Instituto Tecnológico y de Estudios Superiores de Monterrey

“Campus Querétaro"

Increasing security in ROS using AES and CUDA

Programming languages final project

Author:

A01328867

Daniel Jerónimo Gómez Antonio

Professor:

Benjamín Valdés Aguirre

November 20, 2018

Contents

1 Context of the problem **4**

1.1 The Robot Operating System 4

1.2 ROS and the lack of security 6

2 Solution **6**

2.1 The Advanced Encryption Standard and ROS 8

2.2 Use of CUDA to parallelize the AES 10

2.3 Implementation of the AES using CUDA 11

2.4 Integration of the AES-CUDA with ROS 14

3 Results **15**

3.1 Test cases using static files of different sizes 15

3.2 Test cases using ROS messages 17

4 Conclusions **18**

5 Setup instructions **19**

5.1 AES and CUDA 19

5.2 AES-CUDA and ROS 19

List of Figures

1 ROS general functionality diagram 5
2 Comparison of the CPU consumption for 3 encryption algorithms 7
3 Summary of the encryption algorithms analyzed by Rodríguez-Lera et al. . . 8
4 Cipher and decipher AES flow diagram 9
5 CPU vs GPU architecture 11
6 Runtime obtained for the cipher/decipher processes using sequential(a) and parallel(b) AES. Time is reported in ms 16
7 Logarithmic comparison between the runtime obtained for sequential and parallel AES 17
8 Nodes running the AES cipher and decipher in ROS using CUDA 18

List of Tables

1 Technical specs of the computer used to perform the testing 15
2 Runtime measured for sequential and parallel AES when ciphering and deciphering several files 16

Abstract

Nowadays, robots development has increased substantially due to the emerging fields where the robotics can be applied. There are several tools, frameworks and systems that makes this process easier than before. Unfortunately, the security of the software, used to manage the system of a robot and the collected data, is not as trusted as it should be. The main reason to be worried about the security of the information managed by a robot is that this type of data is very sensitive, i.e. images and location data from an indoor environment, like a house. This project focuses on the use of the Robot Operating System, which is a framework used to develop robot software. ROS uses the producer/consumer paradigm to communicate, within the same LAN, the different subsystems that integrate the robot. The problem is that the data sent by each node[1] is just plain text. That means that anybody connected to the same network could see the data. This project presents an alternative to solve this problem using the Advanced Encryption Standard (AES), to cipher the data. Moreover, the implementation of the algorithm is done using CUDA, so it runs in parallel. This project provides two advantages to solve the problem of the security in robots: first, the data is sent encrypted, so unauthorized users are not able to see it; second, the cipher/decipher processes run both in parallel to reduce the latency that could be generated.

[1]In ROS, any executable file is called node and each of them can communicate with others

1 Context of the problem

As the programmable computers emerged some time ago as machines that changed our lives, now the robots are taking more importance in our daily activities, from industrial robots until personal-assistance robots. According to Larry Hardesty, in the MIT News bulletin, scientists and researchers have focused on making more capable robots by developing science and autonomy. However, there is no enough effort to solve the issues related to cybersecurity and privacy. [3] Nevertheless, the robots are exposed to the same security issues that the computers systems or even more. The problem is that the robots manage sensitive information that they acquire from their environment. That is the reason why other people could try to use this information with bad intentions or even take control of the robot and perform malicious activities.

When a robot is deployed in a public space, or a very private space, the data that the sensors collect includes photos, audio and different information that raises a concern about how the information should be processed in order to keep it safe and away from risks, like identity theft [6]. It is evident that the data must be protected in a way that guarantees that the use of the robot is not going to cause a negative impact on society.

The security in the robot can be achieved in different ways, one of these is the use of a framework that manages the data in a safe way. Using this approach, the robot developers can dedicate more valuable time to the final application than the adding of security protocols [6]. That is why, in this project it is presented an alternative to add a security feature to a framework used to develop robot software called ROS.

1.1 The Robot Operating System

The Robot Operating System (ROS) is an open-source framework for writing robot software that provides an easy way to intercommunicate the subsystems of a robot [7]. It has gained popularity as the main framework for robotic software development because it simplifies the task of creating complex and robust robot behavior. ROS also encourages collaborative robotic projects. One of the advantages of ROS is that it allows communication between systems even if they were written in C/C++ or Python, it also provides several packages that can be incorporated to integrate the use of popular software, like OpenCV and Matlab. ROS can run in any device with Ubuntu installed (i.e. on a PC, Raspberry, Jetson, etc).

ROS uses the producer/consumer paradigm to communicate the different systems that integrates the robot. Actually, in ROS this is called publisher/subscriber. The executable files in ROS are called nodes. Thus, the general functionality of ROS is to provide to the nodes a way to communicate between each other to share data. By doing this, all the sensors and actuators can cooperate and help the control units to take decisions even is they are running in different hardware. The only condition is that all the subsystems need to be connected to the same network(usually a LAN).[7] In addition, it makes the teleoperation of the robot easy to accomplish. The Fig. 1 shows a diagram to illustrate how ROS works, it is modeling three nodes, the first one acquires an image from a camera, the second one processes the data and the other displays the image for the user. This diagram is valid for different robots. For example, it could run on a mobile robot sensing the medium; or in a drone flying and taking pictures from above; or even on a Mars Rover exploring the surface of another planet.

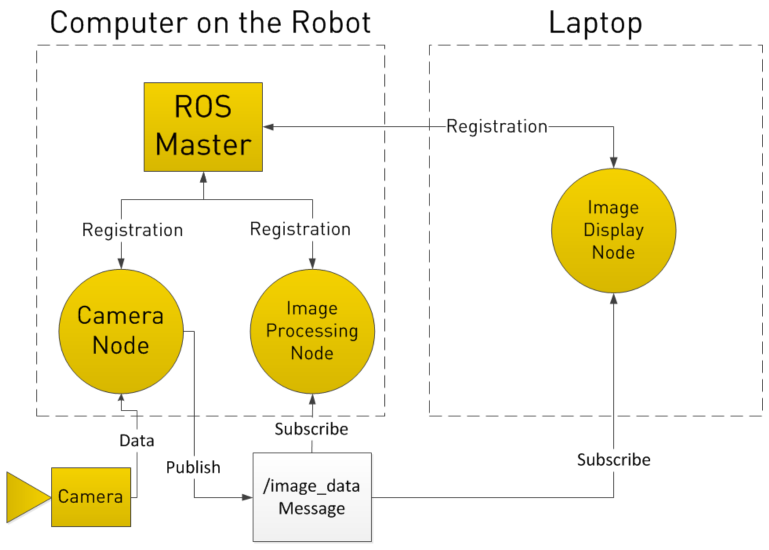


Fig. 1. ROS general functionality diagram

1.2 ROS and the lack of security

The protocol used by ROS to send and receive messages is the TCP/IP. The transport layer is known as TCPROS, it contains standardized headers that help the nodes read different type of messages, like integers, strings, images, speed commands, etc. However, there is no a security protocol that protects the data from unauthorized access. In fact, the data is sent in plain text, so anybody connected to the same network may see the messages or connect to the robot and change its behavior. [6]

In addition, as mentioned by Foote [2] the main security issues regarding the use of the current version of ROS are:

- Plain-text communication
- Unprotected TCP ports
- XML-RPC legacy issues
- Unencrypted data storage

The objective of this project is to propose an alternative to solve the problem of plain-text communications. Currently, there is one project trying to solve the security issues by using Distributed Data Systems instead of TCPROS, it is called ROS2 but it is still in a beta version as showed by Hood and Woodall (cited in [6]).

2 Solution

A possible solution that increases the security level of ROS is the use of encrypted messages. By doing this, the messages are no longer sent as plain-text. Thus, the data can only be seen by nodes that have the correct key used to decipher the messages. This protects the information from unauthorized users.

Based on the analysis presented in the article Message Encryption in Robot Operating System by Rodríguez-Lera et al. [6], the performance of a mobile robot, running

ROS, is just slightly affected when using cipher/decipher processes at both ends of the communication. This fact opens a possibility to implement encryption algorithms and propose optimizations to achieve a good performance on robots. In the Fig. 2 there is a chart showing the CPU consumption obtained in the analysis of 3 encryption algorithms: the AES, 3DES and Blowfish running on 2 different Control Units [6]. As it can be seen, the AES algorithm goes from less than 10% until almost 40% of CPU usage.

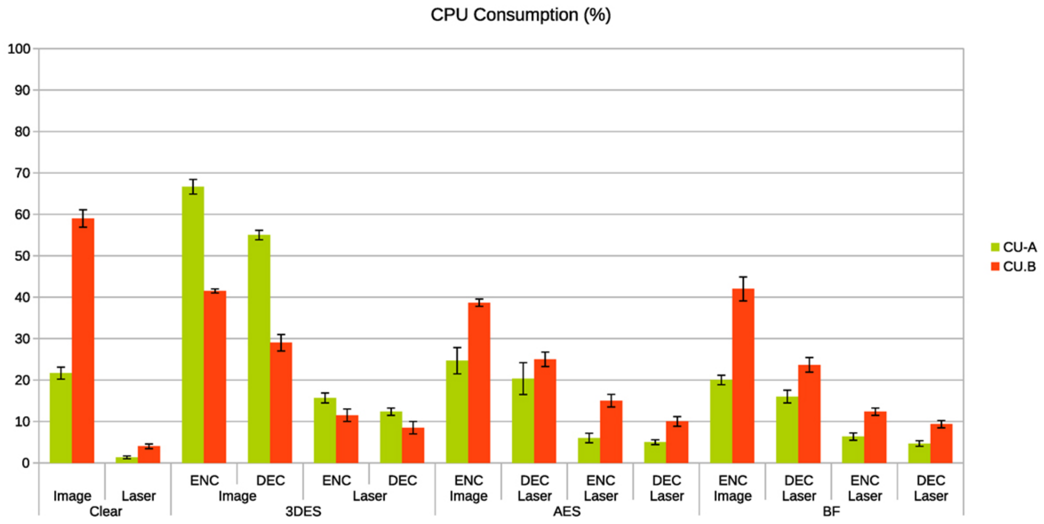


Fig. 2. Comparison of the CPU consumption for 3 encryption algorithms

The results taken from [6] show that it is possible to run encryption algorithms without compromising the final application of the robot, but only if a powerful or medium CU is used. The authors also conclude that using the AES algorithm in the CBC[2] mode, the level of security is superior as shown in Fig. 3. However, the main issue of AES resides on the performance of the robot

[2]Cipher Blocker Chaining, is a form of cipher block encryption that uses previous cipher-text blocks, which adds an extra level of complexity

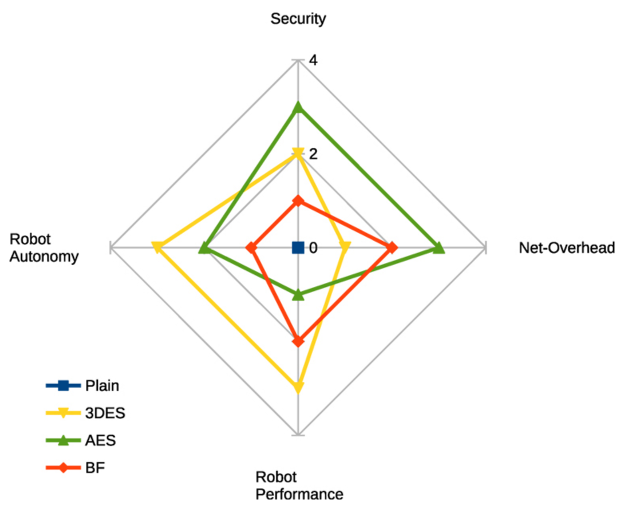


Fig. 3. Summary of the encryption algorithms analyzed by Rodríguez-Lera et al.

That is exactly the motivation for this project: **increasing the performance of a high-capable robot when using the AES algorithm by parallelizing the cipher process with CUDA.**

2.1 The Advanced Encryption Standard and ROS

The AES became the new standard for encryption on November 26, 2001. It is based on the Rijndael algorithm created by Vincent Rijmen and Joan Daemen. It was proposed as a version to substitute the DES, which was cracked in 1998. As stated by Selent the Rijndael algorithm "uses a combination of Exclusive-OR operations (XOR), octet substitution with an S-box, row and column rotations, and a MixColumn. It was successful because it was easy to implement and could run in a reasonable amount of time on a regular computer."[9]

The AES algorithm works in both directions, it is used to cipher and decipher any type of binary file. It receives as input the file divided in blocks of 128 bits (16 bytes) and a key of 128, 196 or 256 bits. This project uses a key of 128 bits. For each block, the output is another block of 128 bits with the data ciphered. The operations are performed over a matrix of 4x4 containing the corresponding block, called *state*.[1] The Fig. 4 shows the graphical representation of the algorithm, remarking the main transformations that take place over the state. It is important to note that these transformations are applied several times using the CBC mode, which means that after the first round of ciphering is done to the plain-text, the further operations will be applied over a cipher-text. By doing this, the security increases. These transformations are described as follows:

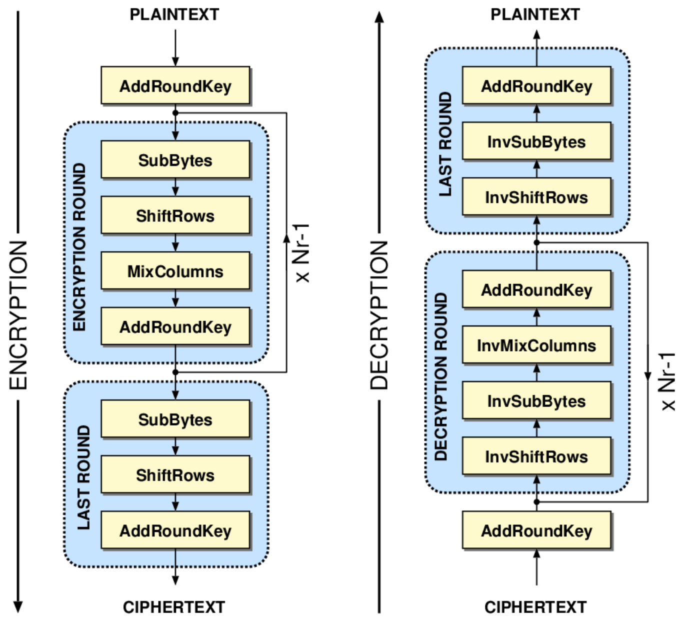


Fig. 4. Cipher and decipher AES flow diagram

- **Key Expansion:** It is a routine used to expand the input key and obtain $16*(Nr+1)$ new bytes. Therefore, a different key is used in each round of the algorithm. The number of rounds depends on the length of the key, since it will be used a key of 16 bytes, then there will be 10 rounds and the expanded key will be 176 bytes. The methods used to expand the key involves cyclic permutations and arithmetic operations over the Galois fields. For more details refer to [1].

- **Add Round Key:** In this part of the process, the key is added to the state using a XOR operation. The portion of the expanded key used depends on the number of round.

- **Sub Bytes/Inverse Sub Bytes:** This transformation uses a lookup table to do a byte by byte substitution, i.e. the value of each byte is changed by another within the S-box matrix or inverse S-box for deciphering. The matrix used was carefully chosen to provide security. Again, the values of the matrices are the result of modular arithmetic over Galois fields. [1]

- **Shift rows/Inverse Shift Rows:** In this step, the last three rows of the state are shifted to the left for ciphering or to the right for deciphering. The row 1 is shifted by an offset of 1, the row 2 by 2, and the row 3 by 3.

- **Mix Columns/ Inverse Mix Columns:** This process is really complex to explain, because it uses Galois fields operations over each column of the state matrix. Fortunately, the arithmetic operations can be precomputed and stored in different lookup tables. Thus, the process is reduced to perform multiplications and XOR operations with each column of the state and the corresponding lookup table.[9]

All of the previous transformations are applied to each block in which the input file is divided, and they are independent between them.

As mentioned by Rodríguez-Lera et al. in their article, the AES algorithm is a good option to integrate within ROS and cipher the plain-text messages. The empirical results show that the level of security is better than using other encryption algorithms (see Fig. 3). However, its weakness is found in the robot performance, which means that due to the time consumed by the algorithm to cipher and decipher data, the robot modifies it behavior. This is caused by ROS messages that are dropped because the buffer got full and the waiting time expired.

2.2 Use of CUDA to parallelize the AES

The aim of this project is to improve the efficiency of the AES algorithm when it is used to encrypt ROS messages. This could be achieved by parallelizing the implementation of the algorithm. This was done using CUDA. Therefore, it is important to mention that only high-capable robots can take advantage of this solution because CUDA needs to run on a dedicated hardware.

According to the NVIDIA website, “CUDA is a revolutionary parallel computing architecture created by NVIDIA. CUDA makes it possible to use the many computing cores in a graphics processor to perform general-purpose mathematical calculations, achieving dramatic speedups in computing performance." [5] As mentioned, CUDA needs a graphics processor unit (GPU) to run. Because of the architecture of the GPUs, it is possible to manage hundreds of threads and get huge speedups. This is one of the main reasons CUDA was chosen for this project. In Fig. 5 it can be observed the general differences between the CPU and GPU architecture. The GPU has a big number of cores that are grouped into blocks and managed by its own control unit. Also, each block has access to a small portion of cache, thus, several threads can run simultaneously but operate in different data.

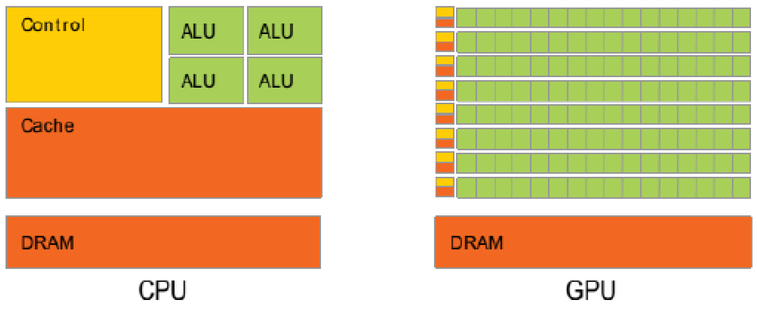


Fig. 5. CPU vs GPU architecture

The main advantage of a GPU is that they can run massive numbers of parallel threads. On the other hand, the disadvantages are the expensive cost of the hardware and the high energy consumption when running.

2.3 Implementation of the AES using CUDA

The AES algorithm implemented in CBC mode can not be parallelized by definition, because each round depends on the results from the previous round of encryption. Although, the transformations can be parallelized, it does not make sense to use a powerful GPU to compute operations over a state of only 16 bytes.

The approach used in this project focuses on executing in parallel the encryption of the blocks in which the file is divided. It is possible to do this because the encryption process for each block is independent from the others. For instance, let us suppose that a file of 1 MB = 1,048,576 bytes needs to be encrypted. This file will be divided in blocks of

16 bytes. Thus, the AES algorithm would be applied to 65,536 blocks. Here is where the parallelization can be used, to distribute the work between the available threads in the GPU.

Now, it is time to present the code written in CUDA C used to manage the cipher process. This method runs in the GPU. It manages the parallelization part, then it invokes the `cipher` method, which performs the AES over the state passed as parameter.

```
__global__ void cipher_control(byte *file_in, byte *file_out, long long *
    file_size, unsigned long *blocks, byte *expanded_key, byte *d_sbox, byte *
    d_m2, byte *d_m3)
{
    byte state[16];
    int block;
    int padding, res;
    // Get the number of the block that the current thread is managing
    block = blockIdx.x * blockDim.x + threadIdx.x;
    // Check if the size of the input file is multiple of 16
    res = *file_size % 16;
    // Verify the current block is not out of boundaries
    while(block < *blocks) {
        //Copy the corresponding input data to the state matrix
        memcpy(state, file_in + block  * 16, 16 * sizeof(byte));

        // Check if it is necessary to add padding to the last block
        if(block == ((*blocks) - 1) && res != 0) {
            padding = 16 - res;
            // Add padding only to the required spaces
            for(int i = res;i < res + padding;i++) {
                state[i] = 0x00;
            }
        }
        // Invoke the cipher process for the corresponding block
        cipher(state, expanded_key, d_sbox, d_m2, d_m3);
        // Copy the encrypted block to the output file
        memcpy(file_out + block  * 16, state, 16 * sizeof(byte));
        // Update the current block moving to the next section of memory
    allowed for this thread
28            block += gridDim.x * blockDim.x;
29        }
30 }
```

Listing 1: Function to control the AES algorithm running in the GPU

This function includes normal C syntax and some CUDA special words. The `__global__` indicates that the method will be executed in the GPU, which means that the same segment of code will be copied and executed in several threads at the same time. The most important instructions are those located in lines 7, 24 and 28. In line 7, the instruction `block = blockIdx.x * blockDim.x + threadIdx.x;` obtains the block number that the current thread needs to cipher. Since the program is running with several blocks of threads, it is important to get the correct index to avoid race conditions. In line 24, the `cipher` method is invoked, here is where the real AES algorithm takes place and perform operations over the current state. Finally, in line 27, the current ciphered block is updated by adding to it `gridDim.x * blockDim.x`. This operation guarantees that the next block to be ciphered by the current thread is not being ciphered by another one.

The `decipher_control` function is similar to the previous `decipher_control`, the main difference is the padding part. Since the size of the input files could be not a multiple of 16, a padding is added to fill the last block during the cipher process. The number of bytes that were added is stored in the first byte of the output file. Then, in the decipher process the information about the padding is read and those last bytes are ignored when saving the decrypted file.

The `cipher_control` method is invoked using the following instruction within the `main` of the program: `cipher_control <<< 128, 128>>> (/*params*/)`. This instruction tells the GPU to use 128*128 = 16,384 threads to execute the function. This number must be chosen carefully depending on the application. In this case it was selected as a recommendation found in [8], where is stated that due to hardware limitations it is not possible neither to use more than 65,535 threads nor more than 512 threads per block. In this project the optimum amount of threads should be equal to the size of the input file, however, this size is variable. When the amount of threads is greater than the required, there will be threads doing nothing but consuming resources, in the other hand, if the number is less then the needed the threads will have to perform more work. A grid of 128*128 is a reasonable number that can deal with small and big files. In further versions this number could be modified to test the results of the algorithm.

All the methods to read/write the input/output messages or files are managed by the CPU. All the other methods related to the AES algorithm, like AddRoundKey, MixColumns, ShiftRows, etc. are managed by the GPU.

The completed code can be found in the following repository: `https://github.com/dannyel2511/AESwithCUDA-ROS.git`

2.4 Integration of the AES-CUDA with ROS

This project is focused in the integration of the AES algorithm running in the GPU, with ROS. However, the computer on the robot needs to be a high-capable computational unit and requires a GPU from NVIDIA. Some of the proposed computers are:

- Any commercial laptop with a GPU computatible with CUDA 9.0. In this project, the PC used is an Acer SW314-52G-55WQ, with the NVIDIA MX150.
- The Jetson board from NVIDIA (TX1, TX2)

The integration of CUDA within ROS was done following the information proposed by https://github.com/air-lasca/ros-cuda. It consist of creating a ROS package containing CUDA as a dependency. Then, make some modifications to the CMakeList in order to compile the CUDA program with nvcc. Finally, the CUDA program is linked to the ROS node passing it as a pointer function. The interchange of data occurs trough this pointer function. Thus, the plain-text is passed to the AES-CUDA program as a reference to a location in the ROS node memory. Also, a pointer to an array of memory is passed to store the ciphered data. When the data is ciphered, it is returned by copying from GPU memory to the array reserved by the ROS node in the CPU.

By doing this, the integration of AES-CUDA within ROS looks easy. The package created is named `ros_cuda_aes`. It contains 2 nodes: `roscuda_aes_cipher` and `roscuda_aes_decipher`. The first one sends a ciphered text to the second one. Both are using the same key, and this is predefined in each node, (i.e. the key is not sent trought the network).

3 Results

The testing of the AES-CUDA program was conducted in two phases: the first one is to test the correct implementation and measure the execution time, this was done without the integration of ROS; the second one is to measure the latency generated when working with ROS, this uses two ROS nodes that send and receive RAW data from a camera.

In both testing phases, the machine that was used has the technical specs presented in Table 1.

CPU	Intel Core i5 (8th Gen) 8250U / 1.6 GHz
Number of cores	Quad-core
Memory	8 GB DDR4 SDRAM
Storage	256 GB SSD
GPU	NVIDIA GeForce MX150 - 2 GB GDDR5 SDRAM
OS	Linux Mint 18.3 Sylvia 64-bits

Table 1: Technical specs of the computer used to perform the testing

3.1 Test cases using static files of different sizes

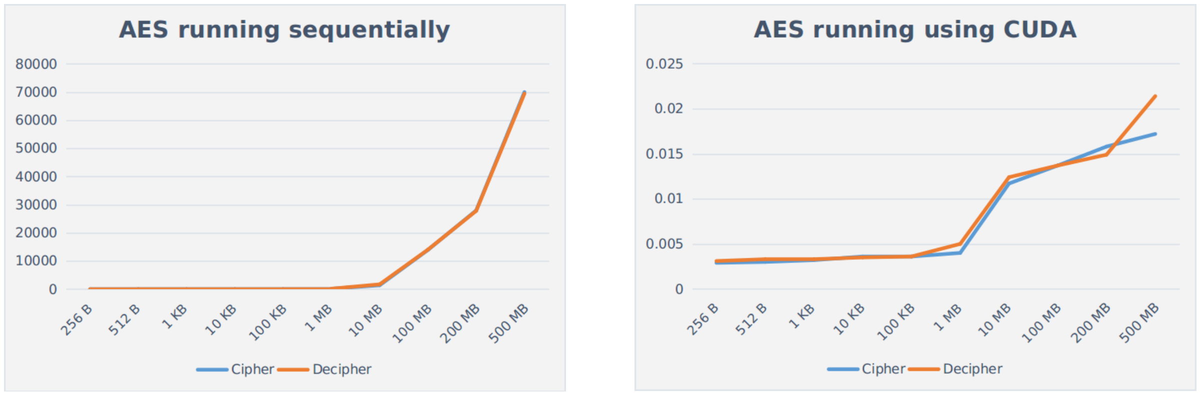
The first test was to use different types of files to be ciphered and deciphered. Then, comparing whether the deciphered file was exactly the same as the original input file. The result of this test was successful. For this case, the key was provided as a file of 16 bytes with the data: 0123456789ABCDEF.

The second test was to cipher and decipher files of different sizes from 256 B to 500 MB, and measure the execution time using a sequential implementation and the one using

CUDA. The Table. 2 shows the data obtained. All the times are registered in milliseconds. To measure the time, the algorithm was executed 10 times. Then, the average is reported in the chart. This was done trying to avoid the influence of external factors in the load of the CPU and GPU.

File size	**Sequential**		**CUDA**	
	Cipher time [ms]	**Decipher time [ms]**	**Cipher time [ms]**	**Decipher time [ms]**
256 B	0.1863	0.1658	0.0029	0.0031
512 B	0.3119	0.3074	0.003	0.0033
1 KB	0.5879	0.4356	0.0032	0.0033
10 KB	2.4344	2.6124	0.0036	0.0035
100 KB	15.4234	18.51	0.0036	0.0036
1 MB	140.8302	172.73	0.004	0.005
10 MB	1394.7174	1716.02	0.0117	0.0124
100 MB	13935.048	14035.06	0.0137	0.0137
200 MB	27959.4598	27807.53	0.0158	0.0149
500 MB	69984.218	69432.78	0.0172	0.0214

Table 2: Runtime measured for sequential and parallel AES when ciphering and deciphering several files



(a) **(b)**

Fig. 6. Runtime obtained for the cipher/decipher processes using sequential(a) and parallel(b) AES. Time is reported in ms

As it can be seen in Fig. 6a, there is a proportional relationship between the time taken to execute the AES and the size of the input file. To encrypt the big file of 500 MB it takes almost 70,000 ms. On the other hand, using CUDA the speedup has a huge increase, it takes only 0.02 milliseconds to encrypt the same file. In this case, the speedup is greater than 4,000. This result shows that the parallelization was successful and that the proposed AES-CUDA represents a good option to use the encryption algorithm in big files.

The Fig. 6b has a chart with the runtime measured for the AES-CUDA. The time measured at the beginning (encrypting small files) does not change until files greater than 1 MB. This behavior is caused by the number of threads that are running. For small files, all the 128*128 threads are active, even when they are not being used, i.e. there are a excessive number of threads. For files greater than 1 MB, the relationship becomes proportional because all the threads are being used and they have an equal distribution of the work.

Finally, the Fig. 7 presents a comparison of the runtime for both implementations and both processes, cipher and decipher. It was used a logarithmic scale, so the differences can be noted because there is a large range of quantities. It is clear that the AES-CUDA is superior to the normal AES.

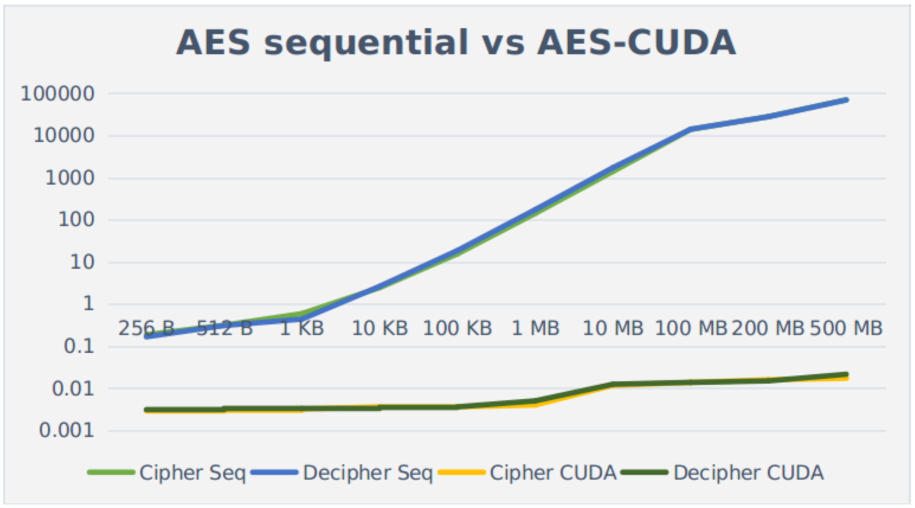


Fig. 7. Logarithmic comparison between the runtime obtained for sequential and parallel AES

3.2 Test cases using ROS messages

This phase of testing was done launching both ROS nodes. One of these ciphers a message, then it is published to the other node. This second one, receives the encrypted data and performs the decipher process. At the end, the message is printed to screen to see the result. It Fig. 8 it can be observed the nodes running in ROS.

The result of this test was successful. The original message was observed correctly at both ends. However, when trying to see the data using a third node, it was impossible because the data was ciphered.

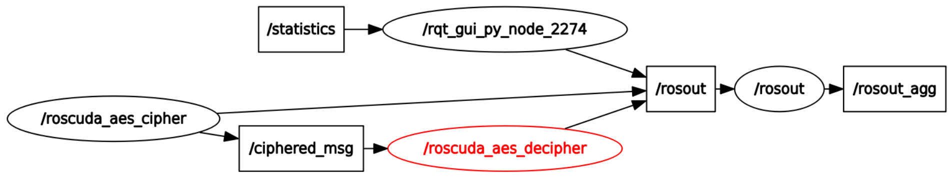


Fig. 8. Nodes running the AES cipher and decipher in ROS using CUDA

4 Conclusions

Cybersecurity in robots is something very important when using technology because we are exposed to the risks that this involves, like identity theft. However, when a robot development team is creating a new project using ROS, it could be exhaustive to create a security protocol to protect the data. This could interfere with the main application of the robot. That is why an alternative at the layer level of communication could be very helpful for developers. In this project, the AES-CUDA was proposed as an alternative to add a level of security to the messages sent by ROS nodes. Based on the results obtained when encrypting several files, the AES-CUDA has improved the runtimte of the normal AES with speedups of 4,000.

This project can contribute to the ROS community making easier the encryption of messages, as well as not affect performance due to the parallel implementation of CUDA. The current project needs further improvements, so it can receive different type of ROS messages and encrypt them. But, by now it is able to cipher and decipher messages of different sizes, so it can be taken as the starting point by other people and create more complete projects using AES CUDA ROS.

5 Setup instructions

5.1 AES and CUDA

To compile and run the AES-CUDA programs you must follow these steps. If you only want to run the executables go to step 4.

1. Install CUDA in your computer. You may find information about how to do this in https://developer.nvidia.com/cuda-zone.

2. Download the repository from https://github.com/dannyel2511/AESwithCUDA-ROS.git.

3. Compile the code using the instruction: `nvcc aes_cipher.cu -o encryp` or `nvcc aes_decipher.cu -o decryp`

4. (Windows) Execute the code using this instruction in the CMD: `encryp <input_name> <ciphered_file>` or `decryp <ciphered_file> <deciphered_file>`.

5. (Linux) Execute the code using this instruction in the terminal: `./encryp <input_name> <ciphered_file>` or `./decryp <ciphered_file> <deciphered_file>`.

6. The key used in the process can be changed by modifying the content of the `key.txt` file. Make sure to use only 16 characters.

5.2 AES-CUDA and ROS

If you want to execute the AES-CUDA running with ROS, first you will have to install ROS Kinetic.

1. Install ROS Kinetic in a computer running Ubun
2. u 16.04 (or above) or Linux Mint 17 (or above)
3. Install CUDA in your computer. You may find information about how to do this in https://developer.nvidia.com/cuda-zone.
4. Download the repository from `https://github.com/dannyel2511/AESwithCUDA-ROS.git` within your `catkin_ws` workspace.
5. Execute `catkin_make` in the directory `~/catkin_ws/src`
6. Launch the ROS launch included using `roslaunch ros_aes_cuda demo.launch`
7. After this, you will see an output data in the main terminal. If you want to see the ciphered data execute rostopic echo `/ciphered_msg in another terminal`

References

[1] FIPS. "Advanced Encryption Standard (AES)". In: (2001). URL: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

[2] Tully Foote. *ROS Community metrics report.* 2016. URL: http://download.ros.org/downloads/metrics/metrics-report-2016-07.pdf.

[3] Larry Hardesty. *Security for multirobot systems.* 2017. URL: http://news.mit.edu/2017/security-multirobot-systems-hackers-0317.

[4] D. Hood and W Woodall. *ROS 2 update - summary of alpha releases, architectural overview.* 2016. URL: https://goo.gl/oCHR7H.

[5] NVIDIA. *What is CUDA?* URL: https://www.nvidia.com/object/io_69526.html (visited on 11/19/2018).

[6] Francisco J. Rodríguez-Lera et al. "Message Encryption in Robot Operating System: Collateral Effects of Hardening Mobile Robots". In: *Frontiers in ICT* 5 (2018), p. 2. ISSN: 2297-198X. DOI: 10.3389/fict.2018.00002. URL: https://www.frontiersin.org/article/10.3389/fict.2018.00002.

[7] ROS. *About ROS*. 2018. URL: http://www.ros.org/about-ros/.

[8] Jason Sanders and Kandrot Edward. *CUDA by example. An introduction to general purpose GPU programming*. Addison-Wesley, 2010.

[9] Douglas Selent. *Advanced Encryption Standard*. 2010. URL: https://www2.rivier.edu/journal/ROAJ-Fall-2010/J455-Selent-AES.pdf.