Cybersecurity in Autonomous Systems: Evaluating the performance of hardening ROS

Francisco Javier Rodríguez Lera, Jesús Balsa, Fernando Casado, Camino Fernández, Francisco Martín Rico, and Vicente Matellán

***Abstract*—As robotic systems spread, cybersecurity emerges as major concern. Currently most research autonomous systems are built using the ROS framework, along with other commercial software. ROS is a distributed framework where nodes publish information that other nodes consume. This model simplifies data communication but poses a major threat because a malicious process could easily interfere the communications, read private messages or even supersede nodes. In this paper we propose that ROS communications should be encrypted. We also measure how encryption affects its performance. We have used 3DES cyphering algorithm and we have evaluated the performance of the system, both from the computing and the communications point of view. Preliminary results show that symmetric ciphers using private keys impose significant delays.**

***Index Terms*—Autonomous systems, Cybersecurity, Data security, Cyber-physical systems, ROS, Performance analysis**

I. Introduction

AUTONOMOUS systems are spreading not just in the virtual world (Internet, software systems), or in science-fiction movies, but in our ordinary real world. It is possible to find driverless cars in the streets, autonomous vacuum cleaners in our homes, autonomous robotic guides at museums, etc. These robotic systems, as any computer-based system, can suffer different types of cyber-attacks, and some degree of cybersecurity [6] is required.

Our research group is developing an assistant robot [4] for the elderly. When we initiated experiments involving potential users, caregivers asked us about the security of our robot and about the privacy of its communications [1]. When an assistant robot carrying a camera is deployed in a home, the access to this camera should be secured; even more when the robot is managing medical information.

We have developed all the software that controls the autonomous behavior of our robot using ROS (Robotic Operating System) [7] which has become the most popular framework for developing robotic applications. It started as a research framework, but currently most of manufacturers of commercial platforms use ROS as the *de facto* standard for building robotic software. For example, object-manipulation robots like Baxter (by Rethink robotics) [2] or service robots as our RB1 (by Robotnik) are ROS based platforms.

Francisco Javier Rodríguez Lera, Jesús Balsa, Fernando Casado, Camino Fernández, and Vicente Matellán are with the Robotics Group (http://robotica.unileon.es) and the Research Institute on Applied Sciences to Cybersecurity (http://riasc.unileon.es) at Universidad de León (Spain).

Francisco Martín Rico is with Robotics Lab, Signal Theory, Communications, Telematic Systems and Computation Department at Universidad Rey Juan Carlos, Madrid (Spain).

Corresponding author: vicente.matellan@unileon.es

A. Security assessment

There are three basic vulnerabilities threatening any computer system: availability (data interruption), confidentiality (data interception) and integrity (data modification). Other authors also add two more [9]: authenticity and non-repudiation (data fabrication from a non-trusted origin). These concepts can be easily translated to industrial control applications [3] or to robotic environments, due to the distributed approach used in most used robotic frameworks (Yarp, ROS, ROBOCOMP).

Let's illustrate this problems considering a robot deployed in a home environment. This robot provides information to users and carries out its behaviors in order to fulfill required tasks. An attacker could attempt to make this robot or its network resources unavailable, this is a denial-of-service attack. The attacker could also intercept and modify command messages in order to change robot behavior, and capture robot information about the environment and about the users. Finally the attacker could simulate a sensor generating new data and sending it to the robot. Fig. 1 summarizes graphically these vulnerabilities in a robotic environment.

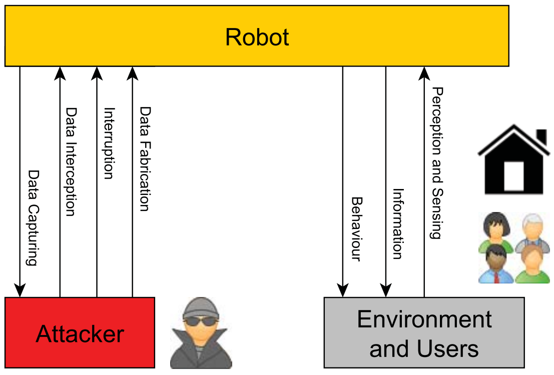


Fig. 1. Conceptual model of the security attacks.

In this research we focus on the confidentiality problem of the data sent to and from the robot. Our proposal consists in encrypt data transmitted between ROS processes, adding two nodes for encryption and decryption task. We don't change ROS messages nor ROS standard functions to send the data. The question is if this hardening of ROS will impact on its performance.

The reminder of this paper is organized as follows: Next section describes the ROS framework communication process.

Section III defines the testbed designed to measure the performance of the encrypted ROS system. Section IV shows the data obtained in the proposed experiments as well as the discussion. Finally section VI presents the conclusions obtained and further work.

II. ROS OVERVIEW

ROS provides specific libraries for robotics similar to classical operating system services such as hardware abstraction (for sensors and actuators), low-level device control, and inter-process communication. Inter-process communication is organized as a graph architecture where computation takes place in ROS processes named nodes. These nodes can receive and send messages. Unfortunately no security was considered in the communication mechanism.

ROS framework is basically a message-passing distributed system. Its architecture is based on processes that publish *messages* to *topics*. For instance, a process (*node*) can be in charge of accessing a sensor, performing the information processing, and publishing it as an information structure on a named topic. Another process can *subscribe* to this topic, that is, read its information. Then the process can make a decision about the movement of the robot. Next, this node will publish the commands in another topic to send them to the motors. ROS nodes can be running in the same computer or in different computers.

Usual ROS configuration is composed by at least one ROS Master and some clients. ROS Master is the key element in the ROS system. It runs as a nameservice and manages registration information about all topics and services used by ROS nodes.

A node communicates with the Master to register its information. Then, the node gets information about other registered nodes, to be able to establish new connections with their topics appropriately. The Master is updated in real time by nodes registering information and topics they publish/subscribe. Fig. 2 summarized the six steps involved in this process as presented in [8].

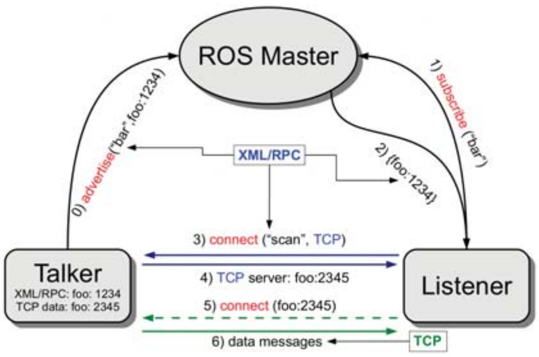


Fig. 2. Conceptual model of ROS topics presented by Radu Rusu in his tutorial[8].

The ROS distributed approach is very convenient for developers but can be easily tampered by malicious hackers. For instance, in [5] an experiment involving a ROS-based honeypot is described. The honeypot was a radio model truck with two cameras and a compass as sensors, and it was controlled from a remote ROS node written in Javascript and hosted in a remote enterprise grade web server. Vulnerabilities described in the paper comprise plain-text communications, unprotected TCP ports and unencrypted data storage.

The first step to solve some of these problems is to secure the communication channels by using an encryption mechanism. But how does encryption impact on the performance of a robotic system? This is the goal of this paper, characterize and evaluate different alternatives to secure ROS communication system and measure their performance.

III. TESTBED DESCRIPTION

In order to evaluate the performance of the encrypted version of ROS communications environment, we designed the following scenario. Fig. 3 provides a graphical representation of the scenario created.

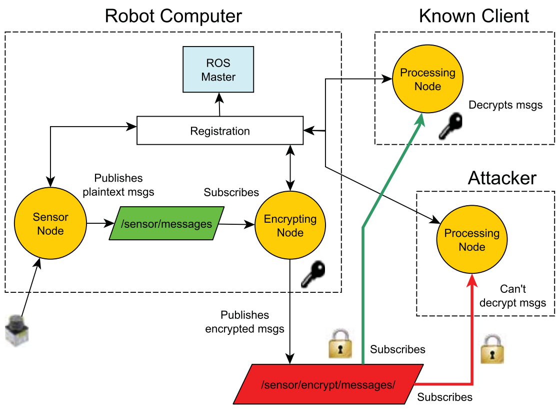


Fig. 3. Scheme of the scenario used for testbed.

First, we installed ROS Indigo in the on-board computer of the our RB1 mobile robot. The ROS master component ran in this platform. The robot was connected by wire to our laboratory network infrastructure for this experiments. The robot computer has two nodes: one node connected to a sensor that publishes the data into */sensor/messages* topic; and one node connected to this topic that performs data encryption and publishes them into a */sensor/encrypt/messages* topic.

Second, we used one desktop computer as a "known client", also with ROS Indigo installed and connected by cable to our network. This client knows the master ROS IP, so it can communicate with master. We run a decryption node in the client computer, which registers to master and subscribes to the topic */sensor/encrypt/messages*. This node decrypts data and prints them on the screen.

Third, we used another desktop computer as a simulated "attacker", connected to the same cable network. This computer has the same ROS version running on it. The attacker doesn't know the master ROS IP, but he can easily discover it performing a network scan with well known tools like *nmap*. Then, the attacker could execute a malicious node for attempting to read laser data, which is being published in the topic */sensor/encrypt/messages*. Despite the node could subscribe to that topic, all data received is encrypted. As a result, the malicious node can't see original laser messages because the attacker doesn't have the key to decrypt them.

The cryptographic key is stored in the master node and it is known by the legitimate clients. In a system in production this mechanism can be implemented as a public-private key schema as RSA to safely share the key between the nodes.

A. *Encrypting ROS messages*

In this first approach we have changed the data published by the node, and instead of publishing the information in a plain manner, we publish the information encrypted.

ROS uses a *messages description language*[1] for describing the data values that each node publish. In this manner, it is easy for ROS tools to automatically generate source code for the message type in several target languages as ICE or IDL specification language.

There are built-in types (14), array types (4) and customized types. For instance, if we get the *sensor_msgs/Image.msg* message, we find a composed message by one non built-int message, and five built-in (3 x uint32, 1 x string, 1 x uint8)and one array type (uint8[]).

```
std_msgs/Header header #custom msg

uint32 height
uint32 width
string encoding
uint8 is_bigendian
uint32 step

uint8[] data #matrix data of an image
```

In this case is *data* field the element that contains the information of the image grabbed by the camera, so we are going to encrypt this element before to be publisher in ROS distributed system.

We use the 3DES algorithm to provide a security layer to ROS data. It is a known that 3DES cyphering speed is slower [10] than other cipher methods as for instance AES. We use this algorithm as the worse environment to analyze its behavior in a real environment with ROS. Triple DES (3DES) references the Triple Data Encryption Algorithm (TDEA or Triple DEA). It is a symmetric-key block cipher that applies the Data Encryption Standard (DES) algorithm three times to each data block. It is standardized by NIST in the Recommendation for the Triple Data Encryption Algorithm Block Cipher (NIST Special Publication 800-67).

3DES algorithm provides three keys that are 128 (option 1) or 192 (option 2) bits long. In option 1, the key is split into K1 and K2, whereas $K1 = K3$. The option 2 is a bundle of three 64 bit independent subkeys: K1, K2, and K3. To highlight that the three keys should be different, otherwise 3DES would degrade to single DES.

The data block of the algorithm has a fixed size of 8 bytes where 1 out of 8 bits is used for redundancy and do not contribute to security. In that manner, the effective key length is respectively 112 in option 1 and 168 bits in option 2.

The algorithm presents the next behavior, the plain text is encrypted three times: first it is encrypted with K1, then decrypted with K2, and finally encrypted again with K3. The ciphered text is decrypted in the reverse manner.

3DES is cryptographically secure, although it is slower than AES algorithm. In a next stage, we will substitute 3DES with AES in our encryption system and repeat all the test we have done, to compare the performance.

From the development side, it was used the PyCrypto package. It is an extended python Cryptography Toolkit that allows to simply the method to encrypt or decrypt in multiple encryption algorithms.

IV. Experimental Measurements

To illustrate the described approach, we present an ad-hoc implementation of an encryption system to change part of the message used for transmit or receive robot sensors information.

We have added a function to our program in order to measure the time spent on each encryption and decryption call. The function is a python method presented as a decorator pattern, which is used here to extend the functionality of encryption/decryption at run-time.

```
def fn_timer(function):
          @wraps(function)
          def function_timer(*args, **kwargs):
              v_time_0 = time.time()
              result = function(*args, **kwargs)
              v_time_1 = time.time()
              return result
          return function_timer
```

We have divided the experiments in three parts. First we have analyzed how the "Publisher-listener" ROS nodes work under encrypted conditions. Then we have stressed the same nodes publishing bigger text messages, in plain text as well as encrypted text. Finally we have analyzed a camera sensor, under the same encryption/decryption conditions.

A. *Hardware/Software Set-up*

We want to evaluate how the encryption of communications would affect the performance of ROS. We have used the better of the cases where a 8x Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz with 16231MB of RAM Memory and running an Ubuntu 14.04.4 LTS Operating System. The ROS master system has 234 process running by default, the client is running 242 process.

The two computers are connected by an Alcatel-Lucent OmniSwitch 6860E switch This hardware conditions are the most favorable against the real environment of a robotic platform for instance wireless or hardware restrictions.

B. *Test 1: HelloWorld Publisher-Listener Node*

In this test we have used the version of talker/listener tutorial proposed by ROS[2]

[1]http://wiki.ros.org/msg

[2]http://wiki.ros.org/rospy_tutorials/Tutorials/WritingPublisherSubscriber

This package distributed by ROS as a demo, presents a simple ROS package that creates two rospy nodes. The ”talker” node broadcasts a *Hellow world + Timestamp* message on topic ”chatter”, while the ”listener” node receives and prints the message.

TABLE I
TIME IN SECONDS OF CPU SPENT ON A SIMPLE PUBLISHER/LISTENER RUNNING TEST.

	Plain Publication	Plain Subscription	Encrypt Publication	Decrypt Subscription
Time running	34.491	34.484	36.882	34.995
Time user	0.184	0.196	0.348	0.24
Time sys	0.024	0.08	0.064	0.056
Total CPU	0.208	0.276	0.412	0.296

Table I presents the CPU time used when the nodes are running in different machines. The values are the result of launching ROS nodes using the Unix command *time*. The values present a minimal consumption of CPU associated to the process. In case of plain publish/subscribe it represents than 1% and in the case of the publisher in encrypt mode it represents approximately the 1.1%, that is almost the same than in plain mode.

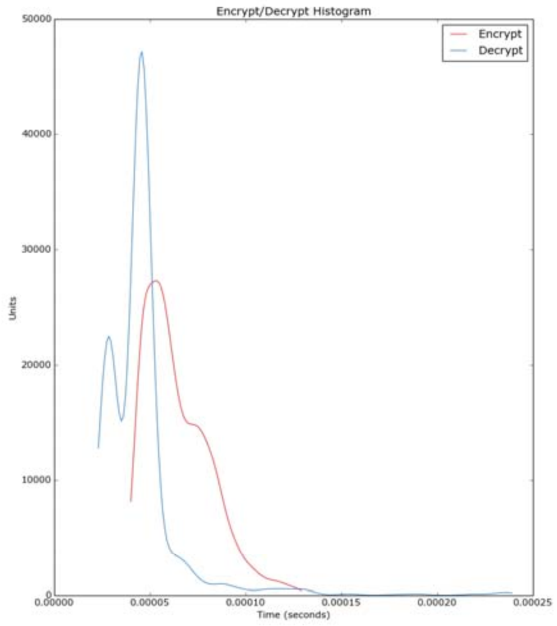


Fig. 4. Histogram showing the time spent on a encrypt/decrypt DES3 function call.

Figure 4 presents the histogram associated to the calls to encrypt/decrypt method used in these tests. We can observe that the average time in the encryption process is slightly higher (0.000065 seconds) than the decryption process (0.000045 second). However, in both cases it is almost negligible. The worst case presented a 0.000129 seconds in ciphering time and 0.000239 second of deciphering time.

Figure 5 presents the screenshot of this experiment. It shows the three terminals from the two nodes involved in this experiment. Publisher node (top-left window) that presents a simple log, subscriber node with key (bottom-left) that presents the message after the decryption call and subscriber node without key (*rostopic echo*) that shows the encrypted message in the topic.

C. Test 2: Custom Text Publisher-Listener Node

In this test we have used the same version of talker/listener nodes proposed by ROS. In this particular test we generate a synthetic text strings with different sizes:

- T1: this is a string message of 262144 bytes (256 KB)
- T2: it is a string message of 524288 bytes (512 KB)
- T3: presents a string message of 1048576 bytes (1024 KB)

We want to determine the duration of execution of our talker and the time spent by our listener nodes. Again, we run the Linux *time* command to measure the total CPU time consumed by the ROS talker process.

Initially we run the test with the three size types of messages using plain text. We have performed the test three times, running the nodes for a time lapse of 30 to 35 seconds. Firstly, reviewing the T1 type (it is presented in a time window of 32.884 seconds) we find a CPU time of 1.408 seconds. It was a user time of 1.364 seconds and a sys time of 0.044s. I Secondly, we analyse the string of type T2. It was launched in a window of 33.749 seconds, and needed a total CPU of a 2.672 seconds (user time of 2.508s and a sys time of 0.164s). Finally, we analyse T3, that presents in a time window of 34.731 seconds a total CPU of 5.260 seconds (it is divided by a user time of 5.140s and a sys time of 0.120s).

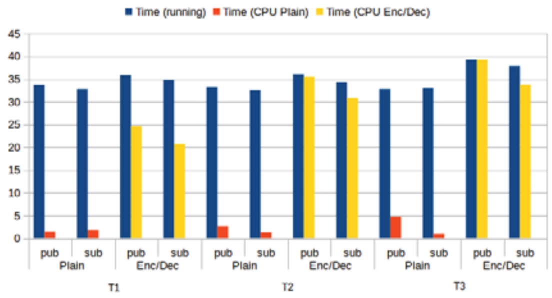
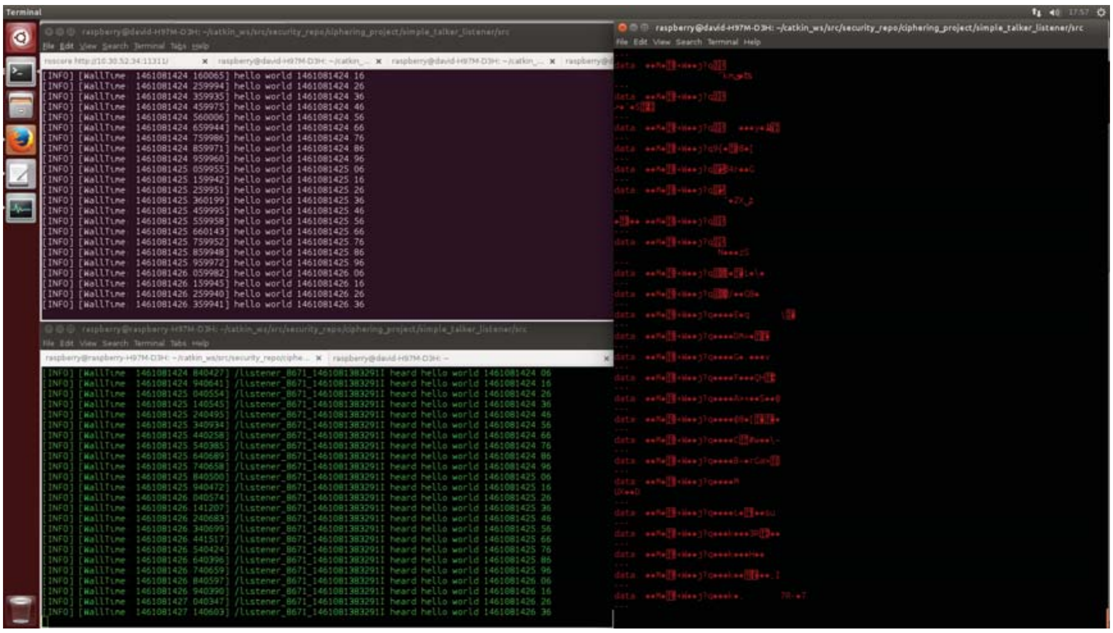


Fig. 6. Time of CPU spent by publiser/subscriber nodes sending T1, T2 and T3 messages in plain and Encrypted/Decrypted on test 2.

This is totally different when we are working with encrypted text messages. Fig 6 presents the main differences. We have repeated the same experiment, but this time calling an encryption method that encrypts from plain text to 3DES. It is possible to see that the encrypt process consumes more CPU than the plain process. For instance, T1 type presents running for 34.486 seconds a total of 23.008 of CPU. This means that the total CPU time increases almost 62%. This is even higher in T2 and T3 types with almost the 98% of real execution time.



```
[INFO] [WallTime: 1461081424.160065] hello world 1461081424.16
[INFO] [WallTime: 1461081424.259994] hello world 1461081424.26
[INFO] [WallTime: 1461081424.359935] hello world 1461081424.36
[INFO] [WallTime: 1461081424.459975] hello world 1461081424.46
[INFO] [WallTime: 1461081424.560006] hello world 1461081424.56
[INFO] [WallTime: 1461081424.659944] hello world 1461081424.66
[INFO] [WallTime: 1461081424.759986] hello world 1461081424.76
[INFO] [WallTime: 1461081424.859971] hello world 1461081424.86
[INFO] [WallTime: 1461081424.959960] hello world 1461081424.96
[INFO] [WallTime: 1461081425.059955] hello world 1461081425.06
[INFO] [WallTime: 1461081425.159942] hello world 1461081425.16
[INFO] [WallTime: 1461081425.259951] hello world 1461081425.26
[INFO] [WallTime: 1461081425.360199] hello world 1461081425.36
[INFO] [WallTime: 1461081425.459995] hello world 1461081425.46
[INFO] [WallTime: 1461081425.559958] hello world 1461081425.56
[INFO] [WallTime: 1461081425.660143] hello world 1461081425.66
[INFO] [WallTime: 1461081425.759952] hello world 1461081425.76
[INFO] [WallTime: 1461081425.859948] hello world 1461081425.86
[INFO] [WallTime: 1461081425.959972] hello world 1461081425.96
[INFO] [WallTime: 1461081426.059982] hello world 1461081426.06
[INFO] [WallTime: 1461081426.159945] hello world 1461081426.16
[INFO] [WallTime: 1461081426.259940] hello world 1461081426.26
[INFO] [WallTime: 1461081426.359941] hello world 1461081426.36
```

```
[INFO] [WallTime: 1461081424.840427] /listener_8671_1461081383291I heard hello world 1461081424.06
[INFO] [WallTime: 1461081424.940641] /listener_8671_1461081383291I heard hello world 1461081424.16
[INFO] [WallTime: 1461081425.040554] /listener_8671_1461081383291I heard hello world 1461081424.26
[INFO] [WallTime: 1461081425.140545] /listener_8671_1461081383291I heard hello world 1461081424.36
[INFO] [WallTime: 1461081425.240495] /listener_8671_1461081383291I heard hello world 1461081424.46
[INFO] [WallTime: 1461081425.340934] /listener_8671_1461081383291I heard hello world 1461081424.56
[INFO] [WallTime: 1461081425.440258] /listener_8671_1461081383291I heard hello world 1461081424.66
[INFO] [WallTime: 1461081425.540385] /listener_8671_1461081383291I heard hello world 1461081424.76
[INFO] [WallTime: 1461081425.640689] /listener_8671_1461081383291I heard hello world 1461081424.86
[INFO] [WallTime: 1461081425.740658] /listener_8671_1461081383291I heard hello world 1461081424.96
[INFO] [WallTime: 1461081425.840500] /listener_8671_1461081383291I heard hello world 1461081425.06
[INFO] [WallTime: 1461081425.940472] /listener_8671_1461081383291I heard hello world 1461081425.16
[INFO] [WallTime: 1461081426.040574] /listener_8671_1461081383291I heard hello world 1461081425.26
[INFO] [WallTime: 1461081426.141207] /listener_8671_1461081383291I heard hello world 1461081425.36
[INFO] [WallTime: 1461081426.240683] /listener_8671_1461081383291I heard hello world 1461081425.46
[INFO] [WallTime: 1461081426.340699] /listener_8671_1461081383291I heard hello world 1461081425.56
[INFO] [WallTime: 1461081426.441517] /listener_8671_1461081383291I heard hello world 1461081425.66
[INFO] [WallTime: 1461081426.540424] /listener_8671_1461081383291I heard hello world 1461081425.76
[INFO] [WallTime: 1461081426.640396] /listener_8671_1461081383291I heard hello world 1461081425.86
[INFO] [WallTime: 1461081426.740659] /listener_8671_1461081383291I heard hello world 1461081425.96
[INFO] [WallTime: 1461081426.840597] /listener_8671_1461081383291I heard hello world 1461081426.06
[INFO] [WallTime: 1461081426.940390] /listener_8671_1461081383291I heard hello world 1461081426.16
[INFO] [WallTime: 1461081427.040347] /listener_8671_1461081383291I heard hello world 1461081426.26
[INFO] [WallTime: 1461081427.140603] /listener_8671_1461081383291I heard hello world 1461081426.36
```

Fig. 5. Screen-shot with the nodes involved in the simple publisher/subscribe test. Upper left terminal presents the publisher node before encryption. Bottom left terminal depicts the subscriber node after decryption. Right terminal presents the results of rostopic echo on */chatter* topic used for transmitting info through nodes.

It is clear that the encrypted/decrypted approach consumes more CPU (yellow bars in Fig. 6) than plain text (orange bars in Fig. 6).

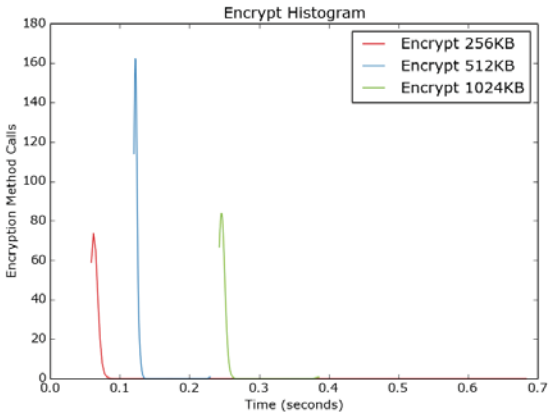


Fig. 7. Histogram by each publisher node attending each call to the encryption method.

We know that the encryption process needs more CPU, but how long does the encryption/decryption takes in each iteration?. Just for clarification, we review the encryption phase in the publisher node. In case of T1 messages, during its window (34.486s) the node is able to encrypt 1040 messages. The minimal time to perform this task is 0.059246 seconds and the maximum time to encrypt the string is 0.683408 seconds. In average, the system spends 0.063858 seconds (standard deviation of 0.000388s). This time increases with T2 and T3 types. T2 type needs 0.123383 seconds in average with standard deviation of 0.000040s and T3 0.247303s with a standard deviation of 0.000137s. Fig. 7 presents the histogram associated to this experiment.

D. Test 3: Camera Node

The third experiment has been developed using a RGB camera. A Logitech, Inc. QuickCam Pro 9000 webcam providing 15 frames per second (fps) using a 640x480 pixels resolution. This sensor was registered on ROS system to deliver images to the system.

This test involved three ROS nodes, two publishers: usb_cam, encrypted_node and one listener: decrypted_node. First ROS node,*usb_cam*[3] was based in the code contributed by Benjamin Pitzer and still maintained by ROS developers. It is in charge of recording information from the camera and publishing in the *usb_cam* topic.

The second node runs in the same computer and it is in charge of encrypting the message published by *usb_cam*. The third node runs on a different computer and it is in charge of decrypting the messages got from the publisher and of displaying the received image on the display.

Figure 8 graphically represents this set up for a given moment T at the listener node. We used the ROS node

[3]http://wiki.ros.org/usb_cam

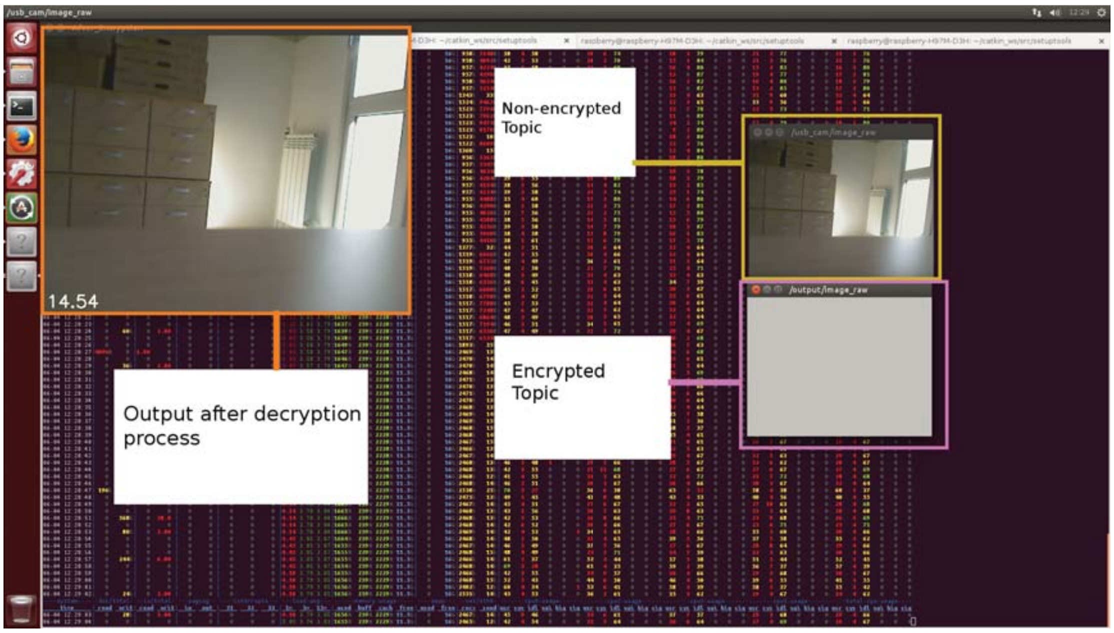


Fig. 8. Screenshot taken during image encryption test.

image_view to visualize images in both topics: the non-encrypted one, that to validate delays, and the encrypted one (that cannot be visualized) and the frame that shows the image after decryption.

We measured the time needed to encrypt and decrypt the ROS message (*sensor_msgs/Image.msg*). We just encrypted and decrypted the field data, defined as `uint8[]` and processed as a text string. Our nodes shows this information during the execution in a visualization frame that we used to measure the frame rate.

We ran this experiment for 18 hours sending in total 971.790 frames. The average time in the process of encrypt the sensor data was 0.010948 seconds (stddev 0.000004s). Minimal processing time was 0.001309 seconds and max processing time was 0.026909 seconds. The average time to decrypt the images was 0.008828 seconds (stddev 0.000003s). The max time decrypting an image was 0.039130s and the minimal 0.001288s.

V. Discussion

This study provides an in-depth characterization of the use of 3DES encryption in ROS communications. First, we have configured two ROS nodes equipped with state of the art computer power capabilities. The only software running on both machines in the first experiment was limited to the one needed for the test, no extra computational process were running during the experiments.

Results presented in this test showed that there is no difference in the performance between sending a plain string of chars *Hello world* + *timestamps* and an encrypted string of chars. However, the experiments showed that CPU time used by the encrypting of messages has a geometric progression with the size of the message, which means that the performance of the global system is reduced.

ROS has a logging system that measures the publish/subscribe performance among connected nodes. When the system is active on a node, the statistical data is published in a specific topic that is updated at 1Hz.Using this system we have observed two main issues:

- Traffic in bytes grows lineally with the number of messages:
  - plain mode delivers 10 messages of 346 bytes (average) of traffic.
  - encrypted mode delivers 10 messages per second whose size is 420 bytes in average.
- Performance decrease quadratically with the size of the messages:
  1) plain mode delivers 10 messages per second of 11.010.132 bytes of traffic (in average).
  2) encrypted mode only delivers 5 messages per second in average of 5.242.880 bytes of traffic (in average).

The performance reduction is due to the time consumed by the encryption and decryption process in each message: Publisher uses 0.247303 seconds per message to encrypt every message. Listener needs 0.240323 seconds per message just for decrypting. This means that the publisher process needs more time to produce messages and the listener more time to consume it.

In the same manner, if the publisher needs to process this

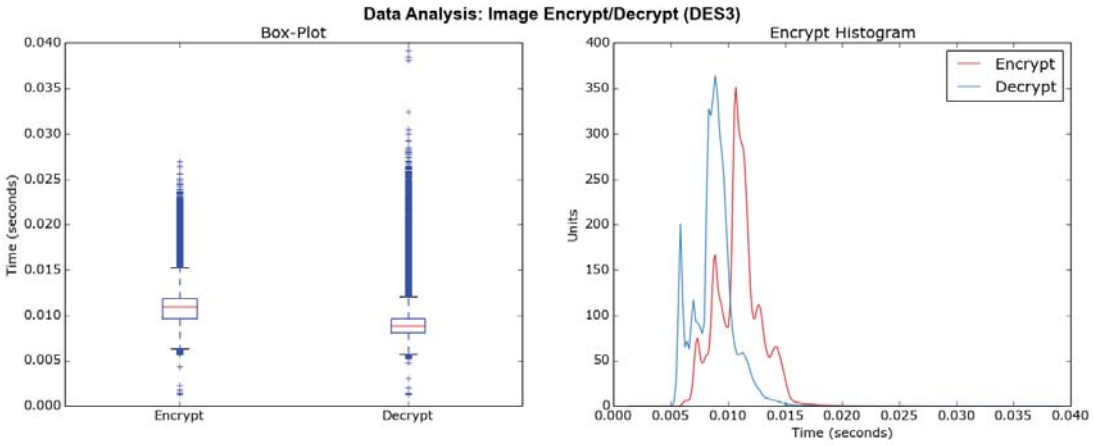


Fig. 9. Time spent on each call to encryption/decryption function during the image processing in test 3.

information in any way, for instance showing the message in the screen, the overall performance producer/consumer is reduced.

Finally, the experimental evaluation made in test 3 does not show a significant delay due to the encryption process. For instance, we got 14.56 frames per second in the publisher, and 14.54 frames per second in the listener.

Fig. 9 shows the performance of the publisher cyphering data. Left part of the figure summarizes the time used for encrypting and decrypting the three types of messages. The right part is the histogram of times used for calls to the encryption method. We observe that processing times in average are similar, the encryption process is slightly higher (2ms) than in the decryption process.

VI. Conclusion and Further Work

We have shown that using ciphered communications avoids security problems related with the plain-text publish/subscribe paradigm used by ROS. However, the overhead of CPU performance and communication load should also be considered in distributed architectures that need to work on real time.

This article presents a performance analysis of the use of encrypted communications in a ROS system. We have evaluated the impact of this added feature from two points of view: CPU consuming and network traffic.

As we pointed out in the introduction, securing communications is just one dimension in the cybersecurity of autonomous systems. If we want to see these machines working in our homes we need to secure navigation abilities and interaction mechanisms, to avoid manipulated or malicious behaviors and make robots reliable assistants, in particular if we want to install mobile robots in private spaces as homes.

As a further work we would like to test Real-Time Publish Subscribe (RTPS) protocol used by the ROS2 communications design. In that manner we plan to evaluate in a quantitative way the network and CPU performance under similar conditions as well as to compare qualitatively the pros and cons of the DDS based solution proposed in the new ROS design[4].

Acknowledgment

The authors would like to thank the Spanish Ministry of Economy and Competitiveness for the partial support to this work under grant DPI2013-40534-R and to the Spanish National Institue of CyberSecurity (INCIBE) under grand Adenda21 ULE-INCIBE.

References

[1] Tamara Denning, Cynthia Matuszek, Karl Koscher, Joshua R. Smith, and Tadayoshi Kohno. A Spotlight on Security and Privacy Risks with Future Household Robots: Attacks and Lessons. In *Proceedings of the 11th International Conference on Ubiquitous Computing*, 2009.

[2] Conor Fitzgerald. Developing Baxter. In *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference*, pages 1–6. IEEE, 2013.

[3] Peter Huitsing, Rodrigo Chandia, Mauricio Papa, and Sujeet Shenoi. Attack taxonomies for the Modbus protocols. *International Journal of Critical Infrastructure Protection*, 1:37–44, 2008.

[4] Francisco Martín, José Mateos, Francisco Javier Lera, Pablo Bustos, and Vicente Matellán. A robotic platform for domestic applications. In *XV Workshop of Physical Agents*, 2014.

[5] Jarrod McClean, Christopher Stull, Charles Farrar, and David Mascareñas. A preliminary cyber-physical security assessment of the Robot Operating System (ROS). *SPIE Defense, Security, and Sensing*, 8741:874110, 2013.

[6] Santiago Morante, Juan G Victores, and Carlos Balaguer. Cryptobotics: Why robots need cyber safety. *Frontiers in Robotics and AI*, 2(23):1–4, 2015.

[7] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.

[8] Radu Bogdan Rusu. Ros-robot operating system. *Tutorial Slides, November*, 1, 2010.

[9] Y. Sattarova Feruza and Tao-hoon Kim. IT security review: Privacy, protection, access control, assurance and system security. *International Journal of Multimedia and Ubiquitous Engineering*, 2(2):17–31, 2007.

[10] Gurpreet Singh. A study of encryption algorithms (rsa, des, 3des and aes) for information security. *International Journal of Computer Applications*, 67(19), 2013.

[4]http://design.ros2.org/articles/ros_on_dds.html