting from the most suitable entry point according to their expertise. The L2 level APIs offer optimized and easy-to-use functionalities to ease the development of applications fully protected (i.e., authenticated and confidential), without being forced to understand in details all the low-level hardware and security mechanisms. The provided APIs include:

- **SE**Key^{TM} : a Key Management System. Whenever a group - pool of one or many users - is created, **SE**Key^{TM} manages the creation of the group communication key (which is used to generate session communication keys and a set of security policies), the security mechanisms and the relative transmission to the entitled users.

- **SE**File^{TM}: a data-at-rest protection facility. It provides encryption and signature

facilities, based on two master keys (EncDBKey and SigDBKey). Every secure file is encrypted and signed, sector by sector, using dedicated keys derived by the two mSE master keys, and - as represented in Figure 16 - the file name is coded in a way that nobody can recognize it looking directly at the physical file system.

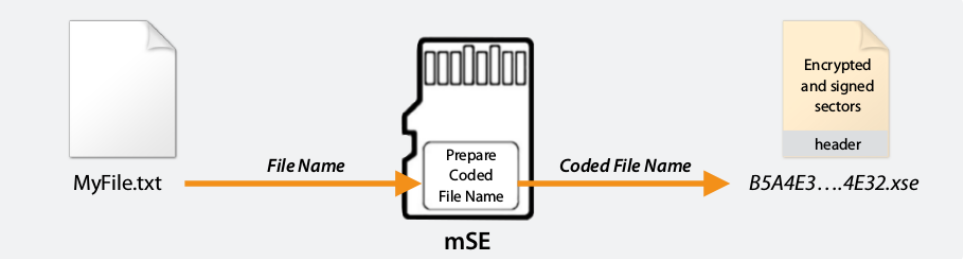


Figure 16: SEfile: process of coding the name of the file.

- **SE**Link™: a data-at-motion protection facility. It secure the network traffic by encrypting network streams originating from any application, regardless the application-level protocol. Encrypts connections silently with no modification to the application. Figure 17 shows the architecture, the Client-side components are **SE**Link driver, **SE**Link service and **SE**Link GUI, and the server-side ones are **SE**Link gateway and **SE**Link gateway web UI. Figure 18 represents how a driver redirects the connection requests to **SE**Link service and, finally, Figure 19 shows the final connections where the Service and Gateway components take care of bridging non-secure connection to the Secure Link.

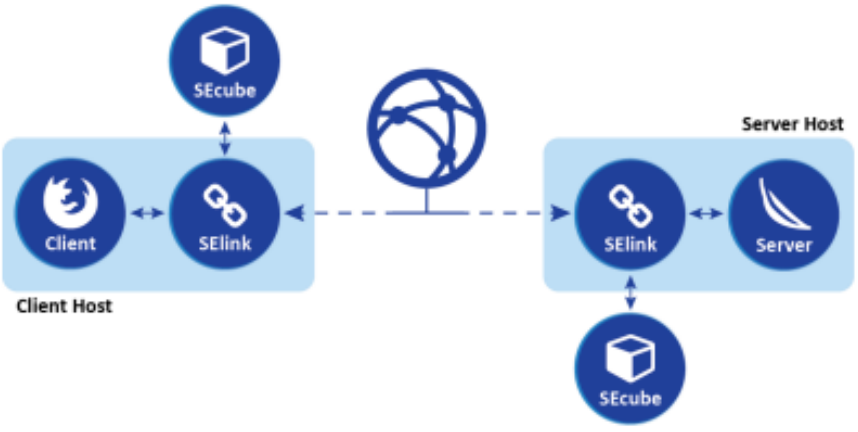


Figure 17: SE Link: Client and Server side components [8].

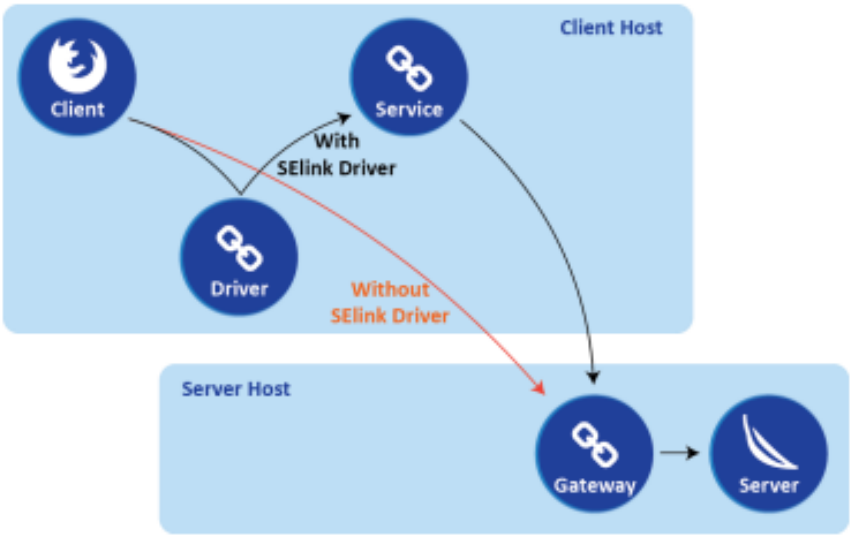


Figure 18: SE Link: Connection establishment process.

Each directed arrow represents a TCP connection request [8].

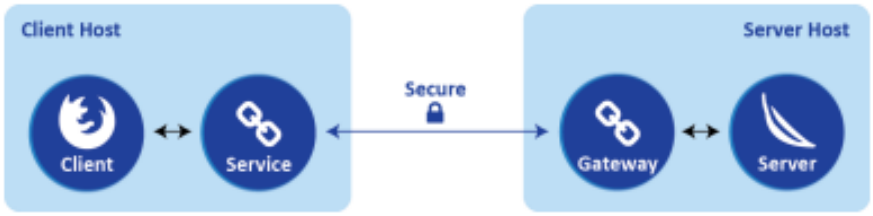


Figure 19: SE Link: final connections.

Each directed arrow represents a TCP connection [8].

2.2.3 Tunnelling - Virtual Private Network (VPN)

Besides **SE**cube^{TM}, another technology used to carry out this purpose is Virtual Private Network (or VPN), which is a technology built using public wires - usually the Internet - to privately connect remote users.

The advantage of using VPN is that connections between remote networks could be established no matter of the distance and that, as added value, it guarantees the security level when the underlying infrastructure cannot provide it on its own. It secures the private network, using encryption and other security mechanisms to ensure that only authorized users can access the network and that the data cannot be intercepted. This type of network is designed to provide a secure, encrypted tunnel in which transmit

the data between one user and another - remotely connected. This means that once a connection through a VPN is established, all the traffic becomes encrypted and - as shown in Figure 20 - the client's IP address (IP X) gets replaced (in eyes of external agents) with the address (IP Y) of the VPN server - the third party that connects to the final destination on your behalf.
In rough lines, using a VPN provides:

- Encryption of the online data
- Activity hiding from any external agent
- Location hiding
- Anonymity

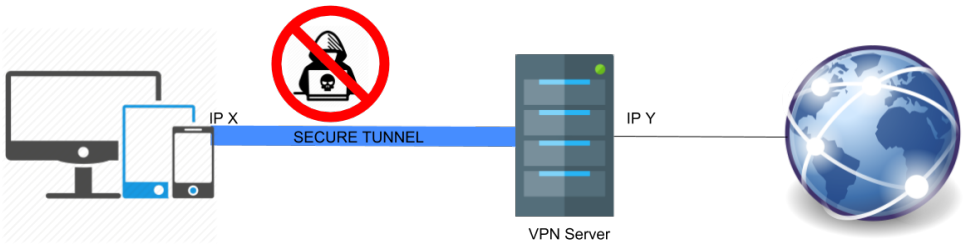


Figure 20: VPN definition summary.

Hosts on Internet will see the IP address Y as the IP address of the devices and an external agent could not be able to manage the online data.

There are different implementations of VPN such as IPSec, OpenSSH, OpenVPN, Socat, etc, and many differents protocols to use within them.

2.2.3.1 VPN Protocols

A VPN protocol defines how the service handles data transmission over a VPN. The most common protocols are all to be cited:

- Point-To-Point Tunneling Protocol (PPTP) was designed by Microsoft as part of the Windows Operation System. It encapsulate the existing protocol PPP (Point-

to-Point Protocol). Due to its easy configuration it is one of the most used protocols, but its autentication protocols, such as MS-CHAP-v1/v2, make it vulnerable.

- Layer 2 Tunneling Protocol (L2TP) is a combination of PPTP and Cisco's L2F protocol and it is also associated with the Internet Protocol security (IPsec) protocol in order to apply a strong encryption and authentication. It establishes a secure connection using keys for authenticating and encrypting each IP packet of the communication, nonetheless there may be ways for an attacker to decrypt the VPN session.

- Secure Socket Tunneling Protocol (SSTP) transports the traffic through Secure Sockets Layer (SSL) protocol and use SSL/TLS encryption.

- Internet Key Exchange Version 2 (IKEV2) establishes a security association over IPsec. IKE uses a Diffie-Hellman secret exchange of keys in order to establish the shared secret of the session.

- OpenVPN is a full-featured open source SSL VPN solution provided in the Ubuntu Repositories and licensed under the GNU General Public License (GPL). It implements OSI layer 2 or 3 secure network extension using the SSL/TLS protocol.

Table 1 represents a comparison between all this protocols. It is focused on their security and uses in order to know which one meets our requirements and specifications: to get the objective of this thesis the whole session has to be secured as much as possible so OpenVPN is used owing to its principal advantages such as portability, compatibility with NAT and dynamic addresses, ease of configuration and fast speeds, and because it is one of the most secure protocols, the most reliable and the most stable.
It accommodates a wide range of configurations, including remote access, site-to-site VPNs, Wi-Fi security, and enterprise-scale remote access solutions with load balancing, failover, and fine-grained access-controls [23].

OpenVPN is the best option for our purpose because it is open-source, its installation is easy and fast, it supports NATs and has cross-platform portability across most of the

known computing devices and, in addition, it has encryption combined with authentication which could be useful for our security purpose.
Still it is not impenetrable by an expert attacker so we need to add extra security - by implementing **SE**cube - that does not conflict with the one offered by OpenVPN: by leveraging on both, we ensure compliance to best standards in security.

	PPTP	L2TP/IPsec	SSTP	IKEV2	OpenVPN
Encryption Strength	128 bits using MPPE protocol	256 bits using AES cipher	256 bits	256 bits using AES cipher	256 bits using AES cipher
Security	Basic encryption	Highest encryption: checks data integrity and encapsulates the data twice	Highest encryption: data gets verified before being sent and received, SSL encryption included	Fastest and most secure VPN protocol	Highest encryption: Authenticates data with digital certificates
Speed	Fast due to lower level of encryption	Relatively slow, requires more CPU processing	Slow speed due to superior level of privacy and security	Fast speeds	Fast speeds
Compatibility	· Windows · MAC · iOS	· Windows · MAC · iOS	· Windows · MAC	· Windows · MAC · iOS	· Windows · MAC · iOS · Android · Linux · Router · Raspberry Pi · Qnap · Synology Nas

Table 1: Comparation between VPN protocols [9].

3 Architecture and methodology

To create the proposal for this thesis, all the exposed and described specifications in State of Art section has been taken into account. By doing this, the implementations thought for securing a tele-surgery scenario are ensuring to be accurately conveyed.
The methodology adopted is based on creating a basic scenario with no security, and add security improvements - OpenVPN and **SE**cube^{TM} - to it until a secure scenario is developed.

3.1 Material and methods

The resulting architecture in Figure 21 shows the secure communication between a node and ROS core. A **SE**cube^{TM} is going to be used for each robot of the network and another for the Master PC (in which ROS core is going to be run) as well.
In the particular case of Surgery Robots, as Figure 22 shows, it will be needed a first **SE**cube^{TM} device - previously introduced in section 2.2.2- for the robot where the surgery is taken place, a second one for the robot which the surgeon will use and a third one for the Master PC.
However, in this thesis, it is going to be used only two **SE**cube^{TM} Development Kits - one in Surgeon node and another in Patient node - instead of three as explained in the previous paragraph. This change in the scenario was applied due to a faster development and because it is not compulsory, nor necessary, that the Master node use it: the only purpose of the mentioned node is to resend the data from the Surgeon to the Patient and for that, it does not need to understand the exchanged data.

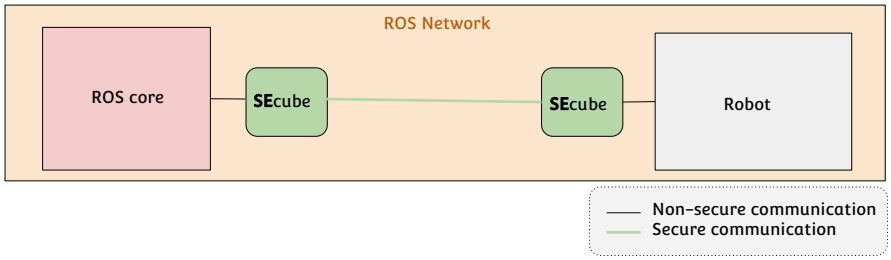


Figure 21: Secure Architecture of a ROS Network using **SE**cube^{TM}.

The network has been thought to enable a tele-surgery, that is to say that is remotely-connected: the surgeon could be in a hospital from, for instance, Germany, and the surgery could take place in, for example, Italy. So, we use VPN to provide connectivity between remote networks no matter how far they are or if the Network Address Translation (NAT) [24] configuration is set - which do not let the connection to be established.

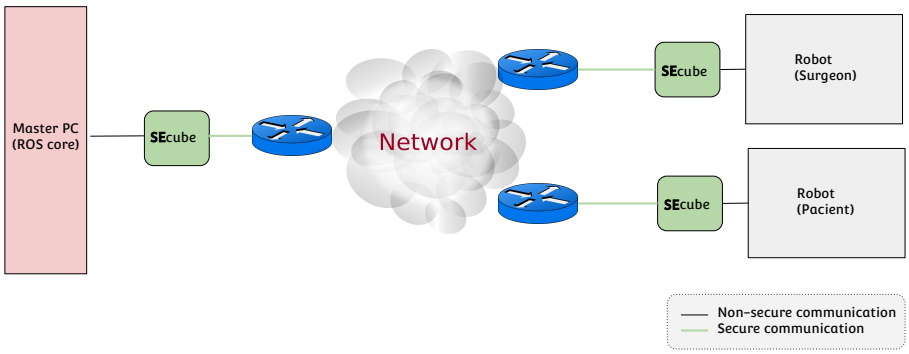


Figure 22: Secure Architecture of a Surgery Robot Network using **SE**cube^{TM}

When the surgeon robot has to communicate with the patient robot, the path which the messages has to go through is:

$$\text{Robot (surgeon)} \longleftrightarrow \text{Master PC} \longleftrightarrow \text{Robot (Patient)}$$

It will never be possible to directly communicate between the surgeon robot and the patient robot: in terms of programming in ROS - as explained in section 2.2.1 -, there is a listener in the Master PC subscribed to a topic in which a publisher in the surgeon publishes; moreover, there is another listener in the patient robot which is subscribed to a second topic, in which the Master PC's publisher publishes. A clarifying schema of the structure could be seen in Figure 23, in which the concepts related to a first topic are printed in a green color and the ones related to a second one are in blue.

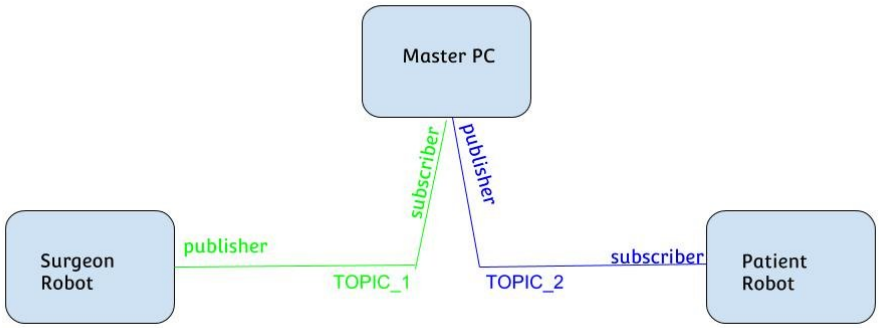


Figure 23: ROS: schema of publishers, subscribers and topics in a Surgery Robot network.

3.2 Scenarios

To study the final scenario, in which a surgery robot will be implemented developing VPNs and using **SE**cube^{TM}, we have worked on different virtualized scenarios.

(I) `Basic scenario:`

Three nodes in different virtual machines are implemented - as represented in Figure 24 -, representing Master PC, surgeon's robot and patient's robot each, in which a communication protocol is used. This protocol is based in 4 types of messages that include different information:

0 = Emergency Stop

1 = Initialization

2 = Pedal UP

3 = Pedal DOWN

Figure 24: Basic scenario: schema of virtual machines.

In the case of Pedal DOWN, besides the code 3 it will be also sent the movement that the patient's robot has to follow [25]. The movement could have different directions so, for simplicity and clarity, three axes are represented: X, Y and Z. Each

of it is shown in Figure 25: X represents right (positive) and left (negative), Y, up (positive) and down (negative) and Z, the depth: when the movement goes to the front it is negative and when it gets closer to the surgeon, positive.
As an example of what has just been explained, in Table 2 it is shown the encode data of some given instructions.

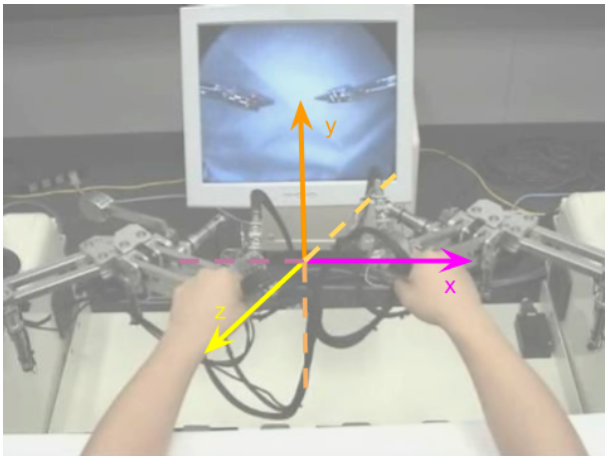


Figure 25: Basic scenario: axes of movement when surgeon moves the robot.

Instruction	Encode result
Emergency Stop	0
Init	1
Pedal Up	2
Pedal Down and the movement that the surgeon does is 1 left, 2 up and 0.3 to the front	3 ->-1 2 -0.3
Pedal Down and the movement that the surgeon does is 0.2 right, 0.1 down and 0.1 to him	3 - >0.2 -0.1 0.1

Table 2: Example of encode commands.

A in-deep explanation and the code of this scenario could be found in Appendix B.1.

To create the scenarios it has been used 3 virtual machines where it has been implemented one node each - with its required ROS files - as Figure 26 shows. The

figure mentioned also represents the network of the scenario: Surgeon and Master machines are in one network and Master and Patient ones, in another, what means that surgeon has one interface in Internal Network mode, master has two in the same mode but different networks each and finally, patient, as surgeon machine, has only one.

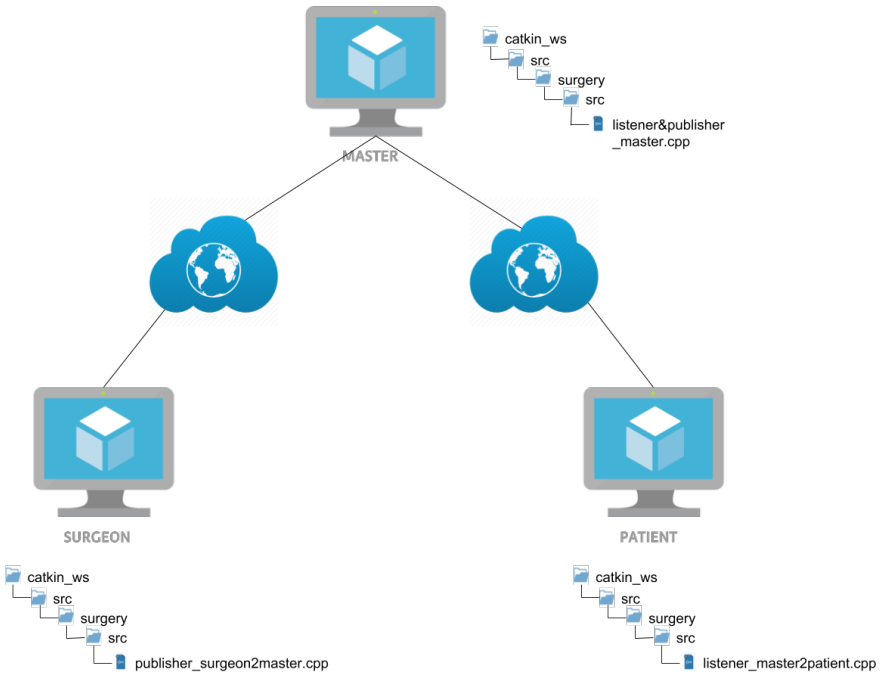


Figure 26: Scenario 1: networks and schema of folders and files in each virtual machine.

Besides, owing to ROS configuration in multiple machines, a configuration of ROS parameters and default routes to concrete IP addresses have to be set. This configuration and its steps is available in Appendix B.1.4.

(II) `Adding the VPN to basic scenario:`

Same structure as the first one but adding tunnels, using OpenVPN - introduced before in Chapter 2.2.3.1 -, in order to provide the connection though Internet between every remote network.

In this scenario, each virtual machine is connected to Internet through a bridged interface as shown in Figure 27. Moreover, the surgeon node and patient node are VPN clients and the master node is the VPN server. This architecture and the

tunnels that are going to be created are represented in Figure 28.

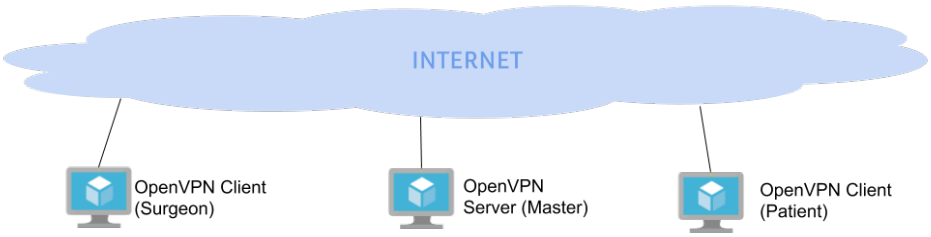


Figure 27: VPN: schema of virtual machines.

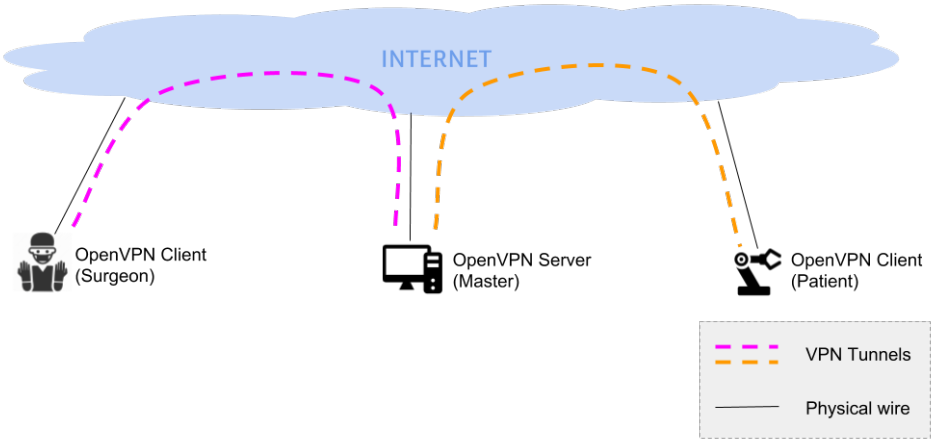


Figure 28: VPN: network.

A in-deep explanation of the architecture and configuration of this scenario could be found in Appendix B.2.

The verification of OpenVPN working properly for our objective has been done analyzing the exchanged messages. For this purpose Wireshark tool is utilized capturing a message exchange between master and surgeon. This results are available in section 4.2.

(III) `Adding SEcube™ to the basic scenario within VPN (complete scenario):` Same structure as the second mentioned, but **SE**cube™ is added in each node to make the communication secure - encrypted and authenticated.

Given the fact that what is needed in the field of tele-surgery within ROS is to secure the exchanging information via ROS topic, there are many possibilities, it is possible to implement the **SE**File^{TM} libraries and **SE**Link^{TM} ones as well to secure the whole software environment where ROS runs.
Besides, installing the ROS core on top of a protected **F**ile **S**ystem on **U**ser**S**pace (FUSE) allows building a system that intrinsically prevents external agents (attackers) from tampering with any of the functionalities exposed by ROS. In addition, the attacks mentioned in section 2.2.1.2 will be hindered, neither an attacker could modify the core libraries used by ROS - the ROS environment -, nor could change any of the configuration files used by a robot to store its inner status and parameters.

In the current work, **SE**file is going to be used: although this process is a data-at-rest protection - it creates a secure, signed and encrypted file with an encrypted name from an unsecured file which contains plain text -, it could be use in ROS systems by creating and reading files in each node:

1.- the content of the ROS message is created as a file in the publisher node.

2.- the data of the file is encrypted and wrote as the file content.

3.- the new encrypted data is read and sent through the communication channel.

Consequently, the opposite process takes place in the subscriber node:

4.- the received data is saved as a file.

5.- the resulted file data is decrypted.

6.- That plain text generated data is read and execute by the patient node.

The schema of the whole process is represented in Figure 29 with the intention of clarifying the concept.

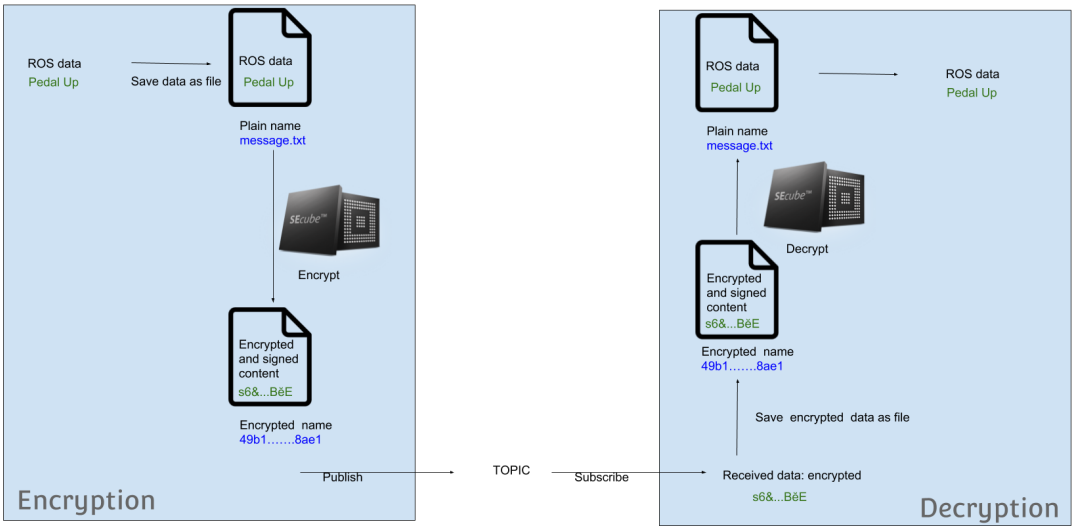
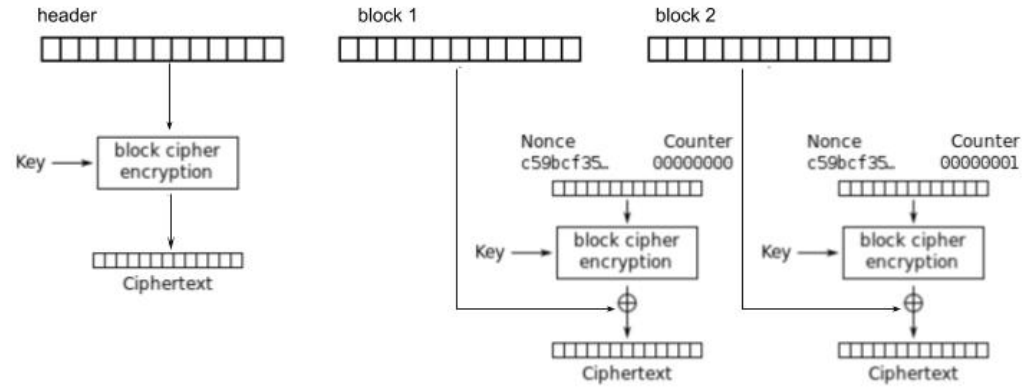


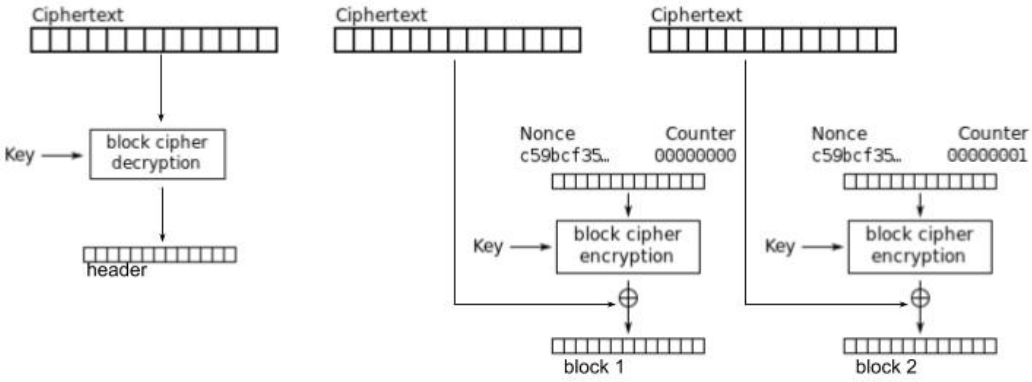
Figure 29: Encryption and decryption process using **SE**file library.

The Encryption Algorithm used is The Advanced Encryption Standard (AES) [26] which is a symmetric cryptography algorithm that ciphers blocks of data. Nowadays, some theoretical attacks have been shown, but still it has not been detected a successful attack to this cipher what makes of it a proper option to use. In **SE**file, the data-to-encrypt is divided into blocks, and each block is encrypted separately: each sector is encrypted using AES-256-CTR with exception of the header that is encrypted using AES-256-ECB. This concept is represented in Figure 30.

The first mode, CTR - counter - generates the next keystream block by encrypting successive values of a value called "counter" which is a sequence that is not repeated for a long time. That way, each block cipher depends on an ascending counter which start from a randomly selected initialization vector. Today, CTR mode is widely accepted and any problems are considered a weakness of the underlying block cipher. The Electronic Codebook mode, instead, is used in the header to provide independence from any initialization vector.



(a) Encryption algorithm process of two blocks and the header.



(b) Decryption algorithm process of two blocks and the header.

Figure 30: Encryption and Decryption algorithm processes. It is represented each mode of encryption/decryption (CTR or ECB) depending on the block.

Data confidentiality is guaranteed with the algorithms mentioned above, but authentication is also required. To sign each sector, including the header, Secure Hash Algorithm (SHA) - in particular SHA-256 - and keyed-hash message authentication code (HMAC) are used what means that within SEfile the signature depends on both the data contained in the sector itself and on a chosen encryption key. For having a deep explanation of how this algorithms works within **SE**file see the pdf file available in [22].

These processes of authentication and encryption are carried out creating a secure environment setting variables that are going to be used within the functions such as a session identifier, key identifier and cipher algorithm and mode.

The code and explanation of this scenario could be found in Appendix B.3.

Finally, Figure 31 and 32 summarize the in-variables needed in Publisher and Patient node functions, respectively.

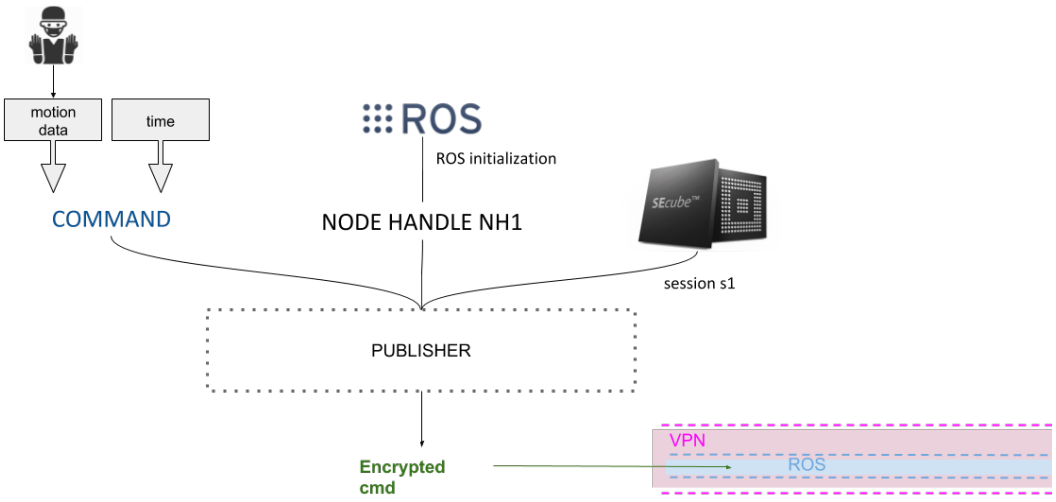


Figure 31: Schema of variables and environment needed in Surgeon node to encrypt the data sent via ROS within VPN.

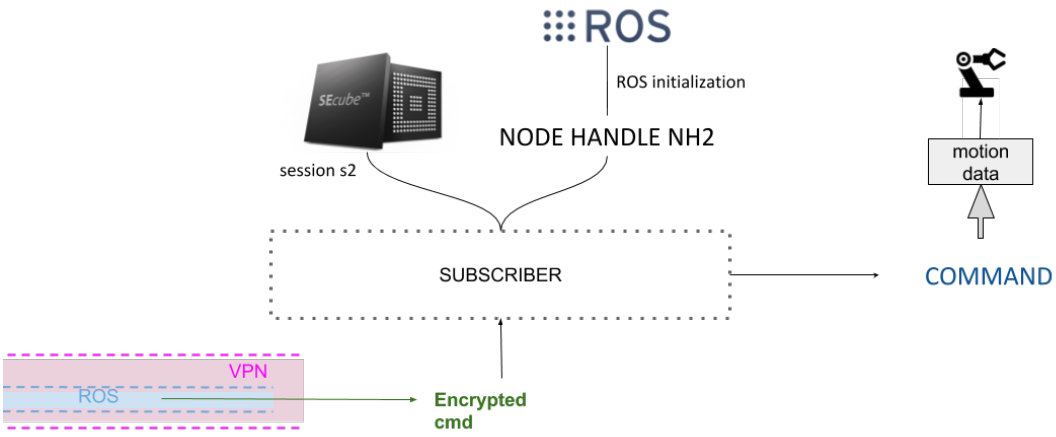


Figure 32: Schema of variables and environment needed in Patient node to decrypt the data received via ROS within VPN.

Since the main goal of this thesis is applying the secure pipeline to tele-surgery, an application where even a small time-lag could cause huge problems, it is mandatory to analyse the performances of the proposed solution.

For this goal, we need to know how much time does the encryption and decryption processes take. To do so, the Boost library of C++ is going to be used and four different measures will be analysed: publisher and subscriber time-lags using **SE**cube and publisher and subscriber time-lags not using **SE**cube. The values obtained with this process correspond to the difference of the time when the data is ready to be sent and the time when the motion data from the instructions file is read. A schema of the included processes when calculating the time-lag of encryption or decryption is shown in Equations 1, 2, 3 and 4. With this measure, the time-lag that distance or network use could cause is not interfering in the analysis.

$$time\_lag = time\_1 - time\_0$$
$$time0 = read\_instructions$$
$$time1 = read\_instructions + encode\_data + write\_encoded\_file + encrypt + read\_encrypted\_file$$

Equation 1: Schema of which functions does the calculation of time-lag in publisher include when **SE**file is being used.

$$time\_lag = time\_1 - time\_0$$
$$time0 = read\_instructions$$
$$time1 = read\_instructions + encode\_data$$

Equation 2: Schema of which functions does the calculation of time-lag in publisher include when **SE**file is not being used.

$$time\_lag = time\_1 - time\_0$$
$$time0 = read\_received\_data$$
$$time1 = read\_received\_data + decrypt + read\_decrypted\_file + decode\_data$$

Equation 3: Schema of which functions does the calculation of time-lag in subscriber include when **SE**file is being used.

$$time\_lag = time\_1 - time\_0$$
$$time0 = read\_received\_data$$
$$time1 = read\_received\_data + decode\_data$$

Equation 4: Schema of which functions does the calculation of time-lag in subscriber include when **SE**file is not being used.

4 Results and discussion

This section deals with the results of applying the explained methods in Section 3, and discuss them taking into account the following aspects:

(1) If it is possible to encrypt and decrypt the exchange data

(2) If that process entails a communication time-lag between nodes which is appropriate to tele-surgery.

To do so, the scenarios have been developed and the exchanged messages between the parts have been observed, analysed and saved using Wireshark tool [2] - a network packet analyzer that captures live packet data from one ore more network interfaces and displays the packets with very detailed protocol information [29] - and ROS command-line tools such as:

`rostopic:` it is a command-line tool for printing information about ROS Topics.

`rostopic info` → prints information about an active topic.

`rostopic echo` → prints messages that are being exchanged in that moment to screen.

`rosnode:` it is a command-line tool for printing information about ROS Nodes.

`rosnode list` → lists active nodes.

`rosnode info` → prints information about a node.

The following results are explained divided in different sections depending of which scenario is it involved.

4.1 Basic scenario

In the case of this scenario, as expected, the messages are sent in plain text and without any kind of security. An example of some data exchanged is shown in Figure 33 where it can be seen that the data is exactly the same as the one after being encode.

[2] `https://www.wireshark.org/`

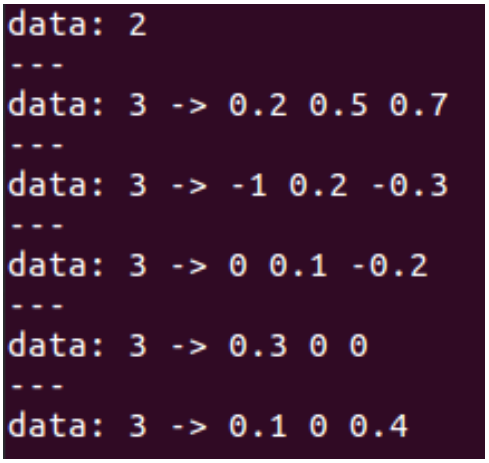


Figure 33: Plain data of a message sent via ROS.

To obtain this information the command "rostopic echo *nameOfTopic*" has been executed.

Due to the lack of security in ROS, in this scenario an attacker is able to eavesdrop the communication and get some information about it. As an example of that, Figure 34 and Figure 35 - two different Wireshark captures of packets between Surgeon node and Master node - show that an attacker could be able to know (among others):

1.- IP and MAC addresses of the nodes

2.- ROS topic

3.- nodes names

4.- information contained in a message

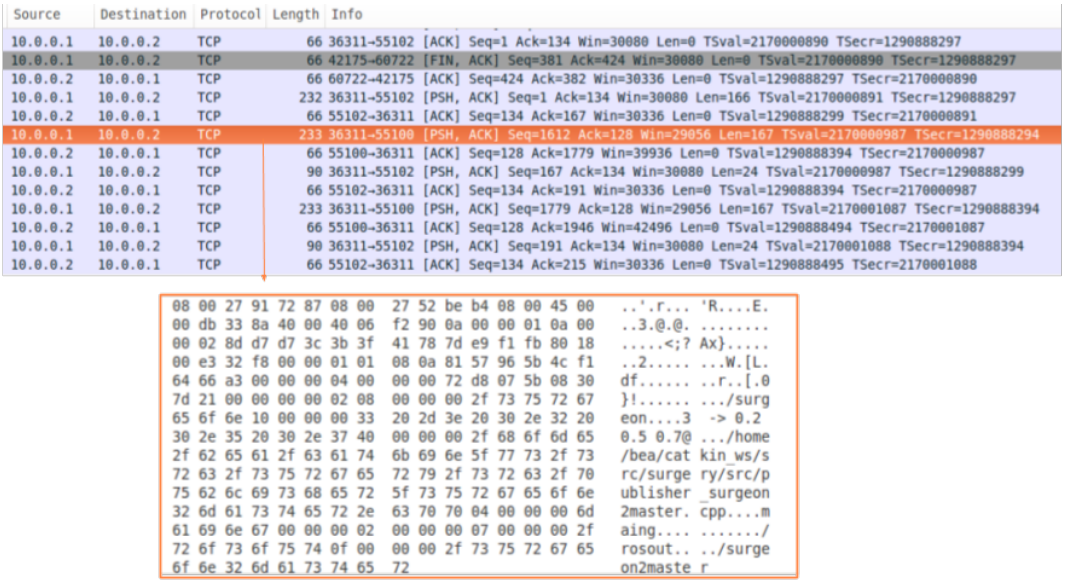


Figure 34: Basic scenario: Wireshark capture of a plain message when surgeon prints its data.

Capture taken from a Virtual Machine in the same network the nodes VMs are. The message shown is a message that the surgeon sends to the master in which ROS topic is surgeon2master, the surgeon IP is 10.0.0.1 and its name, surgeon, the master IP is 10.0.0.2, and its name, master. The message corresponds to one sent when surgeon execute the function ROS_INFO(), which is used to print data in the shell of a node.

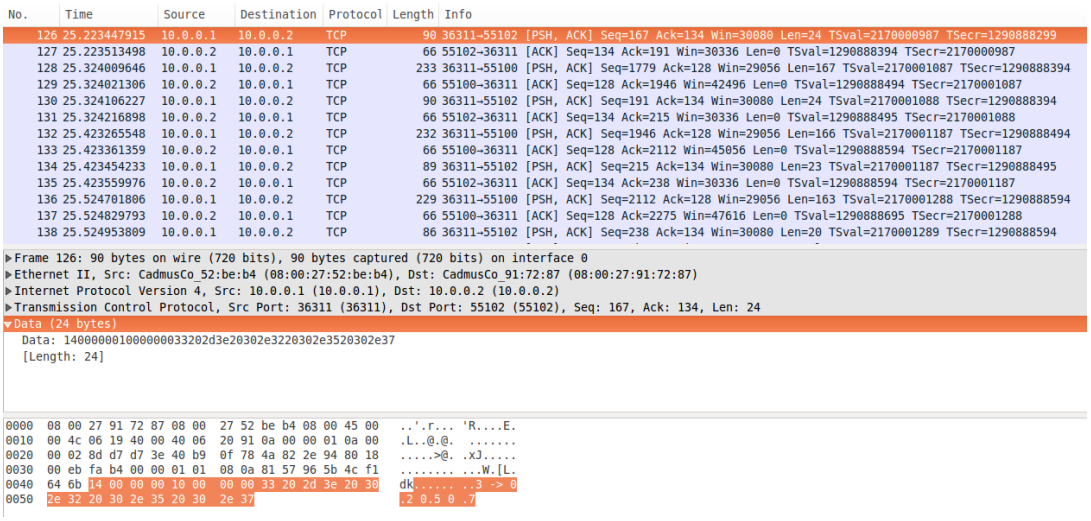


Figure 35: Basic scenario: Wireshark capture of a plain message when surgeon send the motion data.

Capture taken from the master node Virtual Machine. The message shown is a message that the surgeon sends to the master in which the surgeon IP is 10.0.0.1 the master IP is 10.0.0.2 and the data that is being sent is 3 -> 0.2 0.5 0.7.

4.2 Adding VPN to Basic scenario

A Wireshark capture of messages sent via VPN is shown in Figure 36. It seems that the messages are still in plain text, but that is only caused because Wireshark is running in

master and the VPN server has already decrypted the data.

Nonetheless, in Figure 37 - which is another messages capture has been taken in the PC that hosts the virtual machines- it is appreciated how data is encrypted and an external agent is not able to know any information contained within a message.

Another difference between both figures, are the IP addresses: whereas in Figure 36 the IPs correspond to the tun interfaces: surgeon's is 10.8.0.6 and master's, 10.8.0.1, in Figure 37 those are the public IPs: surgeon's is 192.168.1.146 and master's, 192.168.1.208.

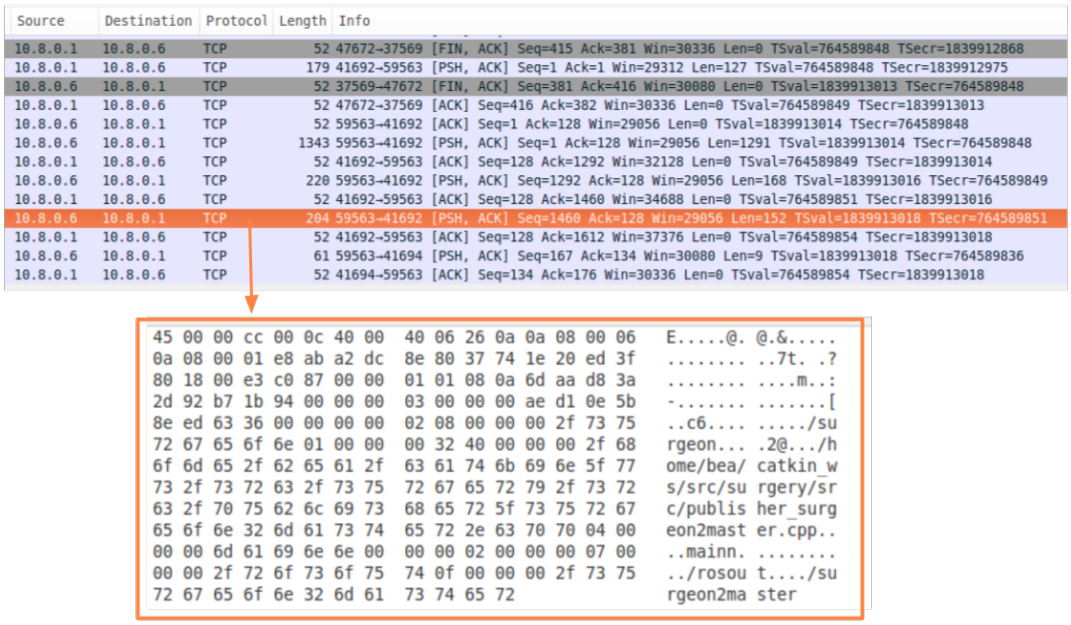


Figure 36: Adding VPN scenario: Wireshark capture of messages in master Virtual Machine.

Capture taken from master Virtual Machine. The message shown is sent from surgeon to master including surgeon2master ROS topic.

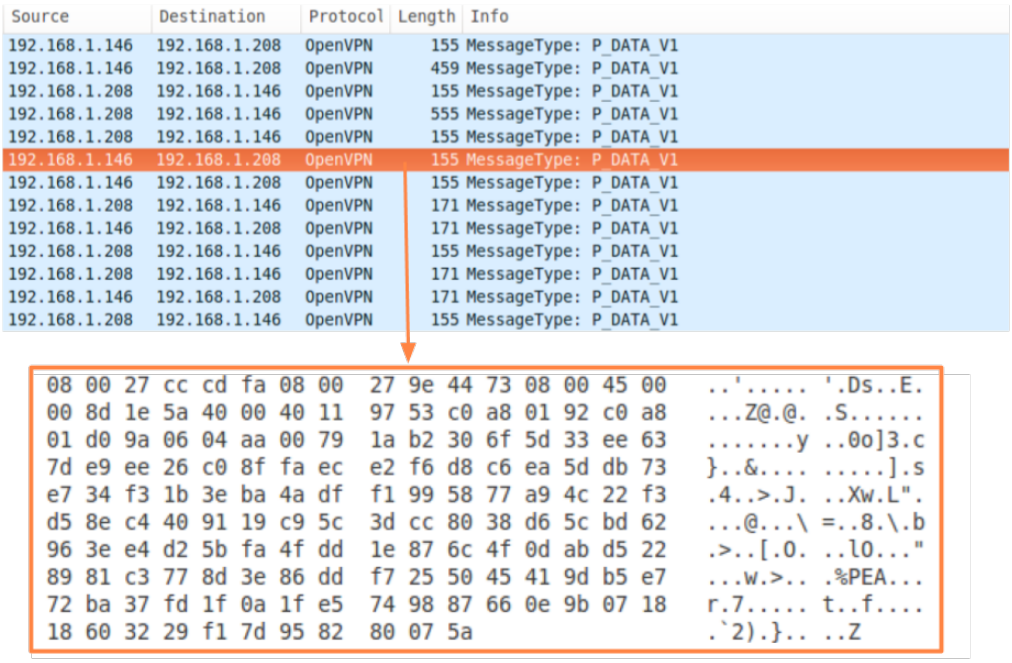


Figure 37: Adding VPN scenario: Wireshark capture of messages in external PC.

Capture taken from the PC which host the virtual machines. The message shown is sent from surgeon to master through public IP addresses. The data is encrypted so no information could be known or deduced.

4.3 Adding SEcube to Basic scenario within VPN

In the final scenario the data sent via ROS is encrypted and authentication is required. The first fact that let us know that effects is the content of the file created by the encryption function. Figure 38 shows part of the binary data contained in the mentioned file. The second one is that, when executing "rostopic echo *nameOfTopic*" - whose result is available in Figure 39 - the data cannot be clearly understood because it is an amount of bytes, whereas in the basic scenario (Figure 34, Section 4.1) the data could be easily read and interpreted.

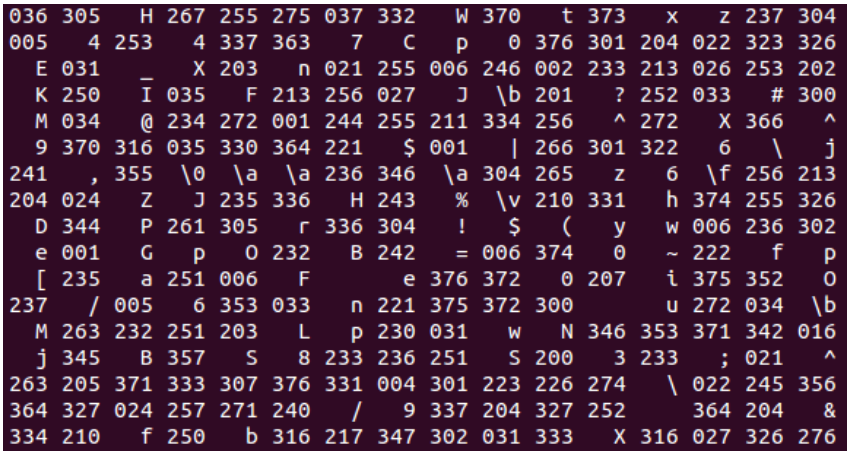


Figure 38: Encrypted data of file created using **SE**file library.

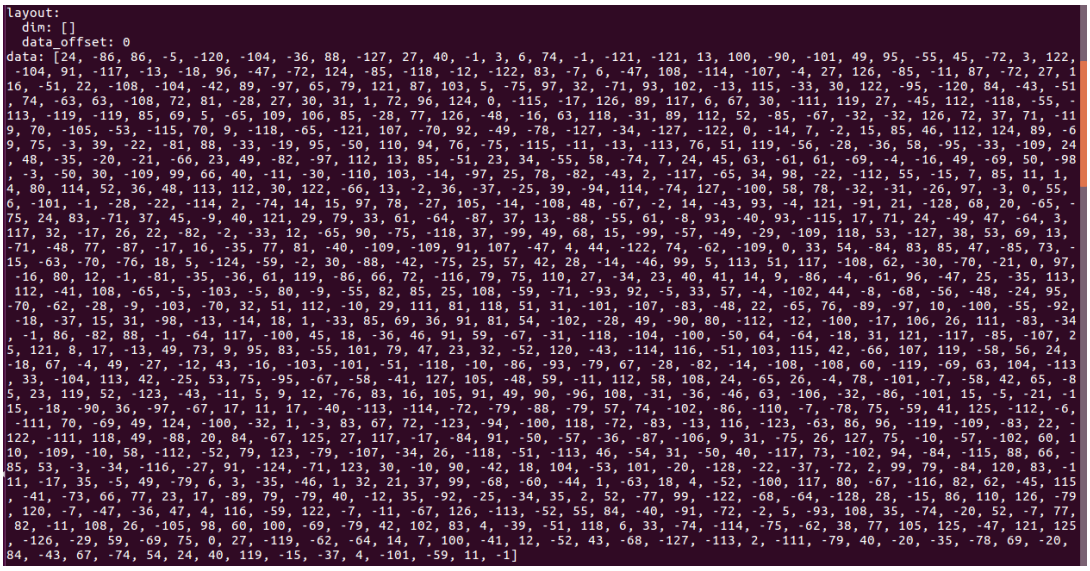


Figure 39: Encrypted data of a message sent via ROS.

To obtain this information the command "rostopic echo *nameOfTopic*" has been executed.

Finally, the last fact that make encryption evident is the encryption and decryption process shown in terminal when running each node. Figure 40 and 41 show those processes when sending and receiving four example messages, respectively. Each message sent correspond to the one received, they are delivered in order, for example, the motion data of the first one is Init and its equivalent message in the decryption process is also the first one which after decrypting shows the result: data is Init. That means that encryption and decryption are successfully working.

Besides, Figure 42 shows the content sent via ROS. The amount of characters seen in it corresponds to the data read from the encrypted file and that is going to be sent. In

this example, the mentioned file contains the result of encrypting 0, which is the encoded data of Init, what in other words is that those characters mean Init.

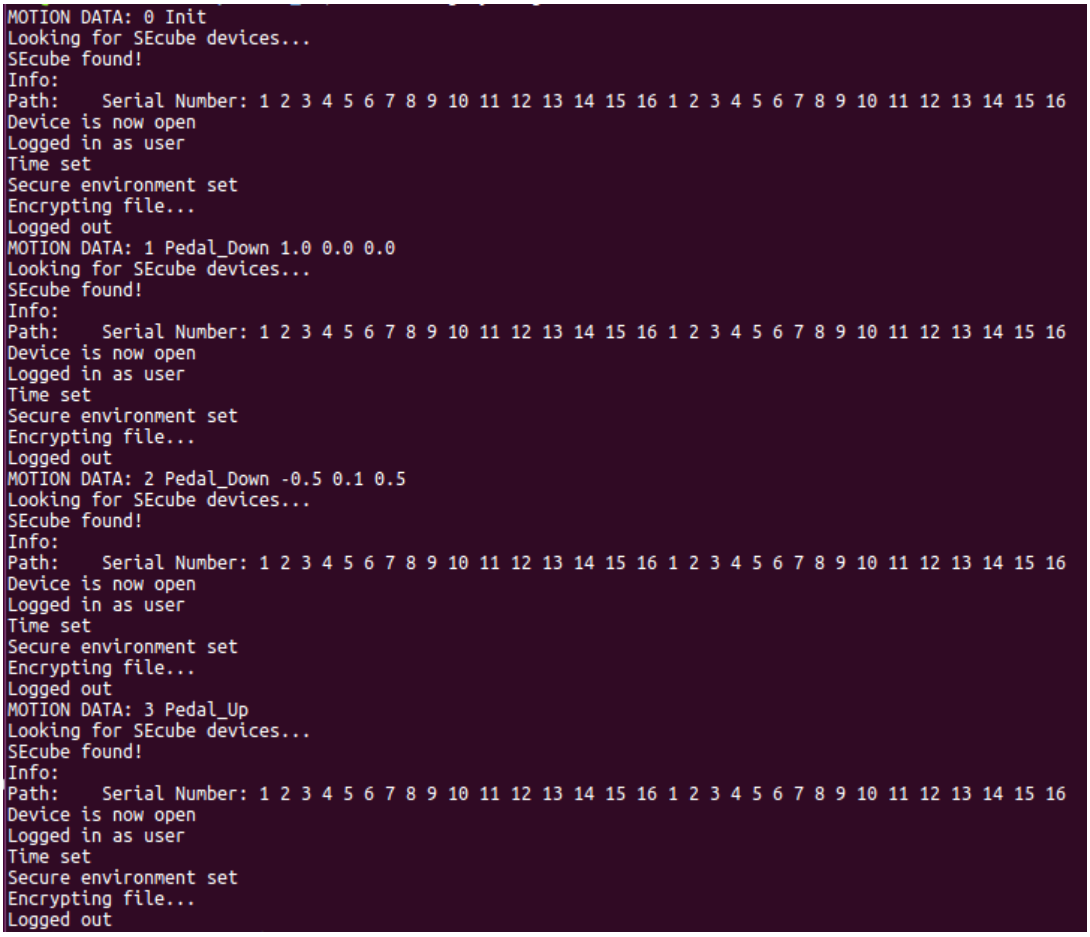


Figure 40: Encryption process of 4 messages in terminal

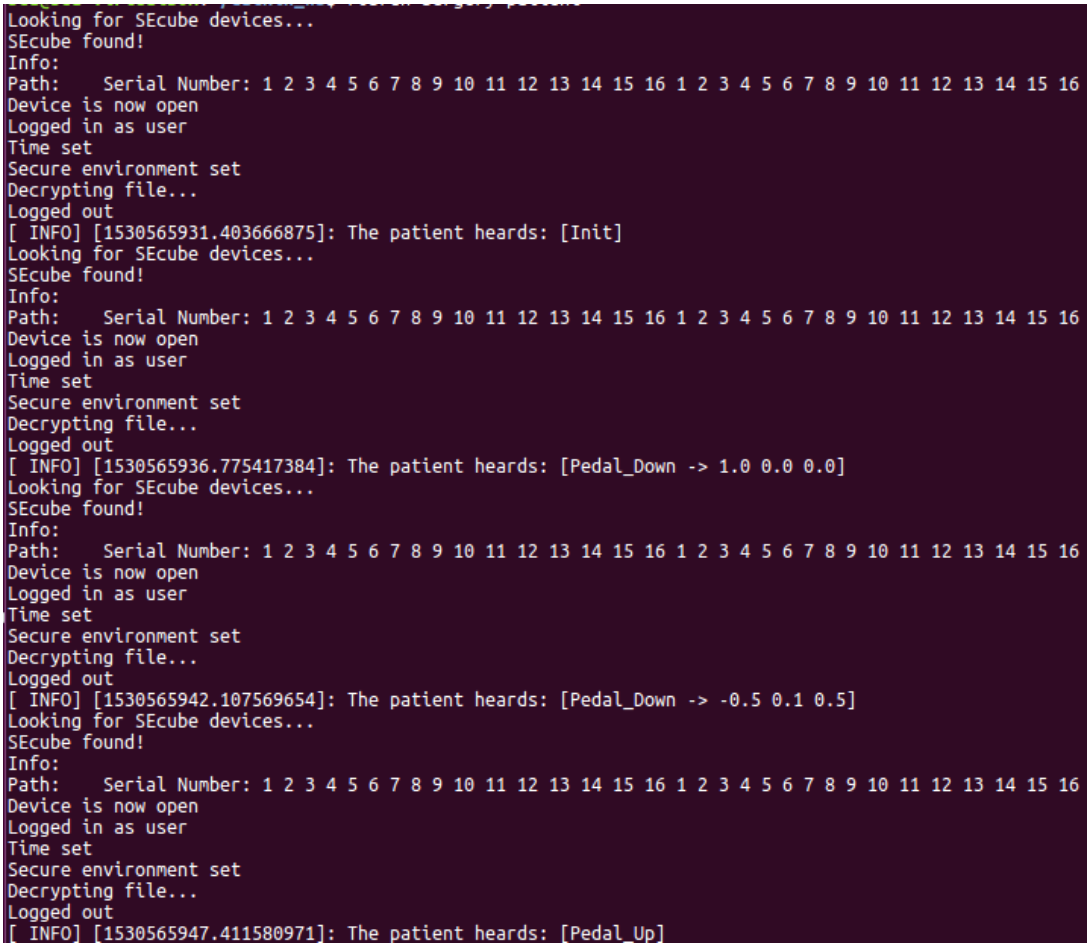


Figure 41: Decryption process of 4 messages in terminal

Figure 42: Encryption process of a message in terminal and the encrypted content that is sent.

Our results demonstrated that the exchanged data can be actually encrypted and decrypted with the **SE**cube that means it provides also with authentication given that both the ROS nodes involved in the communication need to have a valid device connected and the password to unlock it.

The results confirm that **SE**cube is a good choice for securing data but VPNs are also needed because the connection between the nodes is possible through the tunnel and also because otherwise an attacker will still be able to know the rostopic information, IP and MAC addresses of the nodes and nodes names.
Summarizing, VPNs and **SE**cube has to be implemented together in order to create a layered security structure with greater relevance in the data exchanged above all and to have communication between nodes.

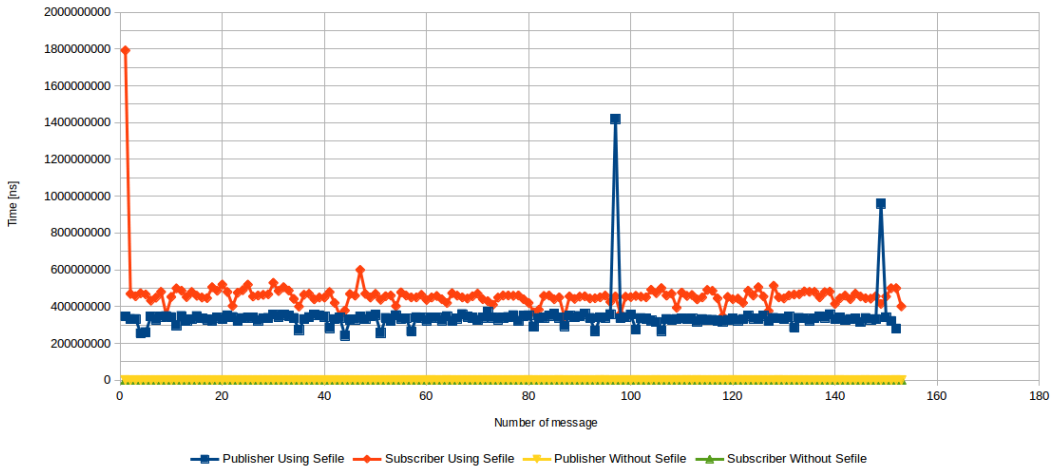
In Figure 43 it is seen that the average of publishing with encryption process is about 342 ms, subscribing with decryption 462 ms , publisher without implementing **SE**file is about 131 us and subscribing, 4 us.

In Section 2.1 it has been explained that the maximum recommended latency value is about 300 ms which means that **SE**cube implementation is not well enough as it is created at the moment and that the implementation needs to be improved because the sum of the time-lags of encryption and decryption processes far exceed the recommendation.
Those high amounts could be caused by:

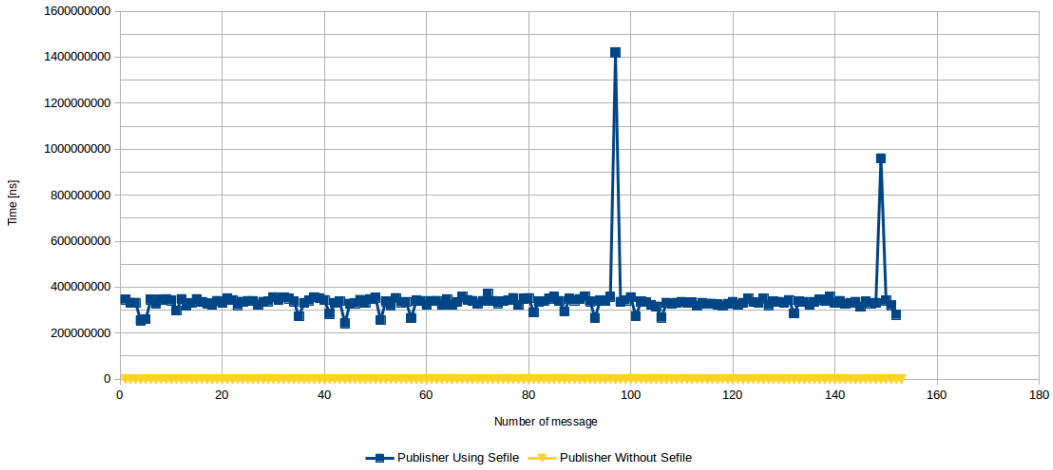
1. Process of encryption and decryption takes lot of time to be done.
2. Non-efficient code. For example: each creation of an encrypted data open and close the device.
3. Creation and Reading of files depend on the power of the computer that is being used to develop the experiment.

The steps of the first and the third one cannot be changed, because it depends of external agents or it is the available material, so the only improvement that could be applied is the second one, changing the connection to de DevKit device to be done just once instead of every time a message is received or sent. The resulting more efficient code is available in Appendix B.4 and the graphs in Graph 44.

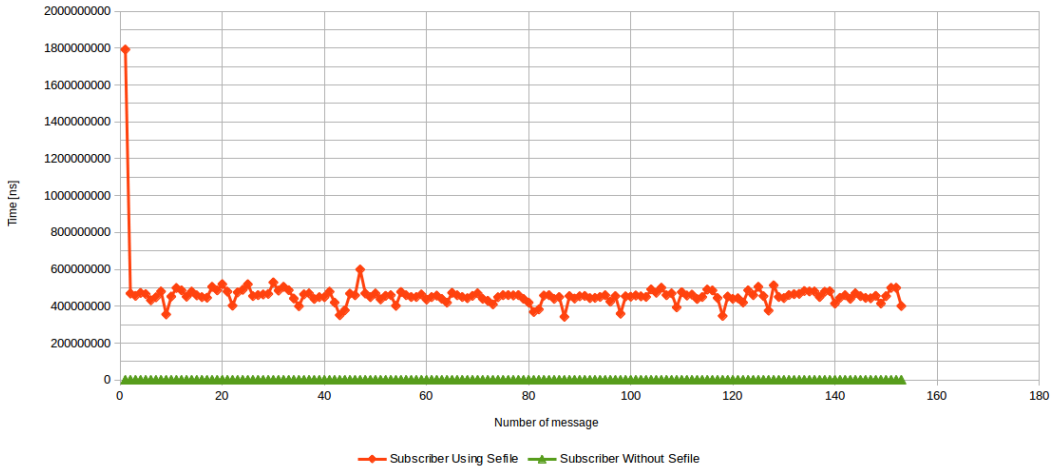
Once the changes are applied, the recollected results have an encryption average of 126 ms and decryption one of 220 ms which means a decrease of more than half time. Even so, the sum of those amounts is still higher than the recommended value mentioned, but it gets closer to it.



(a) Time-lags of publisher using and not using SEcube as well as of subscriber using and not using it.

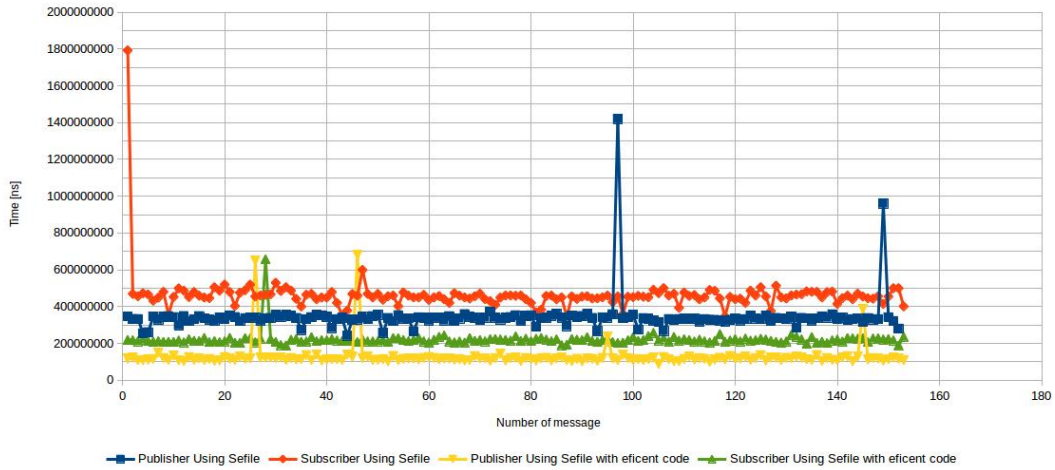


(b) Time-lags of publisher using and not using **SE**cube.

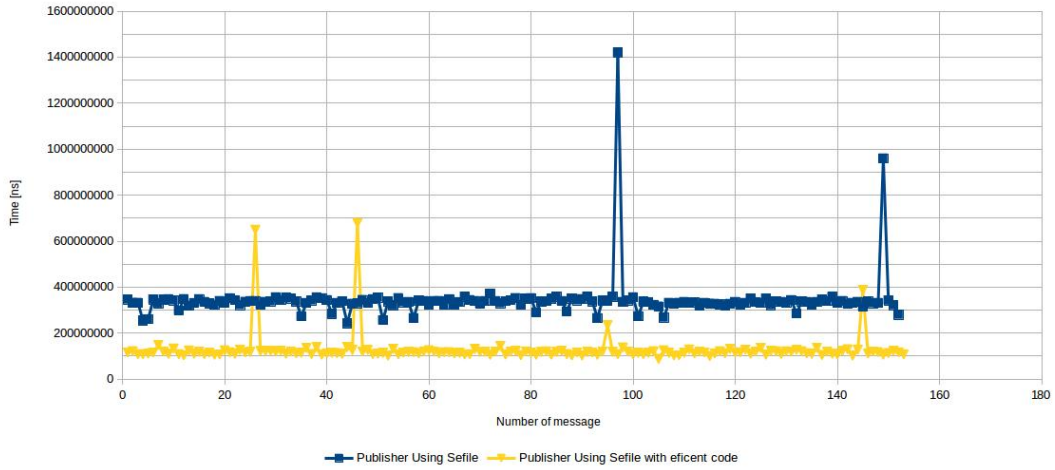


(c) Time-lags of subscriber using and not using **SE**cube.

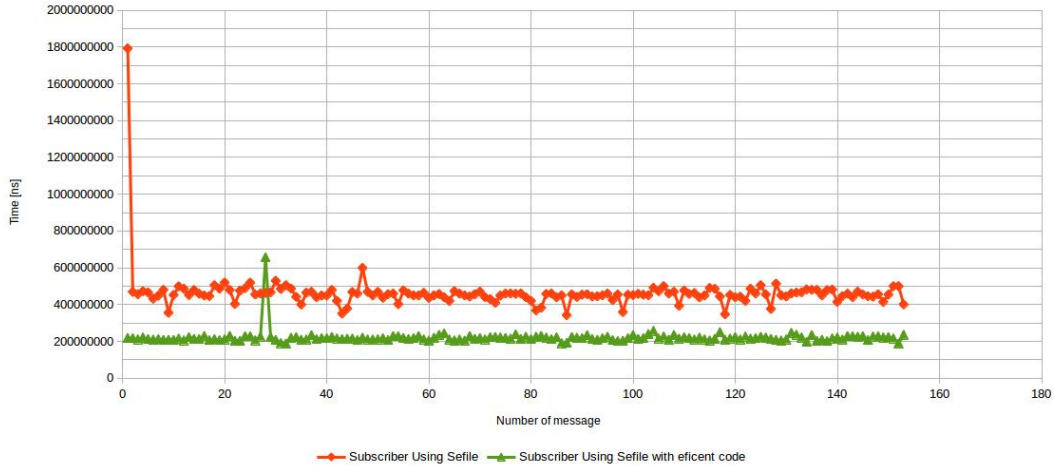
Figure 43: Time-lag graphs of encryption and decryption processes using and not using **SE**cube.



(a) Time-lags of publisher and subscriber using **SE**cubecube when connection with DevKit is every time or just once.



(b) Time-lags of publisher using **SE**cubecube when connection with DevKit is every time or just once.



(c) Time-lags of subscriber using SEcube when connection with DevKit is every time or just once.

Figure 44: Time-lag graphs of encryption and decryption processes using **SE**cube. Comparison between the process of opening and closing the device only once or every time a message is exchanged.

5 Conclusions

The implementation and analysis of this thesis have a double structure that is composed of developing a representative scenario of a tele-surgery robot within security functionalities are implemented.

To address the challenges in analysis of tele-surgery communications security, we leveraged on the **SE**cube^{TM}, an open hardware and software platform for security developments which combine different secure techniques such as signing and encryption.
We used **SE**cube together with one of its libraries, **SE**File to protect the exchanged messages based in Robot operating System. We presented a C and C++ code that can provide security funtionalities and stopped adverse consequences of malicious attacks (e.g., change motion commands) for what was also necessary the implementation of tunnelling protocols. Those were used to provide communication between the robot parts and to encrypt ROS data that could not be encrypted using **SE**File.

Besides, we further studied the delay that the process of encryption and decryption add to the communication and our analysis showed the need of improving the process. Our 150 time-lags samples on the encryption and decryption code proved that the presented implementation can apply safety but it is still not completely adequate to tele-surgery systems if it is not more improved because the delay values were a little bit higher than the recommended one.

In conclusion, the proposal method can be applied to tele-surgery systems because, even the obtained values do not meet the specifications accurately - there will be a dependency of the surgeon capabilities during the surgery -, it will not cause the system to become unsafe.

6 Future work

Many opportunities to progress have been left for future work due to the lack of time (i.e. learning how to program C and C++ efficiently will take lot of time).
Future work concerns deeper analysis based on the research described in this thesis. The following are a few directions that can be explored in the future:

- **Create a more efficient code for encryption and decryption:** it has been developed a code that works properly for the purpose it is created but it is probably not efficient enough: the system is based on C and C++ programming but the developer of the code does not have too much experience on it.

- **Implement tunneling protocols directly via the SEcube™:** the **SE**link library could be used to detect ROS messages by filtering every connection and encrypt the network streams. Implementing this library, VPN will not be necessary as security provider, it will be just implemented to allow communications between the different nodes.

- **Implement the process in a real tele-surgery robot:** the last step and final goal is to implement the resulting process to a real scenario and analyse it for future implementation in real surgeries.

Appendices

A Table of surgical procedures per year

Year	Number of surgical procedures [Thousand]
2011	359
2012	450
2013	523
2014	570
2015	652
2016	753
2017	877
2018	990

Table 3: Number of Surgical Procedures per year from 2011 to 2018.

B Defence scenario

B.1 Basic scenario

B.1.1 Publisher file in surgeon

In order to create the instructions of the surgeon, it has been created a data.txt file (that is saved in the same folder as the cpp files) which contains an example of movements. Its first column represents the time, the second, the action and the third (if needed), the movement that the patient robot has to apply.

```
0 Init
1 Pedal_Down 1.0 0.0 0.0
2 Pedal_Down -0.5 0.1 0.5
3 Pedal_Up
5 Pedal_Down 0.2 0.5 0.7
6 Pedal_Down -1 0.2 -0.3
7 Pedal_Down 0 0.1 -0.2
8 Pedal_Down 0.3 0 0
9 Pedal_Down 0.1 0 0.4
10 Pedal_Up
15 Pedal_Down 0.2 1.5 -0.7
```

```
16 E-Stop
24 Init
25 Pedal_Down 0 0 0.1
26 Pedal_Down 1.2 -0.4 0
27 Pedal_Up
28 E-Stop
29 Init
30 Pedal_Down 0.2 0.5 -0.7
31 Pedal_Down 0.6 0.9 -0.3
32 Pedal_Down 0 1 0
33 Pedal_Down 0 -0.4 -0.6
34 Pedal_Up
```

The first step of the surgeon's publisher file will be receive that instructions, which means, in this case, read the file and convert the data to the format of the communication protocol.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
#include <string>
#include <iostream>
#include <fstream>


int main(int argc, char **argv)
{
        //Variable initialization
        std::string sec[40];
        std::string data[40];
        std::string coord[40];
        int i=0;
        int t=0;

       //Read motion data of surgeon robot
        std::ifstream myReadFile;
        myReadFile.open("/home/bea/catkin_ws/src/surgery/src/data.txt");
        std::string output;

        if (myReadFile.is_open())
        {
                while (!myReadFile.eof())
                {
                        //Variable initialization for each iteraction
                        size_t pos = 0;
                        size_t pos2 = 0;
```

```
                getline(myReadFile,output);
                // The line format of the file is:
                [sec] [movement] [motion coordinates (if needed)]
                std::string delimiter = " ";
                pos = output.find(delimiter);
                sec[i] = output.substr(0, pos);
                data[i]= output.substr(pos+1, output.length());
                pos2 = data[i].find(delimiter);

                if(pos2!= std::string::npos)
                {
                        coord[i]= data[i].substr(pos2+1, data[i].length());
                        data[i] = data[i].substr(0, pos2);
                }
                else
                {
                        coord[i]= " ";
                }
                ++i;
         }
}
myReadFile.close();

i=0;

//ROS parameters and inizialization
ros::init(argc, argv, "surgeon");
ros::NodeHandle n;
ros::Publisher chatter_pub = n.advertise<std_msgs::String>("surgeon2master", 1000);
ros::Rate loop_rate(10);
std::string value = "2";

while (ros::ok())
{
  if (t==atoi(sec[i].c_str()))
  {
        std_msgs::String msg;
        std::stringstream ss;
        //Asign value to send depending on the motion executed
        if(data[i]=="E-Stop")
        {
                 value = "0"; //0=emergency stop
        }
        else if (data[i]== "Init")
        {
                  value = "1"; //1=initiazlizacion
```

```
                }
                else if (data[i]=="Pedal_Up")
                {
                        value = "2"; //2=pedal up
                }
                else if (data[i]=="Pedal_Down")
                {
                        value = "3 -> " + coord[i]; //3=pedal down with info de xyz

                }

                //Send
                ss << value ;
                msg.data = ss.str();
                ROS_INFO("%s", msg.data.c_str());
                chatter_pub.publish(msg);
                ros::spinOnce();
                loop_rate.sleep();
                ++i;
            }
            ++t;
          }
  return 0;
}
```

B.1.2 Publisher and Subscriber file in master

The master node has a subscriber to the topic *surgeon2master* and a publisher to the *master2patient* one. It reads the motion information from the first one and sends it through the second.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
#include <string>
#include <iostream>

//Variable inizialization
 std::string value;
 bool publishing=false;

//Receipt of messages over the ROS system from the surgeon
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
```

```
  ROS_INFO("The master heards: [%s]", msg->data.c_str());
  value=msg->data.c_str();
  publishing=true;
}

int main(int argc, char **argv)
{
  //ROS parameters and inizialization
  ros::init(argc, argv, "master");
  ros::NodeHandle n_listener;
  ros::NodeHandle n_publisher;
  ros::Subscriber sub = n_listener.subscribe("surgeon2master", 1000, chatterCallback);
  ros::Publisher chatter_pub = n_publisher.advertise<std_msgs::String>("master2patient", 1000);
  ros::Rate loop_rate(10);

  while (ros::ok())
  {
                std_msgs::String msg;
                std::stringstream ss;
                ss << value ;
                msg.data = ss.str();
                //Send only if received
                if (publishing){
                        chatter_pub.publish(msg);
                        ROS_INFO("Sending... %s", msg.data.c_str());
                        publishing=false;
                }
                ros::spinOnce();
                loop_rate.sleep();
  }
  return 0;
}
```

B.1.3 Subscriber file in patient

The patient node is subscribed to the topic *master2patient*, where it receives the motion information and execute it - in this case it only shows it on screen.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
#include <string>
#include <iostream>
#include <fstream>
```

```
//Receipt of messages over the ROS system from the master
void chatterCallback(const std_msgs::String::ConstPtr& msg)
{
    //Variable initialization
        std::string value = " ";
    std::string data;
        std::string coord;
    data=msg->data.c_str();

        //Convert received data to legible and comprehensive text and show it on screen
        if(data=="0")
        {
                 value = "E-Stop"; //0=emergency stop
        }
        else if (data== "1")
        {
                  value = "Init"; //1=initiazlizacion
        }
        else if (data=="2")
        {
                 value = "Pedal_Up"; //2=pedal up
        }
        else
        {
                std::string delimiter = " -> ";
                size_t pos = 0;
                pos = data.find(delimiter);
                coord = data.substr(pos+4, data.length());
                value = "Pedal_Down -> " + coord; //3=pedal_down with coordinates information
        }
        ROS_INFO("The patient heards: [%s]", value.c_str());
}

int main(int argc, char **argv)
{
  //ROS parameters and inizialization
  ros::init(argc, argv, "patient_listener");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("master2patient", 1000, chatterCallback);
  ros::spin();
  return 0;
}
```

Besides all of this, the CMakeLists.txt of each node has to be modified. As an example, the master node file results as follows.

```
cmake_minimum_required(VERSION 2.8.3)
```

```
project(surgery)

find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  std_msgs
  message_generation
)

## Generate added messages and services with any dependencies listed here
 generate_messages(
   DEPENDENCIES
   std_msgs  # Or other packages containing msgs
 )

###################################
## catkin specific configuration ##
###################################

catkin_package(
  INCLUDE_DIRS
#  LIBRARIES surgery
#  CATKIN_DEPENDS roscpp
#  DEPENDS system_lib
CATKIN_DEPENDS message_runtime
)

###########
## Build ##
###########

include_directories(
# include
  ${catkin_INCLUDE_DIRS}
)

add_executable(master src/listen&pub_master.cpp)
target_link_libraries(master ${catkin_LIBRARIES})
add_dependencies(master surgery_generate_messages_cpp)
```

B.1.4 Configuration of the network and ROS parameters

First of all, IP addresses of each interface of each machine has to be assigned. Secondly, owing to ROS configuration in multiple machines the next commands are necessary to execute in every terminal [7]:

```
user:~/catkin_ws$ export ROS_MASTER_URI=http://master_IP:11311
user:~/catkin_ws$ export ROS_IP=IP_currentMachine
```

where master_IP is the IP address of the master, where roscore is going to be run, and IP_currentMachine is the IP of the node of the machine which is being configured. After that, a route for one of the machines - surgeon or patient - is necessary because the master_IP only refers to a IP of one of the two existing networks. In the machine which is not directly connected to the network of the IP master_IP it is needed to type the command:

```
$ sudo route add -net net1 netmask mask gw IP_X
```

where net1 is the IP address of the distant network, mask is the netmask of the distant network and IP_X is the IP address of the directly connected node through which the connection will be established.
Finally, roscore and the nodes are run.

B.2 Adding the VPN

To begin, the configuration of default routes of the previous scenario is not needed any longer, so it is reset. Next, only one interface of each virtual machine is needed to be up and it has to be configured as Bridged in order to access Internet.
Secondly, a VPN server - in Master node - and two clients - in surgeon and patient nodes - are created with the steps explained by Digital Ocean in [27] and [28], as well as the firewall configuration.
Once the VPN service is created and running, the configuration has to be adapted to ROS specifications: the ROS port has to be allowed in the firewall and the ROS_MASTER_URI and ROS_IP has to be assigned using the tun interface created by the VPN service.

```
user:~/catkin_ws$ export ROS_MASTER_URI=http://master_tun_IP:11311
user:~/catkin_ws$ export ROS_IP=tun_IP_currentMachine
```

Finally, roscore and the nodes are run normally.

B.3 Adding the SEcube™. Open/Close de DevKit device every time a message is sent/received.

On the basis of the code created in basic scenario, it has been developed a ROS communication between a publisher which encrypt the data file and a subscriber which decrypt it. It has been developed taking care of the explanations mentioned in Section 3.2 and in Figure 29.

B.3.1 Publisher file in surgeon

Next code correspond to the publisher node and following it, the encryption function which is called. Due to the fact that the data to be sent are bytes, some changes had to be applied in the type of content: instead of std_msgs::String, it is used std_msgs::ByteMultiArray.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
using namespace std;
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/ByteMultiArray.h"

//SEfile libraries
extern "C" {
int encryptMessage(char filenameEncryptedChar[200]);
void print_sn(uint8_t* v);
        int decryptMessage();
}
int line=1;
int totalLines=1;
std::ifstream myReadFile, myFile;
int readMotionData(std::string* outputLine);
void createCmdFile(std::string content);


int main(int argc, char **argv)
{
 //Start ROS connection
```

```
  ros::init(argc, argv, "surgeon");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::ByteMultiArray>
  ("surgeon2master", 1024);
  ros::Rate loop_rate(10);
  std_msgs::ByteMultiArray msgEnc;
  while (ros::ok()){
        //Read data
         std::string motionData;
         while (readMotionData(&motionData)==0){
               printf("MOTION DATA: %s \n",motionData.c_str());
            //Write plain file with data of motion
              createCmdFile(motionData.c_str());
            //Encrypt file content
              char filename[200];
              char *encData;
              encryptMessage(filename);
            //Read content of the encrypted file
              filename[64]='\0';
               std::ifstream myReadFileEnc;
               myReadFileEnc.open(filename);
               char temp;
               msgEnc.data.clear();
               std::string  out;
               if (myReadFileEnc.is_open())
               {
                        while (!myReadFileEnc.eof())
                        {
                                temp = myReadFileEnc.get();
                                out=out+temp;
                                msgEnc.data.push_back(temp);
                        }
               }
              cout<<"ENCRYPTED DATA: \n"<<out<<"\n";
               myReadFileEnc.close();
            //Publish via ROS
               chatter_pub.publish(msgEnc);
               //sleep(5);
               ros::spinOnce();
               loop_rate.sleep();
         }
         ros::shutdown();
  }

  return 0;
}
int readMotionData(std::string* outputLine){
```

```
        if(line==1){
                  std::string output;
                  myReadFile.open("/home/bea/catkin_ws/src/surgery/src/data.txt");
                  if (myReadFile.is_open())
                          {
                                  while (getline(myReadFile,output))
                                  {
                                          totalLines++;
                                  }
                          }
                  myReadFile.close();
                  myFile.open("/home/bea/catkin_ws/src/surgery/src/data.txt");

        }else if (line==totalLines)
        {
                 myReadFile.close();
                 return 1;
        }

        if (myFile.is_open())
         {
                 getline(myFile,*outputLine);
         }
        ++line;
        return 0;
}

void createCmdFile(std::string content){
             //Encode data
                std::string sec;
                std::string data;
                std::string coord;
                std::string delimiter = " ";
                size_t pos = 0;
                size_t pos2 = 0;
                std::string value="2";
                pos = content.find(delimiter);
                sec = content.substr(0, pos);
                data= content.substr(pos+1, content.length());
                pos2 = data.find(delimiter);
                if(pos2!= std::string::npos)
                {
                        coord= data.substr(pos2+1, data.length());
                        data = data.substr(0, pos2);
                }
                else
```

```
                {
                        coord= " ";
                }

                if(data=="E-Stop")
                {
                         value = "0"; //0=emergency stop
                }
                else if (data== "Init")
                {
                          value = "1"; //1=initiazlizacion
                }
                else if (data=="Pedal_Up")
                {
                         value = "2"; //2=pedal up
                }
                else if (data=="Pedal_Down")
                {
                         value = "3 -> " + coord; //3=pedal down with motion info

                }

            //Create plain file
                std::stringstream ss;
                std_msgs::String msg;
                ss << value ;
                msg.data = ss.str();
                std::ofstream archivo("message.txt");
                archivo << msg.data.c_str() << '\n';
                archivo.close();
}



/* **** ENCRYPTION FUNCTION **** */
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>

#include "se3/L1.h"
#include "SEfile.h"

static uint8_t pin_user[32] = {
        'u','s','e','r', 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
```

```
};



se3_session s;
char nameEncr=NULL;


void print_sn(uint8_t* v) {
        size_t i;
        for (i = 0; i < SE3_SERIAL_SIZE; i++) {
                printf("%u ", (unsigned)v[i]);
        }
        printf("\n");


        return;
}



char * encryptMessage(){
  char filenameEncryptedChar[200];

  //open SEcube
  se3_disco_it it;
  se3_device dev;
  uint16_t return_value = 0;
  bool logged_in = false;
  printf("Looking for SEcube devices...\n");

  /* List all SEcube attached */
  L0_discover_init(&it);
  while (L0_discover_next(&it)) {
        printf("SEcube found!\nInfo:\n");
        printf("Path:\t %ls\n", it.device_info.path);
        printf("Serial Number: ");
        print_sn(it.device_info.serialno);
        //printf("\n\n");

     /* Open SEcube device */
        return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT);
        if (SE3_OK != return_value) {
                printf("Failure to open device\n");
                return(!SE3_OK);
        }
        else {
```

```
            printf("Device is now open\n");
    }
/* Log in as user */
    return_value = L1_login(&s, &dev, pin_user, SE3_ACCESS_USER);
    if (SE3_OK != return_value) {
            printf("Failure to log in as user\n");
            return(!SE3_OK);
    }
    else {
            printf("Logged in as user\n");
    }

/* Set time for crypto functions */
    return_value = L1_crypto_set_time(&s, (uint32_t)time(0));
    if (SE3_OK != return_value) {
            printf("Failure to set time\n");
            L1_logout(&s);
            return(!SE3_OK);
    }
    else {
            printf("Time set\n");
    }
    /* */

/* Set Secure Environment*/
    return_value = secure_init(&s, -1, SE3_ALGO_MAX+1);
    if (SE3_OK != return_value) {
            printf("Failure to set secure environment\n");
            return(!SE3_OK);
    }
    else {
            printf("Secure environment set\n");
    }
    /* */


/* SEfile */
    char plainFilePath ="/home/bea/catkin_ws/message.txt";
    SEFILE_FHANDLE sefile_file, sefile_file2;
    uint16_t len_enc_filename = 0;
    char filenameEncrypted;
    uint32_t plainFileLength, bytesRead;
    uint8_t *buffer;
    char filenameDecryptedChar[200];

    /*Encryption*/
    printf("Encrypting file...\n");
```

```
  FILE * plainfile = fopen("message.txt", "r");
  char *plainFileContent;
  long plainfilelen;
  if (!plainfile){
          return false;
  }
  //read plain file & save in array of bytes
  fseek(plainfile, 0, SEEK_END);          // Jump to the end of the file
  plainfilelen = ftell(plainfile);        // Get the current byte offset in the file
  rewind(plainfile);                      // Jump back to the beginning of the file

  plainFileContent = (char *)malloc((plainfilelen+1)*sizeof(char));
   // Enough memory for file + \0

  fread(plainFileContent, plainfilelen, 1, plainfile); // Read in the entire file

  fclose(plainfile); // Close the file


  if(crypto_filename("/home/bea/catkin_ws/message.txt", filenameEncryptedChar,
   &len_enc_filename) != 0){
                                  return false;
  }

  // Open encrypted file

  if(secure_open("/home/bea/catkin_ws/message.txt", &sefile_file, SEFILE_WRITE,
   SEFILE_NEWFILE) != 0){
          return false;
  }

  if(secure_write(&sefile_file, (uint8_t*)plainFileContent, plainfilelen) != 0){
                  secure_close(&sefile_file);
                  return false;
  }

  secure_close(&sefile_file);

  printf("Encrypted ");
  printf("in ");
  printf( filenameEncryptedChar );
  printf("\n");

/* Log out */
  secure_finit();
  return_value = L1_logout(&s);
```

```
        if (SE3_OK != return_value) {
                printf("Failure to log out\n");
                return(!SE3_OK);
        }
        else {
                printf("Logged out\n");
        }
        /* */

    /* Close device */
        L0_close(&dev);
        /* */
  }

  return filenameEncryptedChar;
}
```

The files provided by the library **SE**file, such as SEfile.c and SEfile.h are compulsory, as well as the se3 folder with the files of SHA, AES, L0, L1 and more files of **SE**cube.

Besides, due to the fact that **SE**cube and **SE**file files are created in C but ROS files has been created with C++, the CMakeLists.txt has been modified including

```
add_library(encrypt src/SEfile/se3/aes256.c src/SEfile/se3/sha256.c src/SEfile/se3/crc16.c
src/SEfile/se3/pbkdf2.c src/SEfile/se3/se3comm.c src/SEfile/se3/se3_common.c
src/SEfile/se3/L0.c src/SEfile/se3/L1.c  src/SEfile/SEfile.c src/SEfile/encryptMessage.c)
```

and changing the following lines

```
add_executable(surgeon src/SEfile/se3/aes256.c src/SEfile/se3/sha256.c
src/SEfile/se3/crc16.c src/SEfile/se3/pbkdf2.c src/SEfile/se3/se3comm.c
src/SEfile/se3/se3_common.c src/SEfile/se3/L0.c src/SEfile/se3/L1.c
src/SEfile/SEfile.c src/SEfile/encryptMessage.c src/publisher_surgeon2master.cpp)

target_link_libraries(surgeon encrypt ${catkin_LIBRARIES})
```

B.3.2 Publisher and Subscriber file in Master

The master node code is the same as the one presented in the Basic Scenario of Appendix B.1 but with the message type variable changed to std_msgs::ByteMultiArray, the includes that correspond and how to read the message content. The message contents are read as an array:

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
```

```
#include <string>
#include <iostream>
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/ByteMultiArray.h"

std::string value;
bool publishing=false;
std_msgs::ByteMultiArray msgEnc;

void chatterCallback(const std_msgs::ByteMultiArray::ConstPtr& msg)
{

        char Arr[1024];
        int i=0;
        memset(Arr,0,1024);
        msgEnc.data.clear();
        for(std::vector<signed char>::const_iterator it = msg->data.begin();
        it != msg->data.end(); ++it)
        {
                Arr[i] = *it;
                i++;
        }
        for(i = 0; i<1024; i++){
                msgEnc.data.push_back(Arr[i]) ;
        }

        ROS_INFO("The master heards something");
        publishing=true;
}

int main(int argc, char **argv)
{

  ros::init(argc, argv, "master");
  ros::NodeHandle n_listener;
  ros::NodeHandle n_publisher;
//Subscribe
  ros::Subscriber sub = n_listener.subscribe("surgeon2master", 1024, chatterCallback,
  ros::TransportHints().tcpNoDelay());
//Publish
  ros::Publisher chatter_pub = n_publisher.advertise<std_msgs::ByteMultiArray>
  ("master2patient", 1024);
  ros::Rate loop_rate(10);

  while (ros::ok())
  {
```

```
                std::stringstream ss;
                if (publishing){
                        chatter_pub.publish(msgEnc);
                        ROS_INFO("Sending... ");
                        publishing=false;
                }
                ros::spinOnce();
                loop_rate.sleep();
  }

  return 0;
}
```

There is no more difference because the master do not have to work with the data so it does not matter that it does not understand what it is said in the communication between the surgeon and the patient nodes, there is no need to use **SE**file nor **SE**cube.

B.3.3 Subscriber file in patient

Next code correspond to the subscriber node and following it, the decryption function which is called.

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include <sstream>
#include <cstdlib>
#include <string>
#include <iostream>
#include <fstream>
#include <sstream>
using namespace std;
#include "std_msgs/MultiArrayLayout.h"
#include "std_msgs/MultiArrayDimension.h"
#include "std_msgs/ByteMultiArray.h"

//SEfile libraries
extern "C" {
        int decryptMessage(char Arr[1024]);
        void print_sn(uint8_t* v);
}

string decodeData(std::string data);
void readDecryptedData(std::string* out);

char Arr[1024];
int i=0;
```

```
//Receipt of messages over the ROS system from the master
void chatterCallback(const std_msgs::ByteMultiArray::ConstPtr& msg)
{
     //Read content of message
        i=0;
        memset(Arr,0,1024);
        for(std::vector<signed char>::const_iterator it = msg->data.begin();
        it != msg->data.end(); ++it)
        {
                Arr[i] = *it;
                i++;
        }

     //Decrypt received data
         decryptMessage(Arr);

     //Read decrypted data
         std::string out="";
         readDecryptedData(&out);

     //Decode data
         out=decodeData(out);

     //Final action
         ROS_INFO("The patient heards: [%s]", out.c_str());
}

int main(int argc, char **argv)
{
  ros::init(argc, argv, "patient");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("master2patient", 1024, chatterCallback);
  ros::spin();
  return 0;
}

string decodeData(std::string data){
         std::string value = "";

         if(data=="0")
         {
                value = "E-Stop"; //0=emergency stop
         }
         else if (data== "1")
         {
                value = "Init"; //1=initiazlizacion
         }
```

```
        else if (data=="2")
        {
                value = "Pedal_Up"; //2=pedal up
        }
        else
        {
                std::string delimiter = " -> ";
                size_t pos = 0;
                pos = data.find(delimiter);
                std::string coord="";
                if (pos != std::string::npos){
                        coord = data.substr(pos+4, data.length());
                        value = "Pedal_Down -> " + coord;
                }else{
                        value="COMMAND NOT VALID";
                }
        }
       return value;
}

void readDecryptedData(std::string* out){
        std::ifstream myReadFile;
        myReadFile.open("/home/bea/catkin_ws/message.txt");
        std::string output;
        std::string line;
        if (myReadFile.is_open())
        {
                *out="";
                while (!myReadFile.eof())
                {
                                getline(myReadFile,line);
                                *out=*out + line;
            }
        }
}



/* **** DECRYPTION FUNCTION **** */
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <sys/stat.h>
#include "se3/L1.h"
#include "SEfile.h"
```

```
static uint8_t pin_user[32] = {
        'u','s','e','r', 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
};

se3_session s;

void print_sn(uint8_t* v) {
        size_t i;
        for (i = 0; i < SE3_SERIAL_SIZE; i++) {
                printf("%u ", (unsigned)v[i]);
        }
        printf("\n");


        return;
}


int decryptMessage(char* Arr[1024]){

        char filenameEncryptedChar[64];

        //open SEcube
        se3_disco_it it;
        se3_device dev;
        uint16_t return_value = 0;
        bool logged_in = false;
        printf("Looking for SEcube devices...\n");



        /* List all SEcube attached */
        L0_discover_init(&it);
        while (L0_discover_next(&it)) {
                printf("SEcube found!\nInfo:\n");
                printf("Path:\t %ls\n", it.device_info.path);
                printf("Serial Number: ");
                print_sn(it.device_info.serialno);
                //printf("\n\n");

                /* Open SEcube device */
                return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT);
                if (SE3_OK != return_value) {
                        printf("Failure to open device\n");
                        return(!SE3_OK);
```

```
}
else {
        printf("Device is now open\n");
}
/* Log in as user */
return_value = L1_login(&s, &dev, pin_user, SE3_ACCESS_USER);
if (SE3_OK != return_value) {
        printf("Failure to log in as user\n");
        return(!SE3_OK);
}
else {
        printf("Logged in as user\n");
}

        /* Set time for crypto functions */
return_value = L1_crypto_set_time(&s, (uint32_t)time(0));
        if (SE3_OK != return_value) {
                printf("Failure to set time\n");
                L1_logout(&s);
                return(!SE3_OK);
        }
        else {
                printf("Time set\n");
        }
        /* */



        /* Set Secure Environment*/
        return_value = secure_init(&s, -1, SE3_ALGO_MAX+1);
                if (SE3_OK != return_value) {
                        printf("Failure to set secure environment\n");
                        return(!SE3_OK);
                }
                else {
                        printf("Secure environment set\n");
                }


        /* */


        /* SEfile */

                char plainFilePath ="/home/bea/catkin_ws/message.txt";
                SEFILE_FHANDLE sefile_file, sefile_file2;
                uint16_t len_enc_filename = 0;
```

```
char filenameEncrypted;
uint32_t plainFileLength, bytesRead;
uint8_t *buffer;
char filenameDecryptedChar[200];

/* CREATE FILE WITH ENCRYPTED CONTENT */
if(crypto_filename("message.txt", filenameEncryptedChar,
 &len_enc_filename) != 0){
        return false;
}
filenameEncryptedChar[len_enc_filename]='\0';

FILE *fileW;
umask(0000);
fileW = fopen(filenameEncryptedChar,"wb");
fwrite(Arr,1,1024,fileW);
fclose(fileW);
/* */

/* Decryption  */
   printf("Decrypting file... \n");

   // Open encrypted file
   if(secure_open("message.txt", &sefile_file2,
   SEFILE_READ, SEFILE_OPEN ) != 0){
           printf("Failure opening Encrypted File");
           return false;
   }



   // Get the size of the resulting plain file
   int tmpState=get_filesize(&sefile_file2, &plainFileLength);
   if(tmpState != 0){
           printf("ERROR SEFILE %d\n",tmpState);
           secure_close(&sefile_file2);
           printf("Failure getting size of the File");
           return false;
   }



   // Get original filename
   if(decrypt_filehandle(&sefile_file2,
   filenameDecryptedChar) != 0){
           secure_close(&sefile_file2);
           printf("Failure decrypting name of the File");
```

```
                return false;
            }

            buffer = (uint8_t *)calloc(plainFileLength,
            sizeof(uint8_t));
            if(secure_read(&sefile_file2, buffer, plainFileLength,
            &bytesRead) != 0) {
                secure_close(&sefile_file2);
                printf("Failure secure_read");
                return false;
            }

            // Create a plain file to store the plain content
            FILE *fileDec;
            fileDec = fopen(filenameDecryptedChar, "w");
            fprintf(fileDec, buffer);
            fclose(fileDec);


            secure_close(&sefile_file2);
            //chmod(filenameEncryptedChar, 0666);
           // char dir[100]="/home/bea/catkin_ws/";
            //strcat(dir,&filenameEncryptedChar);
        /*    int ret = remove(filenameEncryptedChar);
            printf("REMOVE %d\n",ret);
*/
            /* */



        /* Log out */
        secure_finit();
        return_value = L1_logout(&s);
        if (SE3_OK != return_value) {
                printf("Failure to log out\n");
                return(!SE3_OK);
        }
        else {
                printf("Logged out\n");
        }
        /* */

        /* Close device */
        L0_close(&dev);
        /* */
    }
```

```
return 0;
}
```

Besides, as in publisher node, a modification of the file CMakeList.txt is needed including the corresponding decryption files.

B.4 Adding the SEcube™. Open/Close de DevKit device only once.

B.4.1 Publisher file in surgeon

The main part of the code that changes comparing to the section B.3 is the encryption function due to it has become three different functions: openSEcube(), closeSEcube() and encryptMessage(). In the file of ROS code, the only changes are where to call the two first mention functions:

```
/* **** ROS MAIN **** */
int main(int argc, char **argv)
{
 //Start ROS connection
  ros::init(argc, argv, "surgeon");
  ros::NodeHandle n;
  ros::Publisher chatter_pub = n.advertise<std_msgs::ByteMultiArray>("surgeon2master", 1024);
  ros::Rate loop_rate(10);
  std_msgs::ByteMultiArray msgEnc;

  while (ros::ok()){
         //Read data
          std::string motionData;
          if (openSEcube()!=0){
                return 1;
          }
          while (readMotionData(&motionData)==0){
          printf("MOTION DATA: %s \n",motionData.c_str());
          boost::chrono::high_resolution_clock::time_point start = boost::chrono::
high_resolution_clock::now();

              //Write plain file with data of motion
                createCmdFile(motionData.c_str());
              //Encrypt file content
                char filename[200];
                char *encData;
                encryptMessage(filename);
              //Read content of the encrypted file
```

```
                filename[64]='\0';
                 std::ifstream myReadFileEnc;
                 myReadFileEnc.open(filename);
                 char temp;
                 msgEnc.data.clear();
                 std::string  out;
                 if (myReadFileEnc.is_open())
                 {
                         while (!myReadFileEnc.eof())
                         {
                                 temp = myReadFileEnc.get();
                                 out=out+temp;
                                 msgEnc.data.push_back(temp);
                         }
                 }
                // cout<<"ENCRYPTED DATA: \n"<<out<<"\n";
                 myReadFileEnc.close();
               //Publish via ROS
               boost::chrono::high_resolution_clock::time_point finish = boost::chrono
               ::high_resolution_clock::now();
                 chatter_pub.publish(msgEnc);
                 //sleep(5);
                 ros::spinOnce();
                 loop_rate.sleep();

                 archivotime << boost::chrono::duration_cast<boost::chrono::nanoseconds>
                 (finish-start) << '\n';

          }
archivotime.close();
           ros::shutdown();
        closeSEcube();
  }
  return 0;
}

/* **** ENCRYPTION FUNCTIONS **** */
#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include "se3/L1.h"
#include "SEfile.h"

static uint8_t pin_user[32] = {
        'u','s','e','r', 0,0,0,0, 0,0,0,0, 0,0,0,0,
```

```
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
};

se3_session s;
se3_device dev;
uint16_t return_value = 0;
char nameEncr=NULL;




void print_sn(uint8_t* v) {
        size_t i;
        for (i = 0; i < SE3_SERIAL_SIZE; i++) {
                printf("%u ", (unsigned)v[i]);
        }
        printf("\n");


        return;
}

int openSEcube(){
  //open SEcube and configure the environment
  se3_disco_it it;
  bool logged_in = false;
  printf("Looking for SEcube devices...\n");

  /* List all SEcube attached */
  L0_discover_init(&it);
  while (L0_discover_next(&it)) {
        printf("SEcube found!\nInfo:\n");
        printf("Path:\t %ls\n", it.device_info.path);
        printf("Serial Number: ");
        print_sn(it.device_info.serialno);
        //printf("\n\n");

        /* Open SEcube device */
        return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT);
        if (SE3_OK != return_value) {
                printf("Failure to open device\n");
                return(!SE3_OK);
        }
        else {
                printf("Device is now open\n");
        }
        /* Log in as user */
```

```
        return_value = L1_login(&s, &dev, pin_user, SE3_ACCESS_USER);
        if (SE3_OK != return_value) {
                printf("Failure to log in as user\n");
                return(!SE3_OK);
        }
        else {
                printf("Logged in as user\n");
        }

                /* Set time for crypto functions */
        return_value = L1_crypto_set_time(&s, (uint32_t)time(0));
        if (SE3_OK != return_value) {
                printf("Failure to set time\n");
                L1_logout(&s);
                return(!SE3_OK);
        }
        else {
                printf("Time set\n");
        }
        /* */



        /* Set Secure Environment*/
        return_value = secure_init(&s, -1, SE3_ALGO_MAX+1);
        if (SE3_OK != return_value) {
                printf("Failure to set secure environment\n");
                return(!SE3_OK);
        }
        else {
                printf("Secure environment set\n");
        }
        /* */
  }
  return 0;
}

int encryptMessage(char filenameEncryptedChar[200]){
        /* SEfile */
        char plainFilePath ="/home/bea/catkin_ws/message.txt";
        SEFILE_FHANDLE sefile_file, sefile_file2;
        uint16_t len_enc_filename = 0;
        char filenameEncrypted;
        uint32_t plainFileLength, bytesRead;
        char filenameDecryptedChar[200];

        /*Encryption*/
```

```
        printf("Encrypting file...\n");

        FILE * plainfile = fopen("message.txt", "r");
        char *plainFileContent;
        long plainfilelen;
        if (!plainfile){
                return false;
        }
        //read plain file & save in array of bytes
        fseek(plainfile, 0, SEEK_END);          // Jump to the end of the file
        plainfilelen = ftell(plainfile);        // Get the current byte offset in the file
        rewind(plainfile);                      // Jump back to the beginning of the file

        plainFileContent = (char *)malloc((plainfilelen+1)*sizeof(char));

        fread(plainFileContent, plainfilelen, 1, plainfile); // Read in the entire file

        fclose(plainfile); // Close the file


        if(crypto_filename("message.txt", filenameEncryptedChar, &len_enc_filename) != 0){
                                        return false;
        }

        // Open encrypted file

        if(secure_open("message.txt", &sefile_file, SEFILE_WRITE, SEFILE_NEWFILE) != 0){
                return false;
        }

        if(secure_write(&sefile_file, (uint8_t*)plainFileContent, plainfilelen) != 0){
                secure_close(&sefile_file);
                return false;
        }

        secure_close(&sefile_file);

        //}

return 0;
}

int closeSEcube(){
        /* Log out */
        secure_finit();
        return_value = L1_logout(&s);
        if (SE3_OK != return_value) {
```

```
                printf("Failure to log out\n");
                return(!SE3_OK);
        }
        else {
                printf("Logged out\n");
        }
        /* */

        /* Close device */
        L0_close(&dev);
        /* */
return 0;
}
```

B.4.2 Publisher and Subscriber file in Master

Master's code is exactly the same one as in section B.3.

B.4.3 Subscriber file in patient

The main part of the code that changes comparing to the section B.3 is the encryption function due to it has become three different functions: openSEcube(), closeSEcube() and decryptMessage(). In the file of ROS code, the only changes are where to call the two first mention functions:

```
/* **** ROS MAIN **** */
int main(int argc, char **argv)
{
  openSEcube();
  ros::init(argc, argv, "patient");
  ros::NodeHandle n;
  ros::Subscriber sub = n.subscribe("master2patient", 1024, chatterCallback);
  ros::spin();
  closeSEcube();
  return 0;
}
/* **** DECRYPTION FUNCTIONS **** */

#include <stdlib.h>
#include <stdio.h>
#include <stddef.h>
#include <stdint.h>
#include <stdbool.h>
#include <sys/stat.h>
#include "se3/L1.h"
```

```
#include "SEfile.h"

static uint8_t pin_user[32] = {
        'u','s','e','r', 0,0,0,0, 0,0,0,0, 0,0,0,0,
        0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,0
};

se3_session s;
se3_device dev;
uint16_t return_value = 0;

void print_sn(uint8_t* v) {
        size_t i;
        for (i = 0; i < SE3_SERIAL_SIZE; i++) {
                printf("%u ", (unsigned)v[i]);
        }
        printf("\n");


        return;
}

int openSEcube(){

//open SEcube
se3_disco_it it;
bool logged_in = false;
printf("Looking for SEcube devices...\n");



/* List all SEcube attached */
L0_discover_init(&it);
  while (L0_discover_next(&it)) {
        printf("SEcube found!\nInfo:\n");
        printf("Path:\t %ls\n", it.device_info.path);
        printf("Serial Number: ");
        print_sn(it.device_info.serialno);
        //printf("\n\n");

        /* Open SEcube device */
        return_value = L0_open(&dev, &(it.device_info), SE3_TIMEOUT);
        if (SE3_OK != return_value) {
                printf("Failure to open device\n");
                return(!SE3_OK);
        }
        else {
```

```
                printf("Device is now open\n");
        }
        /* Log in as user */
        return_value = L1_login(&s, &dev, pin_user, SE3_ACCESS_USER);
        if (SE3_OK != return_value) {
                printf("Failure to log in as user\n");
                return(!SE3_OK);
        }
        else {
                printf("Logged in as user\n");
        }

        /* Set time for crypto functions */
        return_value = L1_crypto_set_time(&s, (uint32_t)time(0));
        if (SE3_OK != return_value) {
                printf("Failure to set time\n");
                L1_logout(&s);
                return(!SE3_OK);
        }
        else {
                printf("Time set\n");
        }
        /* */

        /* Set Secure Environment*/
        return_value = secure_init(&s, -1, SE3_ALGO_MAX+1);
        if (SE3_OK != return_value) {
                printf("Failure to set secure environment\n");
                return(!SE3_OK);
        }
        else {
                printf("Secure environment set\n");
        }
        /* */
   }
return 0;
}


int decryptMessage(char* Arr[1024]){

        char filenameEncryptedChar[64];

   /* SEfile */

        char plainFilePath ="/home/bea/catkin_ws/message.txt";
        SEFILE_FHANDLE sefile_file, sefile_file2;
```

```
        uint16_t len_enc_filename = 0;
        char filenameEncrypted;
        uint32_t plainFileLength, bytesRead;
        uint8_t *buffer;
        char filenameDecryptedChar[200];

        /* CREATE FILE WITH ENCRYPTED CONTENT */

        if(crypto_filename("message.txt", filenameEncryptedChar, &len_enc_filename) != 0){
                                                                    return false;
        }

        filenameEncryptedChar[len_enc_filename]='\0';

        FILE *fileW;
        umask(0000);
        fileW = fopen(filenameEncryptedChar,"wb");
        fwrite(Arr,1,1024,fileW);
        fclose(fileW);
        /* */
        /* Decryption  */
           printf("Decrypting file... \n");

           // Open encrypted file
           if(secure_open("message.txt", &sefile_file2, SEFILE_READ, SEFILE_OPEN ) != 0){
                   printf("Failure opening Encrypted File");
                   return false;
           }



           // Get the size of the resulting plain file
           int tmpState=get_filesize(&sefile_file2, &plainFileLength);
           if(tmpState != 0){
                   printf("ERROR SEFILE %d\n",tmpState);
                   secure_close(&sefile_file2);
printf("Failure getting size of the File");
                   return false;
           }



           // Get original filename
           if(decrypt_filehandle(&sefile_file2, filenameDecryptedChar) != 0){
                   secure_close(&sefile_file2);
printf("Failure decrypting name of the File");
                   return false;
```

```
        }

        buffer = (uint8_t *)calloc(plainFileLength, sizeof(uint8_t));
        if(secure_read(&sefile_file2, buffer, plainFileLength, &bytesRead) != 0) {
                secure_close(&sefile_file2);
printf("Failure secure_read");
                return false;
        }

        // Create a plain file to store the plain content
        FILE *fileDec;
        fileDec = fopen(filenameDecryptedChar, "w");
        fprintf(fileDec, buffer);
        fclose(fileDec);


        secure_close(&sefile_file2);

    /* */

return 0;
}

int closeSEcube(){

    /* Log out */
    secure_finit();
    return_value = L1_logout(&s);
    if (SE3_OK != return_value) {
            printf("Failure to log out\n");
            return(!SE3_OK);
    }
    else {
            printf("Logged out\n");
    }
    /* */

    /* Close device */
    L0_close(&dev);
    /* */
    return 0;
}
```

References

[1] Maroun Bechara Patrick Aïdan. Gasless trans-axillary robotic thyroidectomy: The introduction and principle. pages 229–235, 6 2017.

[2] AnThea Gerrie. Four-armed cyborg helping surgeons to conquer lung cancer: New 'da vinci' robot allows patients to leave hospital just two days after major surgery. *The Mail On Sunday*, 8 2015.

[3] Inc. Intuitive Surgical. Intuituve surgical. URL: `https://intuitivesurgical.com/`.

[4] Inc. Intuitive Surgical. Annual report 2013. URL: `http://www.annualreports.com/HostedData/AnnualReportArchive/i/NASDAQ_ISRG_2013.pdf`.

[5] Inc. Intuitive Surgical. Annual report 2016. URL: `http://www.annualreports.com/HostedData/AnnualReports/PDF/NASDAQ_ISRG_2016.pdf`.

[6] Lucas Mearian. Hospital tests lag time for robotic surgery 1,200 miles away from doctor. URL: `https://www.computerworld.com/article/2927471/healthcare-it/robot-performs-test-surgery-1200-miles-away-from-doctor.html`.

[7] Isaac Saico. Ros - ros wiki. URL: `http://wiki.ros.org/ROS`.

[8] Blu5 View Pte. Ltd. System on chip embedded environment - secube^{TM}. URL: `https://www.secube.eu/`.

[9] VPN Comparison Staff. Pure vpn review – scam or not? URL: `http://www.vpncomparison.org/provider/purevpn/`.

[10] Mark Allinson. All's well that ros well: Robot operating system taking over the world. URL: `https://roboticsandautomationnews.com/2016/10/30/alls-well-that-ros-well-robot-operating-system-taking-over-the-world/8369/`.

[11] KWC. Raven ii open-source surgical robots. URL: http://www.ros.org/news/2012/01/raven-ii-open-source-surgical-robots.html.

[12] Russell H. Taylor Zihan Chen, Anton Deguet and Peter Kazanzides. Software architecture of the da vinci research kit. URL: https://raw.githubusercontent.com/wiki/jhu-dvrk/sawIntuitiveResearchKit/chen-deguet-etal-irc-2017.pdf.

[13] The Medical Futurist Institute. The medical futurist. URL: http://medicalfuturist.com/robotics-healthcare/.

[14] Eric J. Moore. Robotic surgery. URL: https://www.britannica.com/science/robotic-surgery#ref1225036.

[15] Steven E. Butner and Moji Ghodoussi. Transforming a surgical robot for human telesurgery. URL: https://www.researchgate.net/publication/3299375_Transforming_a_Surgical_Robot_for_Human_Telesurgery.

[16] Isaac Saico. Ros - master slave apis - ros wiki. URL: http://wiki.ros.org/ROS/Master_Slave_APIs.

[17] Isaac Saico. Ros - master apis - ros wiki. URL: http://wiki.ros.org/ROS/Master_API.

[18] Isaac Saico. Ros - slave apis - ros wiki. URL: http://wiki.ros.org/ROS/Slave_API.

[19] Isaac Saico. Ros - parameter server apis - ros wiki. URL: http://wiki.ros.org/ROS/Parameter%20Server%20API.

[20] Sebastian Taurer Severin Kacianka Stefan Rass Peter Schartner Bernhard Dieber, Benjamin Breiling. Security for the robot operating system. *Robotics and Autonomous Systems, volume 98, Pages 192-203*, 10 2017.

[21] Tariq Yusuf Junjie Yan Tadayoshi Kohno Howard Jay Chizeck Tamara Bonaci, Jeffrey Herron. To make a robot secure: An experimental analysis of cyber security threats against teleoperated surgical robotics, 2015.

[22] Blu5 View Pte. Ltd. Secube^{TM} development kit l2 user manual. URL: https://www.secube.eu/download/SEcube-Development-Kit-L2-Manual-PUBLIC-v0.3.pdf.pdf.

[23] OpenVPN Inc. What is openvpn? URL: https://openvpn.net/index.php/open-source/documentation.html.

[24] Cisco. Network address translation (nat) faq. URL: https://www.cisco.com/c/en/us/support/docs/ip/network-address-translation-nat/26704-nat-faq-00.html.

[25] Homa Alemzadeh. Data-driven resiliency assessment of medical cyber-physical systems, 2016.

[26] National Institute of Standards and Technology (NIST). Specification for the advanced encryption standard (aes). URL: https://web.archive.org/web/20150407153905/http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[27] Digital Ocean. Initial server setup with ubuntu 16.04. URL: https://www.digitalocean.com/community/tutorials/initial-server-setup-with-ubuntu-16-04#step-five-%E2%80%94-disable-password-authentication-(recommended).

[28] DigitalOcean. How to set up an openvpn server on ubuntu 16.04. URL: https://www.digitalocean.com/community/tutorials/how-to-set-up-an-openvpn-server-on-ubuntu-16-04.

[29] Wireshark Foundation. Wireshark · documentation. URL: https://www.wireshark.org/.

[30] Sergi Hernandez Juan. Multi-master ros systems, 2015.

[31] Ecole Polytechnique Federale de Lausanne. Setting up vpn between ros machines. URL: https://wiki.epfl.ch/roscontrol/vpn-setup.

[32] Samplecaptures - the wireshark wiki. openvpn protocol (openvpn). URL: https://wiki.wireshark.org/OpenVPN.

[33] OpenVPN Inc. What is openvpn. how to. URL: https://openvpn.net/index.php/open-source/documentation/howto.html.

[34] Standard C++ Foundation. Standard c++. isocpp.org. URL: https://isocpp.org/wiki/faq/mixing-c-and-cpp.

[35] Marco Magliona. Selegram. gitlab. URL: https://gitlab.com/marco.magliona/SDP/tree/master.