POLITECNICO DI TORINO

Master’s Degree in Mechatronic Engineering

Master’s Degree Thesis

Evaluation and Enhancement of Security in Serial Communication for Mechatronic Systems

Supervisor:
Prof. Marco VACCA

Company Supervisor:
Luca BUSSI

Candidate:

Pasquale CUPI

ACADEMIC YEAR 2023/2024

Abstract

Most of the modern mechatronic systems are equipped with a lot of MCUs that exchange a vast amount of sensitive data among themselves through serial connections, all in plain text and without any protection. This makes them vulnerable to malicious actors who could manipulate original devices and transmit unauthorized messages to sabotage the system.
Therefore, there is a need to introduce protections in serial communications to avoid the decryption of the messages.
To tackle this challenge, I used an STM32F401RE microcontroller as the system to hack and my PC equipped with a COM port connected to a USB-UART adapter as the hacking device. The microcontroller was programmed and configured in C language, while the PC runs a Python program aimed at launching various attacks and monitoring their computational and temporal costs.
The initial step involves discovering the microcontroller's UART configuration parameters using the PC. The problem isn't simple because there are hundreds of configurations that need to be tested, and we have no prior information about the correct one. So, the only method is to test them all, starting with the most common ones and then moving on to the less-used ones, in order to find meaningful messages on the terminal. If a message is composed only of alphanumeric words, the associated configuration is saved in the list of potential configurations used by the microcontroller.
Upon successfully compromising the system and observing the ease with which it was breached, it became evident that enhancing the security of microcontroller communications is paramount.
To address this, I introduced the AES-128 algorithm for encrypting incoming and outgoing messages. This widely used algorithm in cryptography manipulates messages with complex mathematical operations to ensure robust encryption and decryption. The system uses a symmetric key for both encryption and decryption. While this makes it easier to implement and use as a communication system, it forces protection of the key from discovery by third parties.
Analyzing the messages transmitted by the microcontroller, the smartest way to break the system was to use a data stream of a specific size, like 64 or 128 bytes, as the key and the remaining text as the encrypted message. This attack was successful, demonstrating that by sending the key in plain text along with the message, an attacker could easily recognize the communication protocol and decipher the message quickly. If this method did not yield meaningful results, other types of attacks, such as dictionary attacks and brute force, had to be adopted. These are less efficient due to their high computational cost but could be a good option if nothing smarter was available.
Given the vulnerabilities of the AES system, I introduced an additional security layer called RSA, which could be combined with or used independently of the AES algorithm. This algorithm operates in a more complex way and bases its robustness on the difficulty of factoring large prime numbers, a task impractical for most

computers. The biggest difference from the AES algorithm is that RSA is asymmetric: one key is public and used by anyone to encrypt data sent to the system, while the other key is private and kept secret to decrypt incoming messages. Now, with this algorithm, only the encryption key needs to be transmitted, and even if it is leaked, it cannot decrypt any messages without the corresponding private key. The only drawback is that understanding how the communication protocol works is not as immediate as with AES, but the security it provides is very high for our applications. As I did with the AES algorithm, I repeated all the attacks on the upgraded system, and none of them were able to break it. Satisfied with the results, I proceeded with implementing the security system on various devices.

Index

List of figures

List of code blocks

Introduction

Scenario

The Emergence of Cybercrime

Cybersecurity is a growing concern in today's world. In the past year, thousands systems worldwide were hacked, with small and medium-sized enterprises being the primary targets. Italy has also been significantly affected, with 1,382 cyber-attacks in the first half of 2023, indicating a risky rising trend. [1]

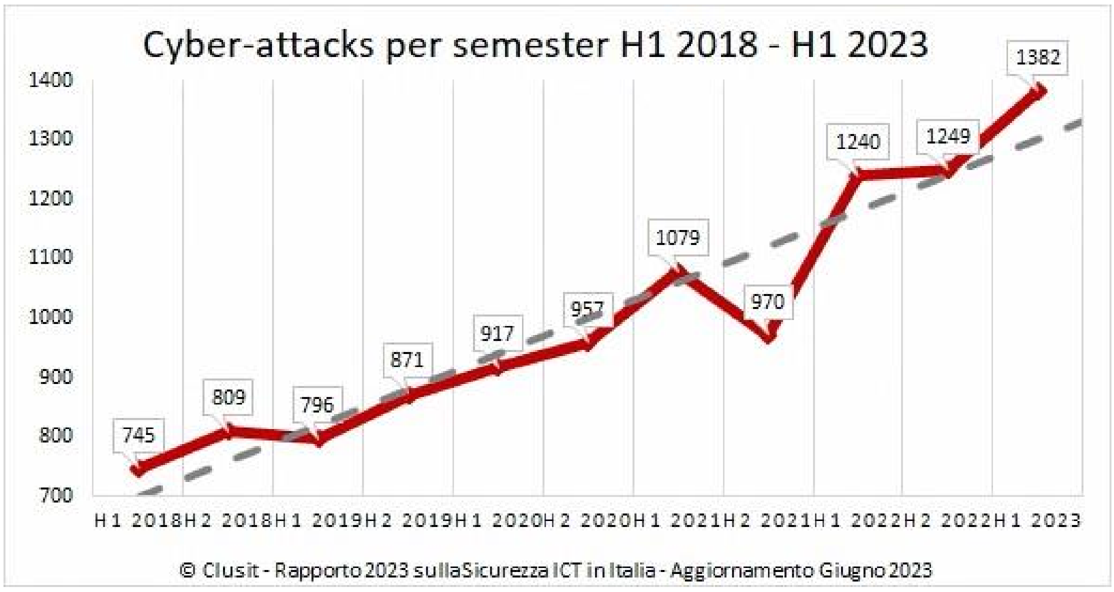


Figure 1 - Trend of the Cyber-attacks per semester.

Considering the problem on a larger scale, cyber-attacks have increased by 61.5%. Italy, on the other hand, has experienced a significant territorial increase of 300%.

The main objective of these attacks is to extort money or steal sensitive data for illicit purposes. Hackers frequently target devices remotely, making it difficult to trace their origin. Although cybersecurity measures in this area are highly advanced, leaving little room for error, one area remains underdeveloped in terms of cybersecurity is embedded systems implemented within IoT products.

Therefore, this thesis aims to minimize the issue considering the increasing significance of security measures in mechatronic systems.

The Importance of Cybersecurity in Embedded Systems

Security in microcontrollers encompasses several aspects, including the protection of firmware intellectual property, safeguarding private data within the device, and ensuring the execution of services.

Examining all potential scenarios, we could consider a gate opening system that operates via authentication code number, revealing a vulnerability: intercepting the code could compromise the security of an entire apartment. Another similar example could involve duplicating the authentication key of a vehicle's ignition system, or, in the worst-case scenario, tampering with the trajectory of a missile. Hence, born the necessity to adopt a generalized approach to safeguard all types of integrated systems.

When designing an integrated system, the first step is to protect the hardware from tampering, followed by securing the firmware.
We particularly focus in this thesis on the most vulnerable part of the microcontroller: the communication interface with the external environment, as it represents the most direct path for system manipulation.

An external device can intercept and analyze every communication transmitted between the microcontroller and external entities. This implies that each message is subject to interception, examination, and possible manipulation by this external device, without exception. While a read-only device might access sensitive data without authorization, a compromised input port could allow the device to replace the original and take control of communication, sending unchecked messages to the master!

Therefore, securing the serial port is essential, as it could trigger a chain reaction affecting the entire mechatronic system in which the device is integrated.

Thesis Objectives

In order to tackle this challenge effectively, my primary objective is to fortify my system against the most common cyber attacks. This is achieved by strategically implementing additional layers of security protocols that can be progressively activated in response to the compromise time, the computational cost, and other parameters that measure the witness of the system.

Both roles are assumed, one as a malicious actor when breaching the system is intended, and the other as the victim when enhancing the protection of the target system is required.
This approach allows us to have a complete understanding of the weaknesses of the system.

As cyber defender, the objectives are as follows:

- Ensuring that messages sent and received remain undetected by any potential snooper.
- Keeping the communication port configuration unknown to external sources.
- Protecting the secrecy of the encryption method employed.
- Preventing the discovery of the decryption key.
- Maintaining the full functionality of the system.
- Optimizing the code for lightweight, portability, and universality.
- Providing comprehensive and explanatory documentation.

On the other hand, as a cracker, objectives involve:

- Decoding the contents of transmitted messages.
- Deciphering the encryption protocol used.
- Understanding the configuration of the microcontroller.
- Facilitating the discovery of the decryption key.
- Minimizing the computational resources required for the attacks.
- Erasing all traces of the activities.
- Using acquired knowledge to potentially replace the receiver and take control of communications.

Thesis Overview

Given the highly experimental nature of the thesis, a chronological order approach has been chosen, wherein each chapter represents a progressive step taken to enhance the security of the microcontroller.

Specifically, the first section provides an overview of the workbench setup, explaining the chosen microcontroller configuration and the methods of interfacing with the PC.

Each following subsection offers a brief theoretical background upon which the code implementation is founded.

Subsequently, the encryption algorithms used, namely AES and RSA, are discussed. They are first explained theoretically and then their implementation is shown. This exploration is followed by an examination of the attacks employed to monitor the security indicators utilized.

Finally, a comparative analysis of the results obtained is conducted, assessing the fulfillment of predetermined objectives and exploring both positive and negative implications arising from the findings. Additionally, potential enhancements, future progress, and optimizations for the analyzed system are discussed.

Chapter 1: Testing Setup

Workbench Setup

The security system implemented requires a wide range of applications, which is why the stm32f401re microcontroller and C programming language have been selected as the target device and programming language, respectively.

The stm32f401re is a reprogrammable microcontroller that is well-suited for various applications and has the necessary features for our purposes.

- Easily configurable UART port
- HAL functions facilitating the communication management
- Easy implementation of external libraries
- Dedicated IDE based on Eclipse, allowing both code writing and debugging.

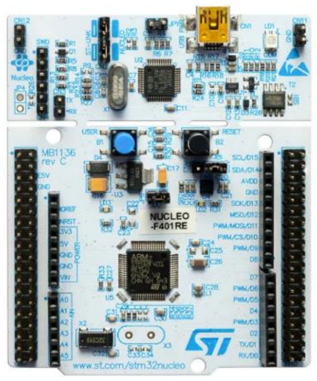


Figure 2 - NUCLEO F401RE

The only drawback (which, from the perspective of a potential cracker, can be a disadvantage) is that it has only one USB communication channel to the computer. Therefore, if someone wants to read the serial port with another USB port (no matter if it is from the same PC or a third-party one), a USB to UART adapter is necessary to connect the transmission pins on the board to the PC's USB port.

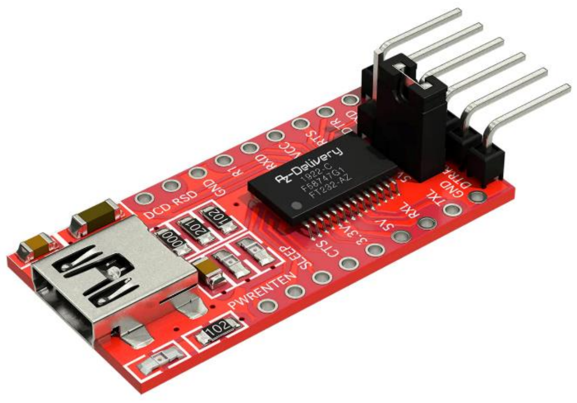


Figure 3 - FT232-AZ USB to TTL Serial Adapter

While the adapter may be affordable, finding the correct configuration at both the hardware and software levels can often require trial and error technique that could discourage the cybercriminal. Later on, a detailed discussion will be provided on this matter.

The experiment was conducted using a Dell XPS 13 with a dual-core processor clocked at 2.50 GHz and 16 gigabytes of RAM. The console application used was entirely written in Python.
To interface with the microcontroller, it was necessary to use a USB-UART adapter connected to the computer, following the guidelines written in the datasheet. Regarding the USB serial port, configuration was performed at the software level using a dedicated Python library.

The advantages of using a Python application to execute attacks and evaluate benchmark results are as follows:

- Easy configuration of the serial port
- Python is the most widely used software worldwide on Windows, so it is likely to be used for attacking the system
- Easily available and implementable external libraries
- Unique code design simplicity and portability
- Availability of dedicated benchmark packages.

In addition, to read real-time transmitted messages, Realterm software can be used as a debugging terminal. Its purpose is to print exchanged messages on the screen based on the serial port configuration.

Putting everything together, the system is connected as follows:

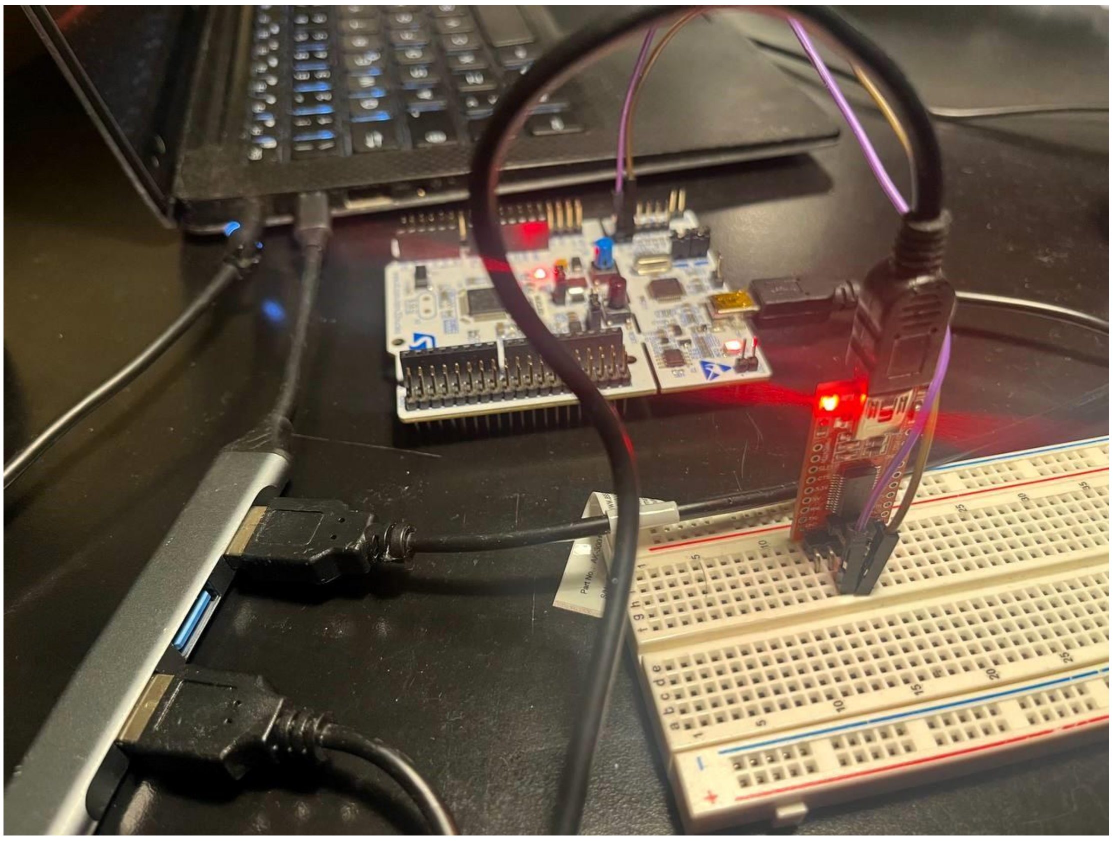


Figure 4 - Workbech setup

The microcontroller sends and receives messages from the PC's USB port via the black mini-USB to USB-A cable, and the PC reads the data in real-time through Realterm. However, the potential hacker interferes with the transmission and connects to the microcontroller via the pins coming out of the board, which are directed to the PC's second USB port via the adapter. Here, the PC receives them and feeds them to the Python application.

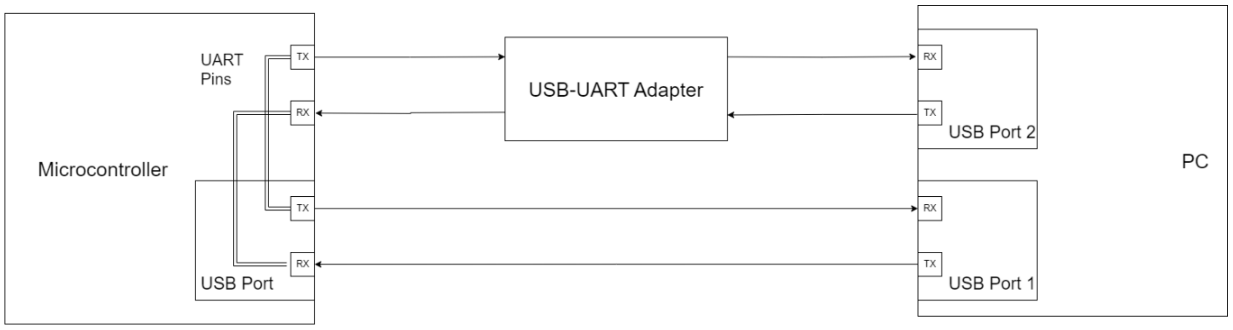


Figure 5 - Workbench connections diagram

Microcontroller Setup

The research begins by downloading and installing the IDE from the following link: [https://www.st.com/en/development-tools/stm32cubeide.html#get-software]

After the installation is complete, the IDE (Integrated Development Environment) and the necessary drivers for debugging the board are ready for use.
(The installation process will not be detailed here as it is straightforward.)

As mentioned on the website, after selecting the preconfigured microcontroller from the board selection, the project is created, and initialization code is generated. At any point during development, the user can revisit the initialization and configuration of peripherals or middleware and regenerate the initialization code without impacting the user code.

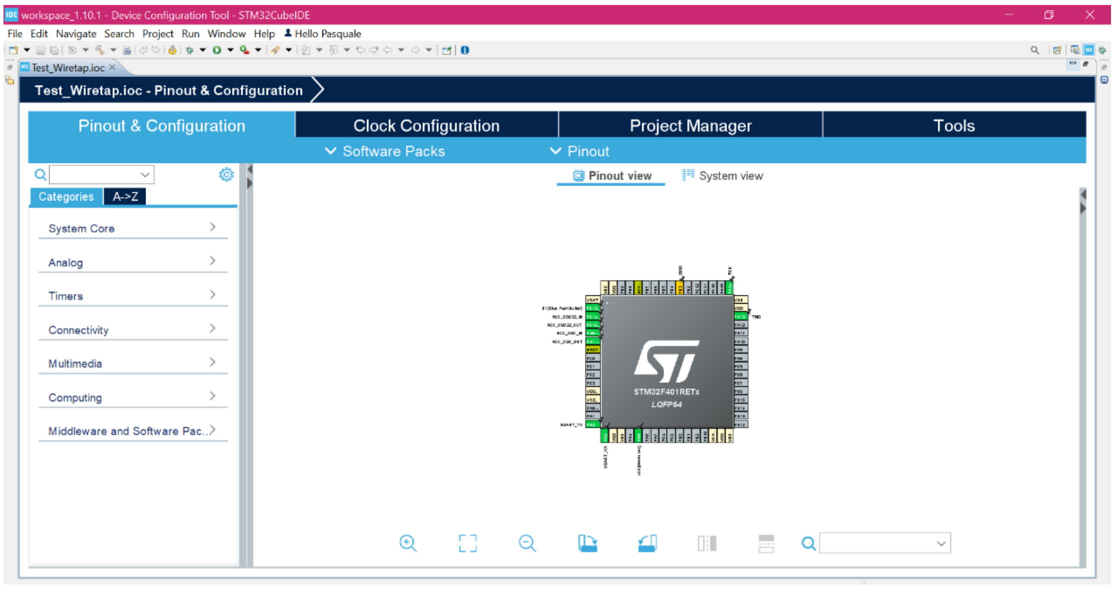


Figure 6 - Configuration tool of a new project in STM32CubeIDE

Considering that the microcontroller is often intended for testing the security of serial communication, the essential functionalities to activate are the timer for sending periodic message and the UART port discussed in detail in the next sections.

The UART protocol

The UART (Universal Asynchronous Receiver Transmitter) is a crucial communication tool widely used in industries. Almost every microcontroller has at least one UART, built right into the microprocessor.

The primary function of this mechanism is to send and receive data bit by bit over a single line, as opposed to parallel methods where data travels over multiple lines simultaneously.

This transition to serial transmission has effectively addressed numerous issues:

- **Cost:** Managing lots of data lines gets expensive because it needs more parts and careful design.
- **Power Usage:** Running multiple lines at once needs more energy, especially to keep the signals strong.
- **Size and Weight:** More lines mean more wires, which can get heavy and take up space. It is tough fitting them into small circuit boards or organizing them in tight spaces like electrical panels.
- **Reliability:** Getting the pins aligned just right is crucial for good connections. With parallel transmissions, there's more chance of things wearing out or breaking, causing connection problems.
- **Timing:** Making sure all the signals line up perfectly is tricky with parallel transmissions. They usually end up with some delay or mismatch.

Considering all these challenges, especially in mechatronic applications, employing serial communication emerges as the optimal choice.

Delving into the mechanics, when initiating communication, the UART sequentially places bits onto the line, respecting a predetermined order for the receiver to interpret the message. This message structure, known as a frame, is composed by:

1. **Start Bit:** It tells the receiver that a new message is starting.
2. **Payload:** This is the main message, starting with the least important bit and ending with the most important one. Sometimes, there's a special bit to check if the message is right.
3. **Stop Bit:** It shows that the message is finished.

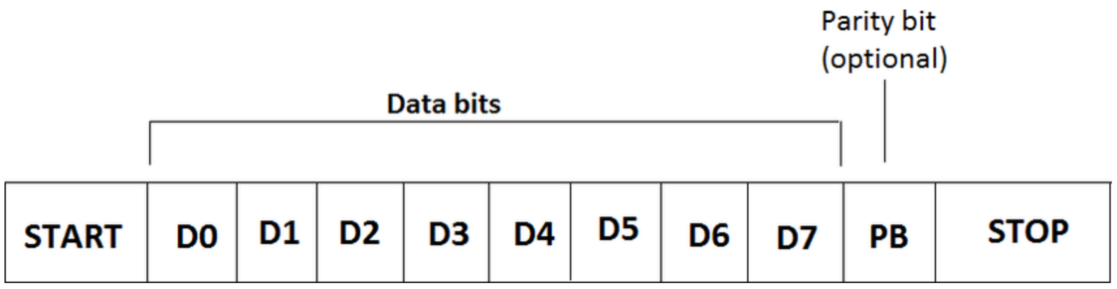


Figure 7 - Bit assignment of the UART message

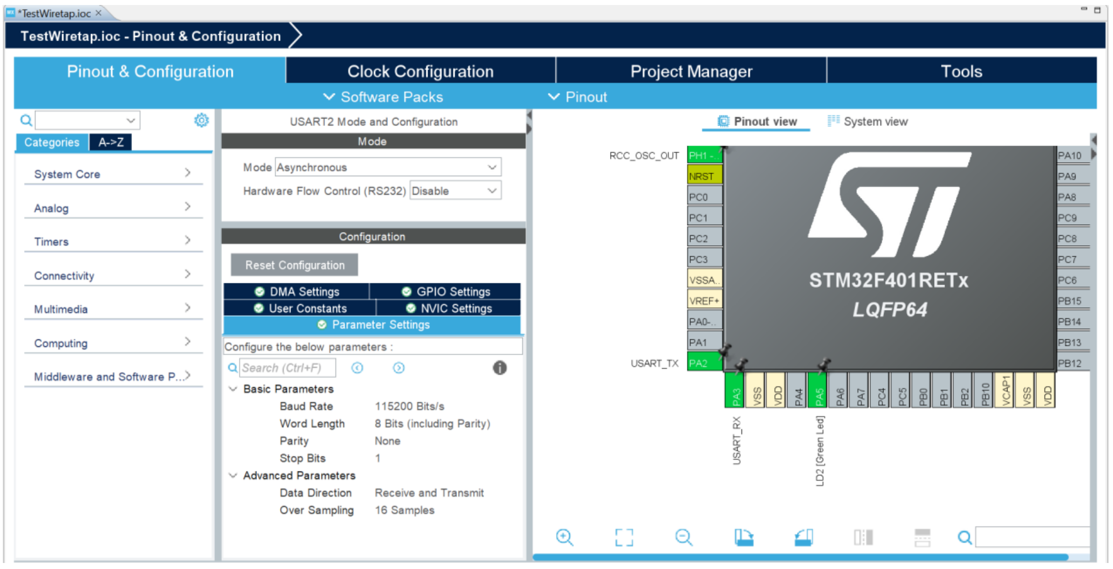
[2]
Factors such as Baud rate (data transmission speed), line sampling frequency, idle value, and pin mapping must also be considered during configuration.

Unlike other protocols, UART enables independent transmission and reception of data by both devices. Therefore, although the behavior of the UART is straightforward, accurate manual management of message traffic is necessary. Detailed explanations on data transmission and reception using this protocol will follow in the subsequent chapters.

To initialize the huart, appropriate settings must be selected from the configuration interface menu when starting a new project. In this case, a standard configuration has been chosen.

- **Baudrate:** 115200 bits/s
- **Parity:** None
- **Word length:** 8 bits
- **Communication Mode:** Full duplex
- **Data Direction:** Transmitting and Receiving
- **Oversampling:** 16 samples

Once the interrupt service routines are enabled to execute specific functions upon data arrival or transmission, the configuration process is complete.



Mode Asynchronous
Hardware Flow Control (RS232) Disable

Configure the below parameters :

Basic Parameters

Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1

Advanced Parameters

Data Direction	Receive and Transmit
Over Sampling	16 Samples

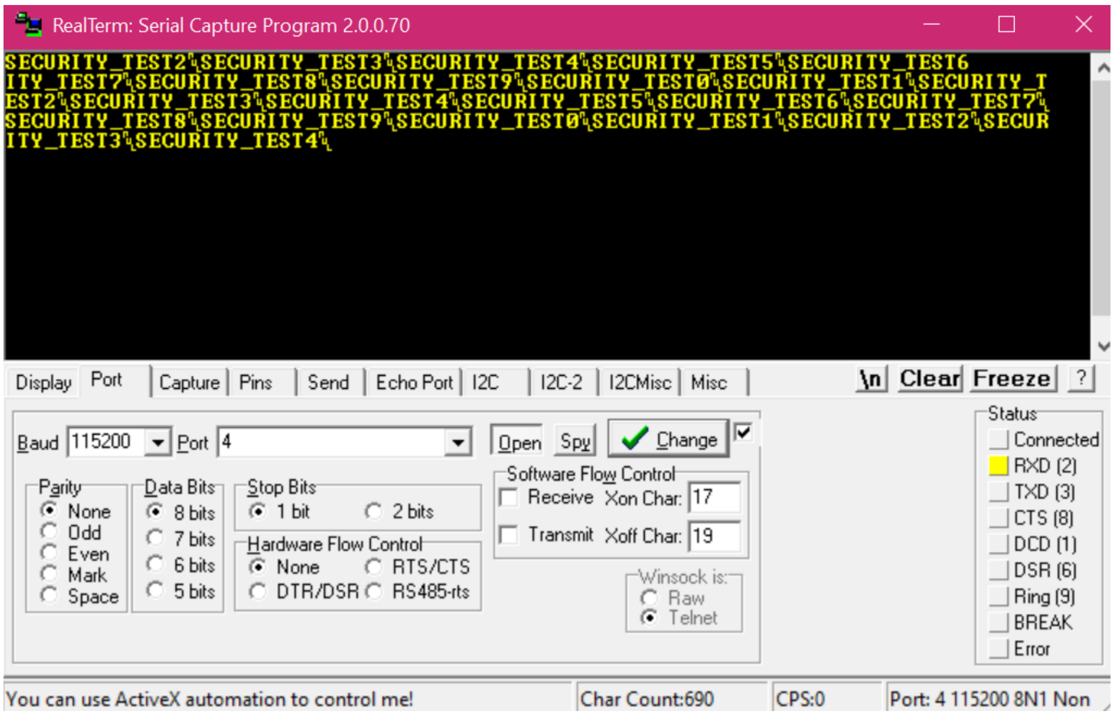
Figure 8 - Configuration of the USART on the IDE

From the pin mapping, the pin designated for data transmission is PA2, and the one for reception is PA3. However, upon inspecting the signal on these pins with an oscilloscope, it is discovered that it is null.

According to a discussion on this forum website [3], it is concluded that the board is configured to route the UART2 signals to the ST-Link instead of the pin connectors. This explains why data can be received over USB but not observed on the pins.

PC Setup

Configuring everything from the computer's perspective was remarkably straightforward. Initially, the process involved navigating to this link, [https://sourceforge.net/projects/realterm/], and downloading the serial terminal, an essential tool for interfacing with hardware devices. RealTerm was selected for its reliability, robust feature set, completeness, and ease of use.



```
SECURITY_TEST2␊SECURITY_TEST3␊SECURITY_TEST4␊SECURITY_TEST5␊SECURITY_TEST6
ITY_TEST7␊SECURITY_TEST8␊SECURITY_TEST9␊SECURITY_TEST0␊SECURITY_TEST1␊SECURITY_T
EST2␊SECURITY_TEST3␊SECURITY_TEST4␊SECURITY_TEST5␊SECURITY_TEST6␊SECURITY_TEST7␊
SECURITY_TEST8␊SECURITY_TEST9␊SECURITY_TEST0␊SECURITY_TEST1␊SECURITY_TEST2␊SECUR
ITY_TEST3␊SECURITY_TEST4␊
```

Baud 115200 Port 4

Parity: [✓ None] Odd Even Mark Space

Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits

Stop Bits: [✓ 1 bit] 2 bits

Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts

Software Flow Control: ☐ Receive Xon Char: 17; ☐ Transmit Xoff Char: 19

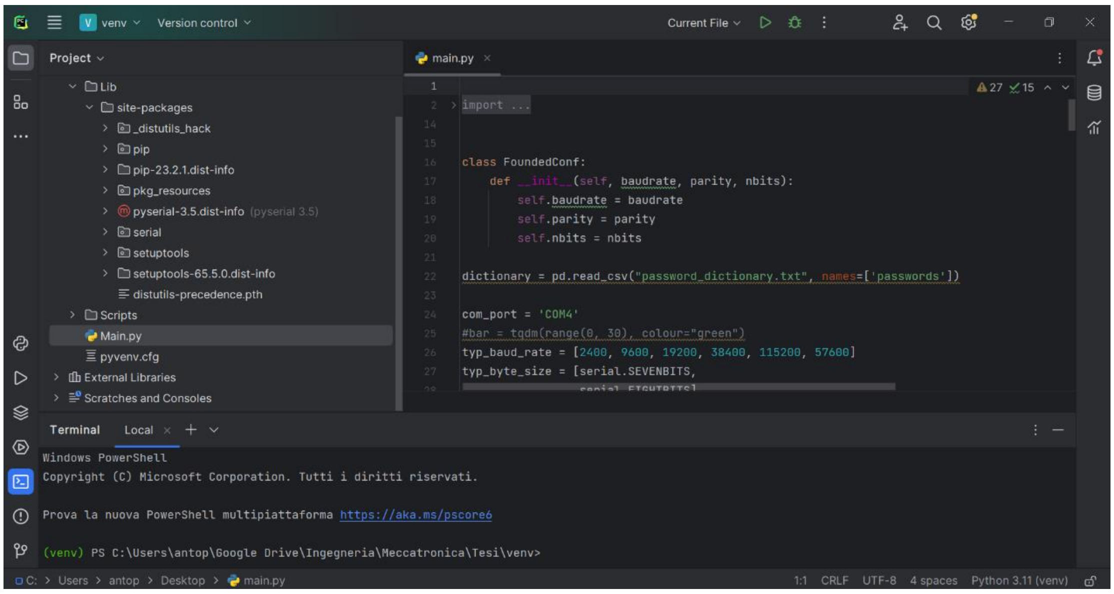
Winsock is: Raw [✓ Telnet]

Status: ☐ Connected, 🟨 RXD (2), ☐ TXD (3), ☐ CTS (8), ☐ DCD (1), ☐ DSR (6), ☐ Ring (9), ☐ BREAK, ☐ Error

You can use ActiveX automation to control me! | Char Count:690 | CPS:0 | Port: 4 115200 8N1 Non

Figure 9 - Interface of Realterm software

Once the RealTerm application was successfully installed, the next step was to set up the integrated development environment (IDE) for coding and testing purposes. Among the various IDE options available, PyCharm 2023 was chosen for its comprehensive suite of features and reputation for being user-friendly, making it an ideal environment for Python development. The latest version was obtained directly from the official website to ensure compatibility and access to the latest enhancements.



```python

import ...


class FoundedConf:
    def __init__(self, baudrate, parity, nbits):
        self.baudrate = baudrate
        self.parity = parity
        self.nbits = nbits

dictionary = pd.read_csv("password_dictionary.txt", names=['passwords'])

com_port = 'COM4'
#bar = tqdm(range(0, 30), colour="green")
typ_baud_rate = [2400, 9600, 19200, 38400, 115200, 57600]
typ_byte_size = [serial.SEVENBITS,
                 serial.EIGHTBITS]
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. Tutti i diritti riservati.

Prova la nuova PowerShell multipiattaforma https://aka.ms/pscore6

(venv) PS C:\Users\antop\Google Drive\Ingegneria\Meccatronica\Tesi\venv>
```

Figure 10 - Interface of PyCharm

Upon completion of the installation, PyCharm was launched, and a new project was created. Python 3.10.11 was selected as the interpreter version, and the project structure, including the main file and its corresponding virtual environment, was meticulously set up.

The creation of the virtual environment was a fundamental step as it encapsulated all project dependencies and libraries essential for seamless execution and portability. This meticulous setup ensured that the project environment was isolated and well-equipped to handle any future developments or modifications. Once that's done, we can proceed with the configuration of the serial port.

The COM Port

A communications port, also known as a serial port or COM port, is a hardware interface on a computer that allow us to connect external devices for data transfer. It serves as a bridge between your computer and peripherals enabling serial communication between them.

In the older computer there was a dedicated connector called RS232 with 9 pins, where every one of them has its own function:

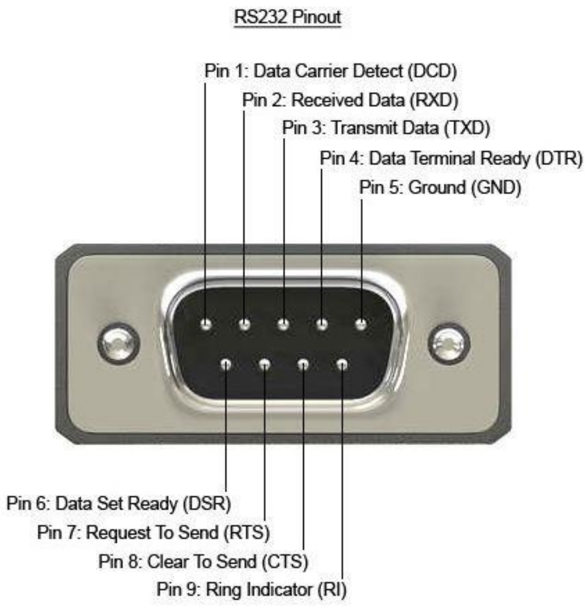

RS232 Pinout
Pin 1: Data Carrier Detect (DCD)
Pin 2: Received Data (RXD)
Pin 3: Transmit Data (TXD)
Pin 4: Data Terminal Ready (DTR)
Pin 5: Ground (GND)
Pin 6: Data Set Ready (DSR)
Pin 7: Request To Send (RTS)
Pin 8: Clear To Send (CTS)
Pin 9: Ring Indicator (RI)

Figure 11 - RS232 Pinout [3]

[3]

1. Data Carrier Detect - After a data terminal is detected, a signal is sent to the data set that is going to be transmitted to the terminal.

2. Received Data - The data set receives the initial signal via the receive data line (RxD).

3. Transmitted Data - The data terminal gets a signal from the data set, a confirmation that there is a connection between the data terminal and the data set.

4. Data Terminal Ready - A positive voltage is applied to the data terminal ready (DTR) line, a sign that the data terminal is prepared for the transmission of data.

5. Signal Ground - A return for all the signals on a single interface, the signal ground (SG) offers a return path for serial communications. Without SG, serial data cannot be transmitted between devices.

6. Data Set Ready - A positive voltage is applied to the data set ready (DSR) line, which ensures the serial communications between a data terminal and a data set can be completed.

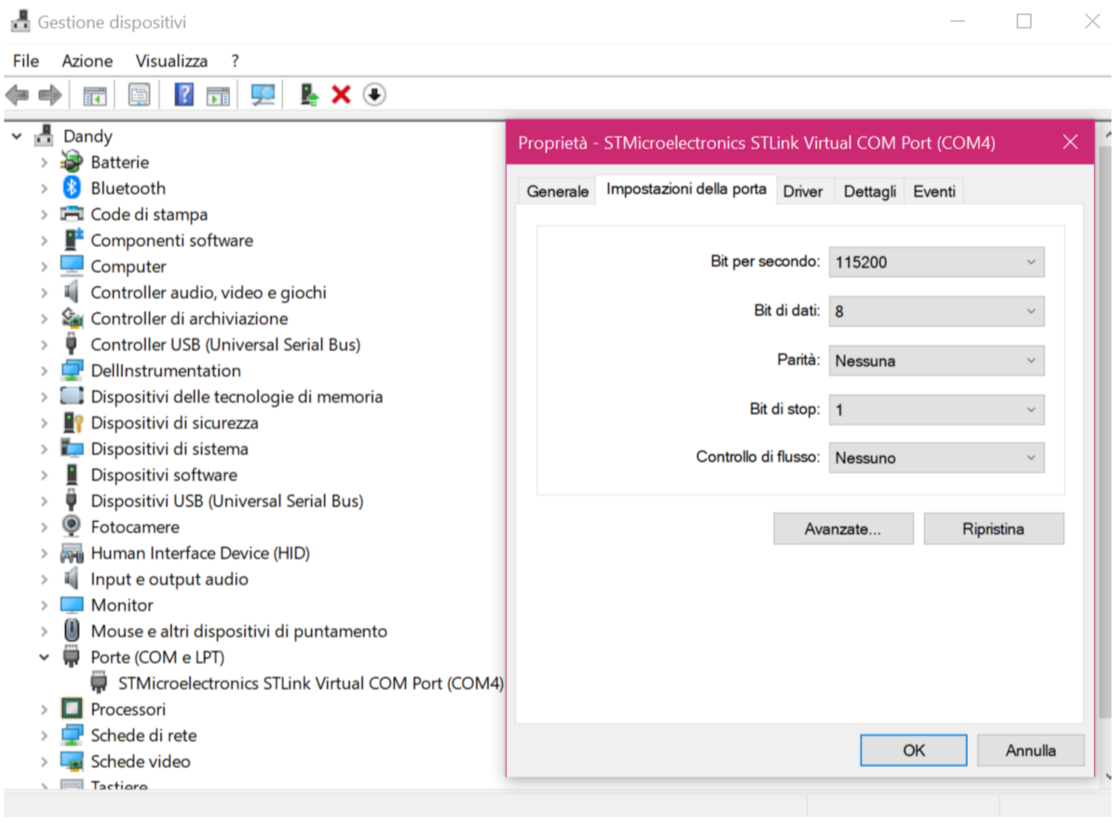
7. Request to Send - A positive voltage indicates the request to send (RTS) can be performed, which means the data set is able to send information to the data terminal without interference.

8. Clear to Send - After a connection has been established between a data terminal and a distant modem, a clear to send (CS) signal ensures the data terminal recognizes that communications can be performed.

1. Ring Indicator - The ring indicator (RI) signal will be activated if a modem that operates as a data set detects low frequency. When this occurs, the data terminal is alerted, but the RI will not stop the flow of serial data between devices

In our case, we don't use the traditional connector; instead, we directly utilize a USB connection that works as the COM port, facilitated by an internal adapter.

In order to configure the UART port of the microcontroller, it is simply necessary to access the Device Manager and set the appropriate parameters.


Proprietà - STMicroelectronics STLink Virtual COM Port (COM4)

Bit per secondo:	115200
Bit di dati:	8
Parità:	Nessuna
Bit di stop:	1
Controllo di flusso:	Nessuno

Figure 12 - Configuration of COM port in Device Manager

Chapter 2: Clear Message Evaluation

Introduction

Upon initial analysis, let's delve into the most common scenario where message protection is absent. There are several reasons why data transmission might lack security measures, among which the most common include:

- **Cost:** Implementing security measures can be expensive, both in terms of time and money. This can pose a barrier for small businesses or organizations with limited resources.

- **Complexity:** Depending on the complexity of the communication line, it may be challenging or even impossible to fully secure it.

- **Lack of priority:** Security might not be deemed a priority for the organization, which may be more focused on other aspects such as functionality or cost.

- **Exposure to threats:** The level of protection required for a communication line depends on the level of threat it faces. For instance, a communication line used to transmit sensitive information will need a higher level of protection than one used for transmitting non-sensitive information.

- **Trust:** Users of the communication line may trust each other and not see the need to protect their communications. A concrete example is when communications occur between components produced by the same company and are not intended to be read by anyone else.

Given these reasons, it is important to consider the situation where messages are transmitted in clear text, thus it is useful to analyze this configuration as well. Moreover, this can be a useful step in better structuring the tasks to be carried out during the process of securing a telecommunication channel and understanding which security benchmarks and analyses need to be successfully surpassed for a communication line to be deemed secure.

Let's begin by configuring our microcontroller to receive and transmit messages in a completely transparent manner to anyone who wishes to read the message.

Given the wide range of applications and the significant development potential, a modular and generic approach has been chosen to ensure flexibility, portability, and code universality as much as possible.

A special module containing the header file UART_Comm.h and the source file UART_Comm.c has been created to manage communication with the outside world.

In the first file there are all the public macros, the public functions prototypes designed for communicating and an enum that indicates the error code corresponding to the return value of the module functions.

The second file instead contains all the implementations of the public functions, private macros and global variables. As public macros there are only the activation of the various security levels, and therefore of portions of code present in the implementations. Instead, the macros used to activate the portions of code to be tested and the lengths of the private vectors used are present as private data.

We will delve deeper into the implementation of the code when needed.

Message Transmission

Communicating effectively is essential for humans to connect and be understood.
The same is true for electronic devices.
It is impossible to imagine a mechatronic system without considering the exchange of data among its components.
There are numerous reasons why data transmission is crucial, including:

- Facilitating communication between different parts of the system, enabling them to exchange data, instructions, and feedback.
- Coordinating activities within the system to ensure various parts work together harmoniously and efficiently.
- Providing important data and details that can be used to make informed decisions within the system.
- Monitoring and controlling operations within the system, allowing for corrections or adjustments as needed.
- Using transmitted information for learning and continuous improvement of the system, enabling it to adapt to new conditions.

Given these factors, it is important to have a robust and efficient transmission management system.

At the implementation level, the HAL functions of the microcontroller are used to ensure the necessary security and stability, and to simplify the implementation.
The first fundamental step is to initialize all module data. To achieve this, there is a dedicated public function that must be called during the microcontroller configuration phase. Inside this function, all module values are initialized, and upon completion, it returns a return value to ensure proper execution.

```
UART_COM UART_COM_Init (UART_HandleTypeDef *huart)
{
	if(huart!= NULL)
	{
		pmyhuart = huart;

		for(uint8_t i=0; i<TX_BUF_SIZE; i++)
		{
			abyTXBuf[i]=0;
			abyRXBuf[i]=0;
		}

#ifdef AES_CONF

		for(uint8_t i=0; i<KEY_SIZE; i++)
		{

	#ifdef CLEAR_PSW

			if(i<sizeof(testPSW))
			{
```

```
                                abyKeyEn[i] = testPSW[i];
                        }
                        else
                        {
                                abyKeyEn[i] = 0;
                        }

        #else
                                abyKeyEn[i]=rand()%UINT8_MAX;
        #endif

                        mx_decripted[i]=0;
                        mx_encripted[i]=0;
                }

#endif
#ifdef RSA_CONF

                if(MicroRSA_GenerateKeys (7, 11, 17))
                        return UART_COM_OK;
                else
                        return UART_COM_PAR_ERR ;

#endif

        }
        else
        {
                return UART_COM_PAR_ERR ;
        }
}
```

Code block 1 - Implementation of UART_COM_Init

Once the initialization is completed, the system is ready to transmit data.

For simplicity, the transmission is scheduled by a timer that sends a new message to the PC every second. When the timer triggers an interrupt service routine (ISR) every second from the previous transmission, the transmission function is executed within the ISR. This function is a public function of the UART_Comm module, which internally calls the HAL function responsible for the interrupt transmission of a data bus. It takes only two parameters as input: a pointer to the data bus to be transmitted and its size.

```
UART_COM UART_COM_Send (uint8_t *pbyTXBuf, uint16_t wdBufSize);
```

Code block 2 - Prototype of UART_COM_Send function

Checks are performed on both parameters at the beginning and if something goes wrong return the id of the error.

Regarding the message content, since our system is purely simulative and does not involve any specific functionality other than testing serial communication, it was decided to write the following message to the buffer:

"SECURITY_TESTN"

The last digit is a number between 0 and 9, indicating the test instance number and illustrating how the transmission buffer changes over time. There is a line of code in the ISR to do the character update.

```
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef *htim)
{

UART_COM_Send(abyTest,sizeof(abyTest));

//Updating of the last printable character with an increasing digit from
0 to 9
abyTest[sizeof(abyTest)-2] = (abyTest[sizeof(abyTest)-2] -'0'+1) % 10
+'0';
}
```

Code block 3 - Implementation of HAL_TIM_OC_DelayElapsedCallback

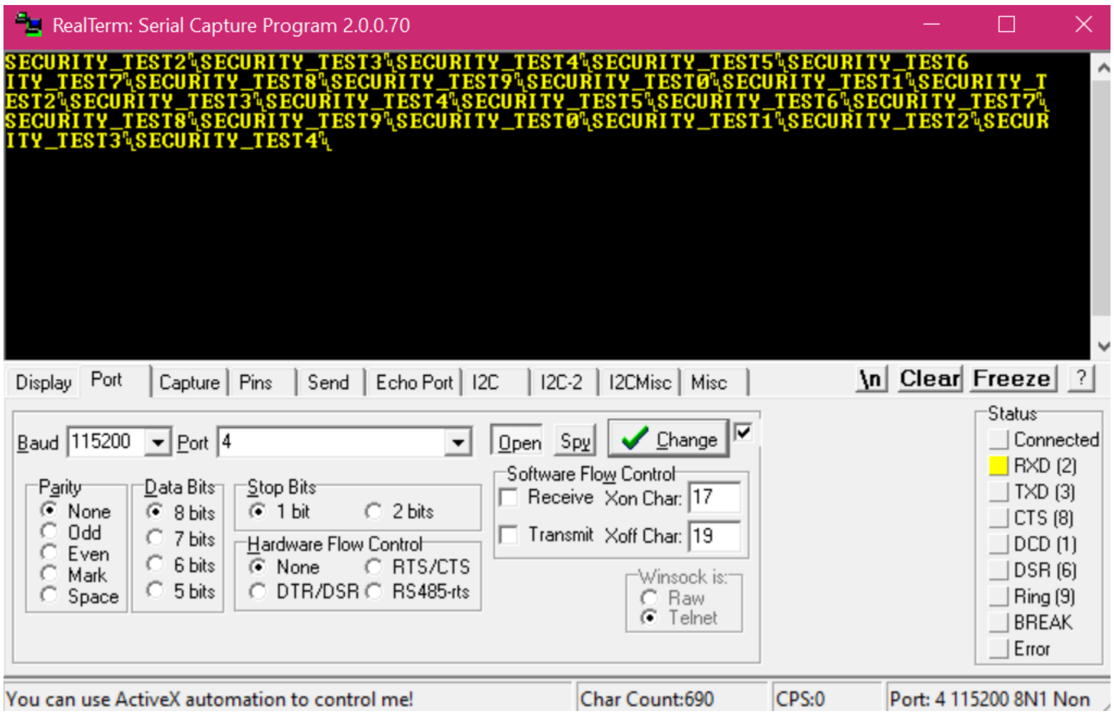
Before incrementing, the offset relative to the ASCII character '0' is subtracted from the penultimate character (including the end-of-string character). This simplifies subsequent mathematical operations. Once this is done, it is incremented by one, advancing the previous digit by one step. To limit the value between 0 and 9, a common technique in the embedded domain is used: comparing the result of the operation with the maximum reachable value and saving only the remainder. To better understand the mechanism, here's a numerical example:

(0+1)%4=1 (1/4=0 with remainder 1)
(1+1)%4=2 (2/4=0 with remainder 2)
(2+1)%4=3 (3/4=0 with remainder 3)
(3+1)%4=0 (the value returns to zero here because 4/4=1 with remainder 0)
(0+1)%4=1 (It starts over because 1/4=0 with remainder 1)

Table 1 - Numerical example of the mathematical operations per step

After updating the digit, the offset that was removed at the beginning is added back to represent the ASCII value of that character.

To ensure functionality, the code was compiled and uploaded to the board. After confirming the absence of errors, the serial monitor was opened and it was noted with satisfaction that the transmission was successful, delivering the correct message on time.



```
SECURITY_TEST2␊SECURITY_TEST3␊SECURITY_TEST4␊SECURITY_TEST5␊SECURITY_TEST6
ITY_TEST7␊SECURITY_TEST8␊SECURITY_TEST9␊SECURITY_TEST0␊SECURITY_TEST1␊SECURITY_T
EST2␊SECURITY_TEST3␊SECURITY_TEST4␊SECURITY_TEST5␊SECURITY_TEST6␊SECURITY_TEST7␊
SECURITY_TEST8␊SECURITY_TEST9␊SECURITY_TEST0␊SECURITY_TEST1␊SECURITY_TEST2␊SECUR
ITY_TEST3␊SECURITY_TEST4␊
```

Baud 115200 Port 4

Parity: [✓ None] Odd Even Mark Space

Data Bits: [✓ 8 bits] 7 bits 6 bits 5 bits

Stop Bits: [✓ 1 bit] 2 bits

Hardware Flow Control: [✓ None] RTS/CTS DTR/DSR RS485-rts

Software Flow Control: Receive Xon Char: 17 Transmit Xoff Char: 19

Winsock is: Raw [✓ Telnet]

Status: Connected, 🟨 RXD (2), TXD (3), CTS (8), DCD (1), DSR (6), Ring (9), BREAK, Error

You can use ActiveX automation to control me! | Char Count:690 | CPS:0 | Port: 4 115200 8N1 Non

Figure 13 - Test of the reception functionality

Satisfied with the results obtained, the receiving module is then implemented.

Message Receiving

After the design and successful testing of message transmission, attention now turns to describing and implementing message reception within the system.

As mentioned in the previous section, communication in mechatronic systems is fundamental. Just as sending messages is important, receiving them is equally crucial, as they may contain vital and essential information necessary for anomaly detection within our system and fundamental data required from the receiver to perform specific actions.

To implement this functionality, a function similar to the transmission has been developed within the UART_Comm module. This function internally calls the HAL receive function for a specific data buffer. Unlike transmission, which is synchronized with a timer, reception is asynchronous. Therefore upon the arrival of a new message, the Interrupt Service Routine (ISR) is immediately triggered to initiate the read and analysis process, ensuring full compliance with the established protocol.

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  UART_COM_Receive(abyRXBuf,sizeof(abyRXBuf));
}
```

Code block 4 - Implementation of HAL_UART_RxCpltCallback

Regarding security, it is important to carefully control incoming messages. Imagine if every message read was flagged as correct; what would happen if incorrect or harmful data were inserted into it? Since we have no control over incoming data, a potential hacker could read outgoing messages, impersonate a listener, understand our communication protocol, and respond on its behalf, taking control of the communication. It is necessary to at least partially mitigate this issue by checking if the received message is in the list of valid ones. To implement this check, I compare the data buffer byte by byte with my reference buffer. If the message is valid, I turn on a notification LED on the microcontroller board.

In terms of security, it is crucial to carefully control incoming messages. It is important to verify every received message for accuracy and safety to prevent potential security breaches caused by incorrect or harmful data. If incoming data is not controlled, there is a risk of interception by a hacker who could impersonate a listener, decode the communication protocol, and respond on its behalf, essentially taking control of the communication.
To address this issue, it is necessary to verify if the received message is among the list of valid ones. This verification involves comparing the data buffer byte by byte with a reference buffer. If the message is considered valid, the microcontroller board activates a notification LED to indicate its validity.

```
#ifdef CLEARTEXT_CONF
		for (uint16_t i = 0; (i < wdBufSize) && (RxMexCheck == true);
i++)
		{
			if (i < (wdRefMexLen-1))//check if the alphabetic part
is correct
			{
				if (abyRXBuf[i] == abyReferenceRxMex[i])
				{
					RxMexCheck = true;
				}
				else
				{
					RxMexCheck = false;
				}
			}
			else if (i == (wdRefMexLen-1) )//check if the numeric
part is correct
		   {
				if ((abyRXBuf[i] >= '0') && (abyRXBuf[i] <= '9'))
				{
					RxMexCheck = true;
				}
				else
				{
					RxMexCheck = false;
				}
		   }
			else
			{}
			pbyRXBuf=(pbyRXBuf-wdBufSize);

			if(RxMexCheck==true)//if the message is correct
			{
				HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5,
GPIO_PIN_SET);
				return UART_COM_OK;
			}
			else
			{
				HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5,
GPIO_PIN_RESET)
				return UART_COM_INVALID_RX;
			}

		}

#endif
```

Code block 5 - Implementation of the Transmission using cleartext configuration

To verify the functionality, the code was compiled and uploaded to the board. Upon successful compilation without errors, the serial monitor was opened to send the designated message from the dedicated interface. The correctness of the message was visually confirmed. Subsequently, various incorrect messages were sent for testing purposes:

1) SECURITY_TEST
2) SECURITY_TESTT
3) SECURITYTEST0

All tests have yielded positive results, indicating readiness for security analysis.

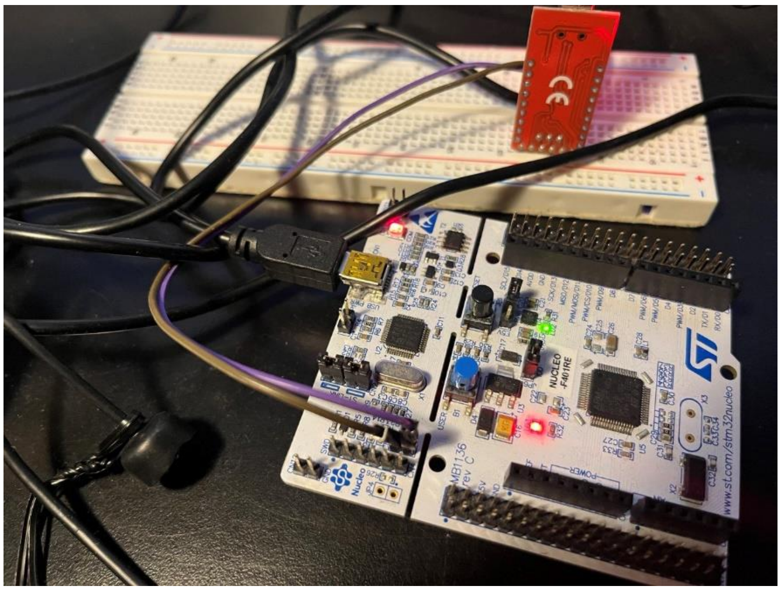


Figure 15 - Nucleo board when the message is correct

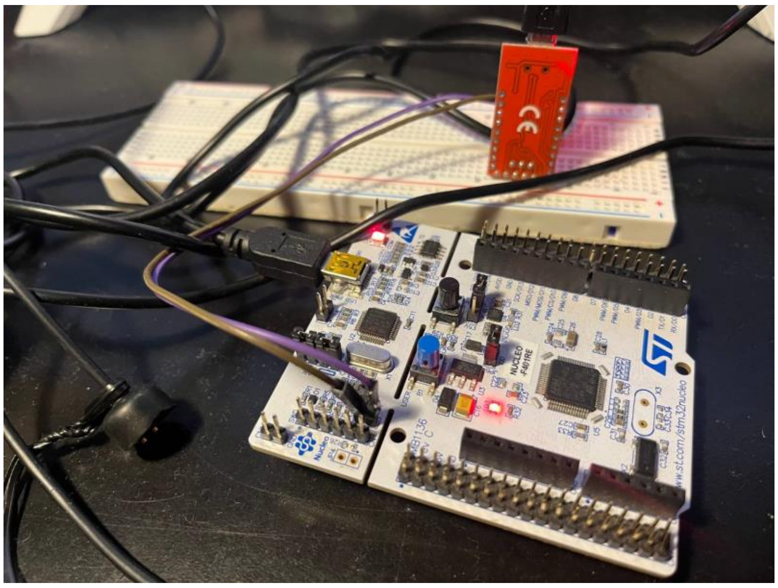


Figure 14 - Nucleo board when the message is wrong

Attack: Finding the Configuration

Now it is time to test the security of our system by simulating a hacker's attempt to breach it and evaluate its robustness and resistance to attacks.

As a premise, access to information is limited to the microcontroller's user manual [4]. This document explains in detail how the UART works and all the configuration parameters it can assume, along with pin mapping, where a potential wiretapping device could spy on external communications. With this information, the only way for a hacker to read messages exchanged between the microcontroller and the outside world is to try all possible configurations of the UART port and check if the readed message makes sense or not.

To configure the COM port, a special Python package was installed to handle the PC's serial port efficiently and simply. As the hacker, the selected USB port is the one that is connected to the adapter to read the messages. The installation process was straightforward, requiring only the execution of this line of code on the Python command line.

```
pip install pyserial
```

Code block 6 - Installation command of the pyserial package

To initialize all parameters, one can refer to available documentation found on the web. [5]

To complete this task, a Python script was developed to systematically test all feasible configurations of the microcontroller's UART port.
After initialization and settingof the port, the script evaluates all possible configurations of a UART port by iterating through different baud rates, byte sizes, and parity options. It initializes a serial connection, then loops through each combination of settings, reads data from the serial port, and checks if the received data is valid. As a criterion for determining message validity, each character was analyzed to determine its printability. If any character is found to be unprintable, the configuration is ignored. Conversely, if all characters are printable, the configuration is included in the list of potential configurations.

The following Python code outlines the algorithm employed for this purpose:

```
# Lists of typical UART configurations.
typ_baud_rate = [9600, 19200, 38400, 57600, 115200, 2400]
typ_byte_size = [serial.EIGHTBITS, serial.SEVENBITS]
typ_stop_bits = [serial.STOPBITS_ONE, serial.STOPBITS_ONE_POINT_FIVE,
serial.STOPBITS_TWO]
typ_parity = [serial.PARITY_NONE, serial.PARITY_ODD,
serial.PARITY_EVEN, serial.PARITY_MARK, serial.PARITY_SPACE]

try: # Initialize serial connection ser = serial.Serial() ser.port =
com_port ser.timeout = 10 count = 0 founded = []
    # Iterate through baud rates, byte sizes, and parity options for
i_baudrate in typ_baud_rate: ser.baudrate = i_baudrate for i_bytesize
in typ_byte_size: ser.bytesize = i_bytesize for i_parity in
typ_parity: ser.parity = i_parity ser.open()
count += 1 x = ser.read(ser.bytesize)
                # Check if received data is valid if len(x) != 0:
datavalid = True
for i_scanbit in x: if datavalid: if 32 > i_scanbit or i_scanbit >
126: datavalid = False
if datavalid: founded.append(FoundedConf(ser.baudrate, ser.parity,
ser.bytesize))
                else: datavalid = False ser.close()
    # Print founded configurations if len(founded) != 0:
print("Possible configurations of the UART port:") for i_founded in
founded: print("Number of bits:", i_founded.nbits, "Baudrate:",
i_founded.baudrate, "Parity:", i_founded.parity) else: print("No
possible configuration was found")

except serial.SerialException as e: print(f "Error opening COM port:
{e}")
```

Code block 7 - Python code of the configuration brute-force attack

Once the attack is complete, all possible configurations with their respective captured messages are displayed on the screen.

```
Possible configurations of the UART port:
Number of bits: 7 Baudrate: 19200 Parity: M
Number of bits: 8 Baudrate: 115200 Parity: N
Number of bits: 7 Baudrate: 115200 Parity: N
Number of bits: 7 Baudrate: 115200 Parity: O
Number of bits: 7 Baudrate: 115200 Parity: E
Number of bits: 7 Baudrate: 115200 Parity: M
Number of bits: 7 Baudrate: 115200 Parity: S
```

Code block 8 - Results of the attack

The one with the most information is likely the one actually used.

This comprehensive approach allows us to rigorously test the security of our communications system and assess its resilience to potential attacks.

Analysis of Timing and Computational Cost

After discovering that this method easily breaks the system, we proceeded to analyze the compromise time.
To do this, we used the time library included in Python. At the beginning of the code, we set the initial time and then printed the elapsed time. Based on our analysis, we can conclude that the compromise time in this workbench condition is:

$$T_{Compromise} = 257.24\ s$$

The value is relatively low, indicating that our system is highly susceptible to compromise. Anyone could wait for such a short period of time to hack into the system.

If the focus shifts on the computational cost, here is a breakdown of costs in the code:

Dominant Costs:

- Nested loops for configuration testing:
  1. Outermost loop iterates over baud rates (6 iterations): O(N1)
  2. Middle loop iterates over byte sizes (2 iterations): O(N2)
  3. Innermost loop iterates over parity settings (5 iterations): O(N3)
  4. Overall cost within these loops: O(N1 * N2 * N3)
- Reading and validating data within the innermost loop:
  1. Reading data: O(N1 * N2 * N3)
  2. Data validation loop: O(N1 * N2 * N3 * N4), where N4 is typically small (number of bytes read)

Other Costs:

1. Initialization and setup: O(5) for constant-time operations
2. Exception handling: O(1) if no exception occurs; O(E) for exception handling, where E depends on the specific exception
3. Printing results: O(1) for simple messages; O(2 + 2 * (N1 + N2 * N3)) in the worst case when all configurations are printed

$$\text{Worst-case theoretical time complexity: } O(N1 * N2 * N3 * N4)$$

In summary, the time complexity is heavily influenced by the number of valid configurations found, which affects nested loop iterations and printing, as well as the time required for serial port operations such as `ser.open()`, `ser.read()`, and `ser.close()`.

The conclusion is that the computational cost of the process is manageable on modern personal computers. However, this manageability also makes it vulnerable to potential attacks from enemies.

Therefore, there is an urgent need for significant enhancements in security measures to fortify against these emerging threats. This addition is necessary to ensure the system's robustness and integrity against potential malicious exploits.

Chapter 3: AES

Introduction

The UART serial communication system previously operated without any encryption or data protection, making messages vulnerable to interception and reading by potential attackers. Specifically, attackers could easily expose plaintext messages through brute force attacks on the UART port configuration, thus risking the secrecy of transmitted data.

In communication security, using cryptographic algorithms is vital for protecting sensitive data from unauthorized access. Among these algorithms, the Advanced Encryption Standard (AES) stands out for its well-known robustness and efficiency in preserving data confidentiality.

To enhance the security of the UART serial communication system, the AES algorithm was implemented for message encryption before transmission and decryption upon reception. AES proves to be an ideal choice for this purpose due to its ability to find a good balance between security and efficiency.

The implementation of AES involves several key phases:

1. **Initialization**: During this phase, a cryptographic key and any necessary initialization vectors for AES are generated. It is crucial to keep the key confidential, limiting its access solely to authorized parties within the system.

2. **Encryption:** When transmitting messages via UART communication, they undergo encryption using the AES algorithm and the designated key. The resulting ciphertext replaces the plaintext and is then transmitted through the channel.

3. **Decryption:** Upon receiving the ciphertext message, the intended recipient uses the same cryptographic key to decrypt it, thereby restoring the original plaintext. Ensuring that only authorized recipients possess access to the key is fundamental for maintaining communication security.

Integrating AES into the UART serial communication system is an effective measure to safeguard sensitive data from unauthorized access. This robust cryptographic algorithm ensures the confidentiality and integrity of transmitted messages, significantly enhancing the system's overall security.

AES Algorithm

AES-128/192/256 algorithm operates on plain data blocks of 128 bits and produces cipher data blocks of the same length. It utilizes cipher keys of 128/192/256 bits. The fundamental unit of AES operation is a two-dimensional array comprising 16 bytes, referred to as "states," with a mapping relation depicted in the figure below.
[6]

Input[0]	Input[4]	Input[8]	Input[12]
Input[1]	Input[5]	Input[9]	Input[13]
Input[2]	Input[6]	Input[10]	Input[14]
Input[3]	Input[7]	Input[11]	Input[15]

Input bytes

State(0,0)	State(0,1)	State(0,2)	State(0,3)
State(1,0)	State(1,1)	State(1,2)	State(1,3)
State(2,0)	State(2,1)	State(2,2)	State(2,3)
State(3,0)	State(3,1)	State(3,2)	State(3,3)

States bytes

Figure 16 - Status and Input arrays [6]

The encryption process involves five main stages: KeyExpansion, SubBytes, ShiftRows, MixColumns, and AddRoundKey.

KeyExpansion generates 11/13/15 round keys from the original cipher key, aligning with the structure of the 2-D array as states.

The AES encryption begins by performing an XOR operation, adding the input plain data blocks with the first round key. Subsequently, AES-128/192/256 encryption

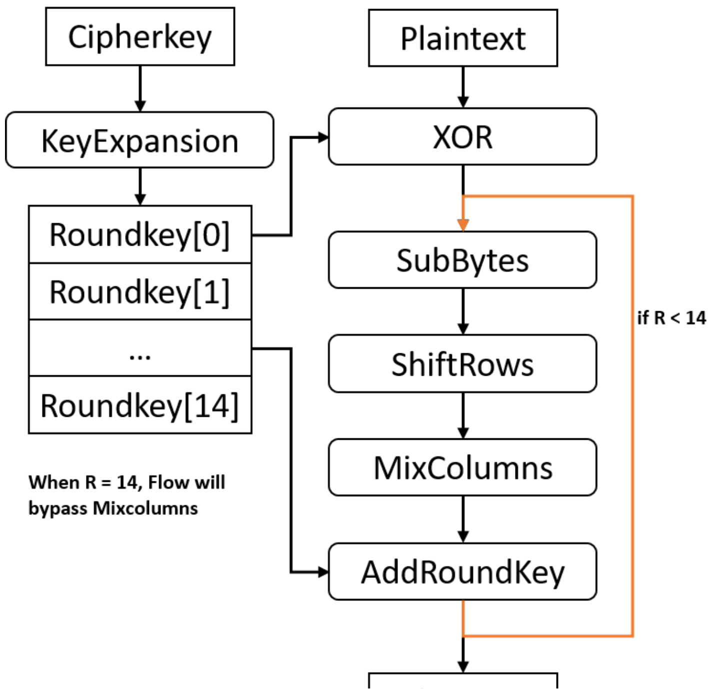


Figure 17 - Algorithm flux diagram [6]

executes 10/12/14 rounds of processing with the remaining round keys, one at a time. Each round sequentially executes SubBytes, ShiftRows, MixColumns, and AddRoundKey operations.

During SubBytes, each byte of the states undergoes transformation by utilizing a lookup table known as the S-box, using their respective values as addresses.

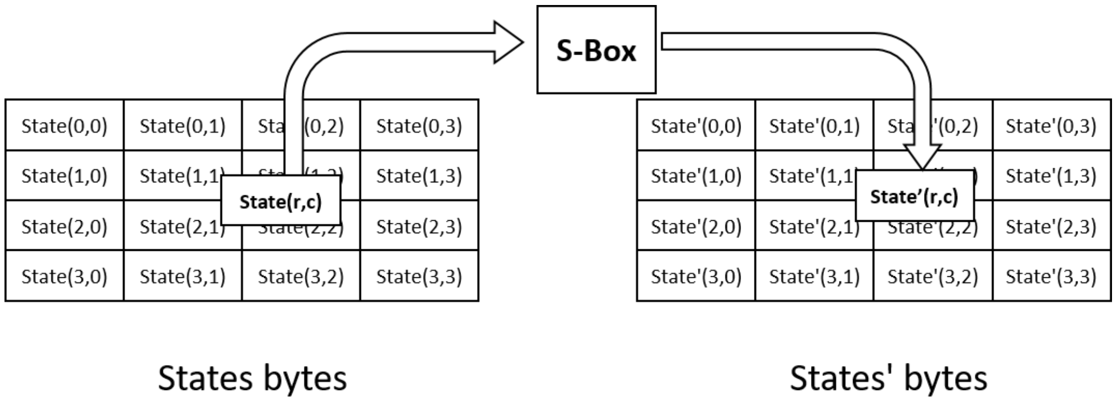


Figure 18 - SubBytes diagram [6]

ShiftRows involves cyclically shifting the last three rows of states over a different number of bytes based on the row number.

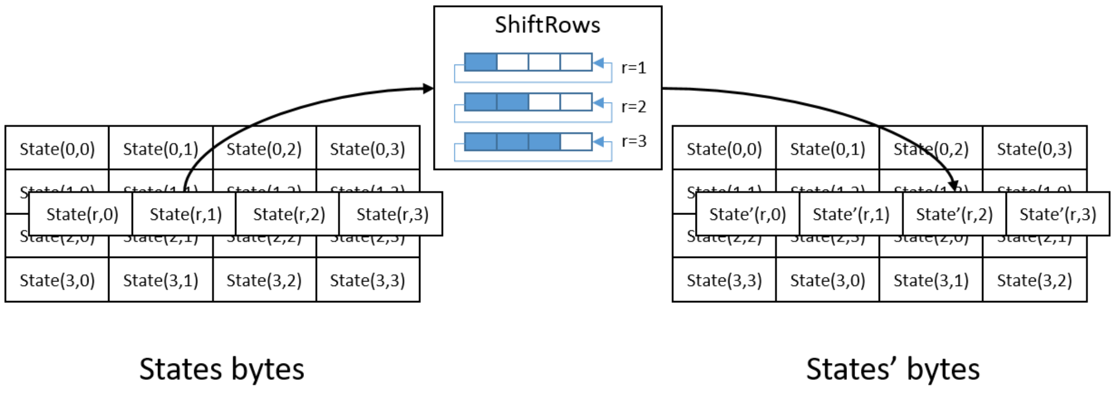

ShiftRows
r=1
r=2
r=3
State(r,0) State(r,1) State(r,2) State(r,3)
State'(r,0) State'(r,1) State'(r,2) State'(r,3)
States bytes
States' bytes

Figure 19 - ShiftRows diagram [6]

MixColumns operates on each column of the states. It utilizes matrix multiplication to transform each column, with the transformation matrix being fixed. The calculation treats each byte as a polynomial with coefficients in GF(2^8), modulo x^4 + 1.

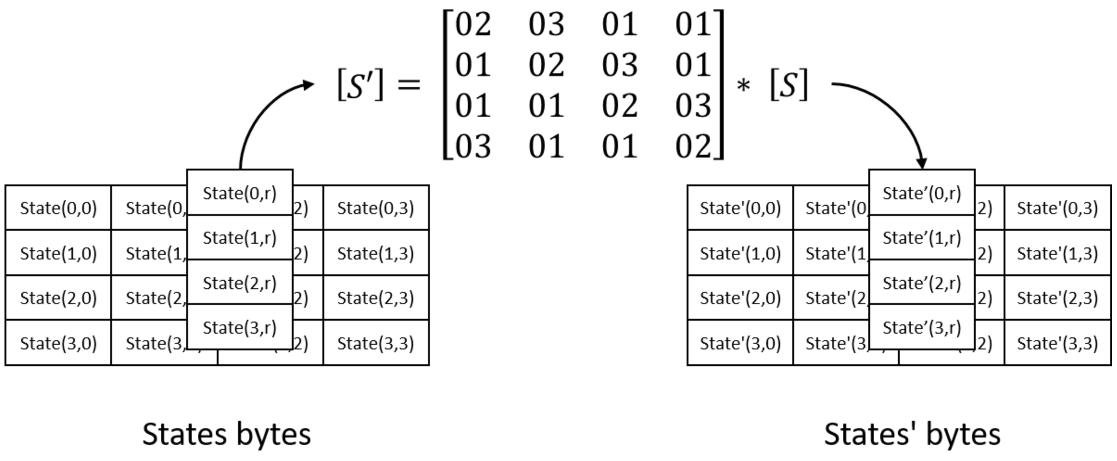


Figure 20 - MicColumns diagram [6]

AddRoundKey involves XORing the states with the round key of the current round.

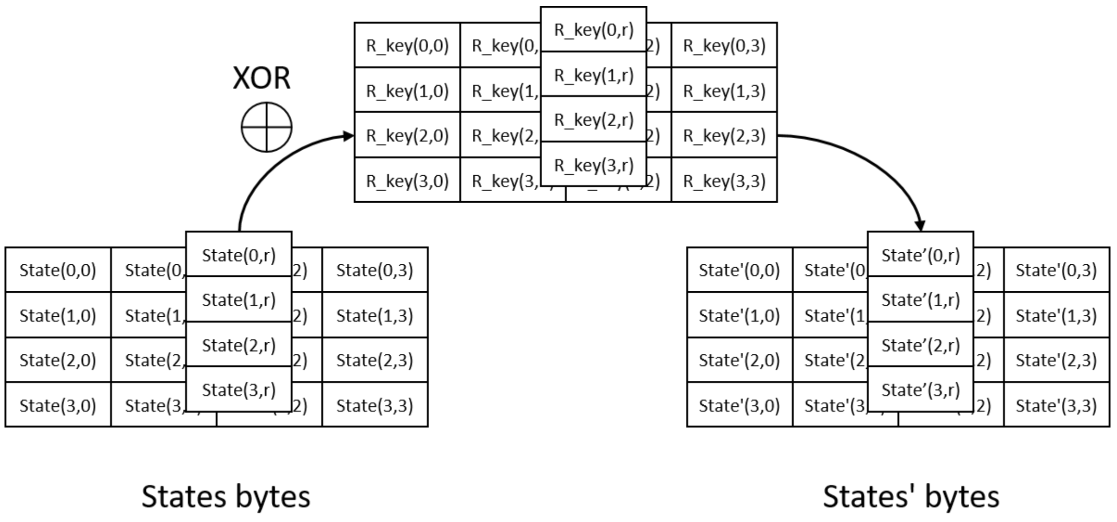


Figure 21 - AddRoundKey diagram [6]

AES Implementation

Now that the theory behind the algorithm is in hand, it's time to implement it on the system.

The microcontroller benefits from a library available on GitHub [https://github.com/kokke/tiny-AES-c?tab=readme-ov-file] [7]
 that offers a compact and portable implementation of the AES ECB, CTR, and CBC encryption algorithms in C.

In this scenario, for the sake of simplicity, the AES algorithm with a key size of 128 bits and 10 rounds of encryption is employed.

The encryption method used is ECB (Electronic Codebook). In this mode, each plaintext block is encrypted individually and autonomously, resulting in identical plaintext blocks being encrypted in the same way. This can potentially make the system vulnerable to pattern repetition in the ciphertext data.

In this case, there is no need to complicate the algorithm by using additional encryption methods. The algorithm is already strong and robust with just ECB and a minimal key size.

Focusing on the code, a new segment of code was incorporated into the dedicated function within the UART_Comm module for encrypting the data to be transmitted. This functionality can be activated using the AES_CONFIG macro.
The first step is to save the buffer designated for encryption into a dedicated buffer space. After initializing the algorithm with a randomly generated key, the message is encrypted using the appropriate function provided by the AES library.
Finally, the microcontroller's HAL function is used to send a message, which consists of two parts: the first half contains the encrypted message, and the second half contains the decryption key.

```
#ifdef AES_CONF

             //save data to encrypt

             for(uint16_t i=0;i<KEY_SIZE; i++)
             {
                    if(i<wdBufSize)
                    {
                           mx_encripted[i]=pbyTXBuf[i];
                    }
                    else
                    {
                           mx_encripted[i]=0;
                    }
             }
```

```
		AES_init_ctx(&ctx, abyKeyEn);

		AES_ECB_encrypt(&ctx, mx_encripted);

#ifdef TEST_AES_TX

		for(uint16_t i=0;i<KEY_SIZE; i++)
		{
			mx_decripted[i]=mx_encripted[i];
		}
		AES_ECB_decrypt(&ctx, mx_decrypted);
#endif
		//fill buffer to send
		for(uint16_t i=0;i<KEY_SIZE; i++)
		{
#ifdef TEST_AES_TX
			abyTXBuf[i] = mx_decripted[i];
#else
			abyTXBuf[i] = mx_encripted[i];
#endif
			abyTXBuf[i+KEY_SIZE] = abyKeyEn[i];
		}

		HAL_UART_Transmit_IT(pmyhuart,abyTXBuf,TX_BUF_SIZE);
#ifndef CLEAR_PSW
		for(uint8_t i=0; i<KEY_SIZE;i++)
		{
			abyKeyEn[i]=rand()%UINT8_MAX;
		}
#endif
		return UART_COM_OK;
#endif
```

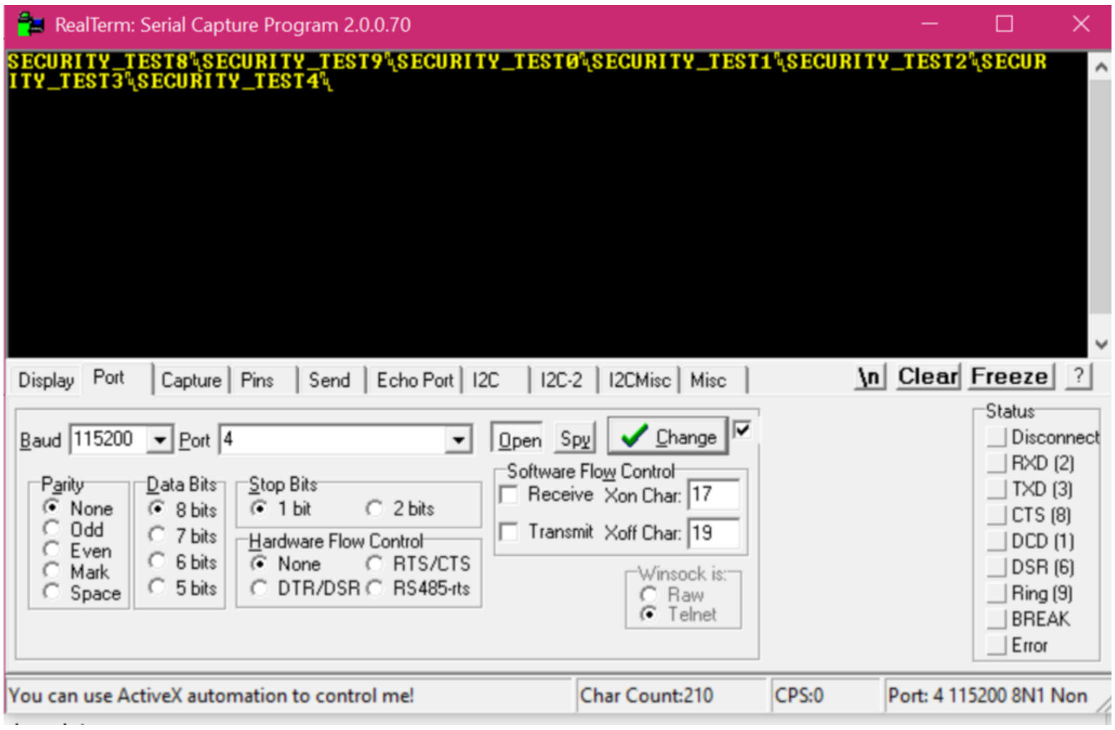
Code block 9 - Implementation of the AES transmission in UART_COM_Send

To test the algorithm's effectiveness, the corresponding macro is activated to enable the relevant section of the code. Upon activation, the encrypted message is decrypted using the same key, and verification is conducted via the serial monitor to ensure alignment between the decrypted message and the original message.

```
#ifdef AES_CONF

		//save data to encrypt

		for(uint16_t i=0;i<KEY_SIZE; i++)
		{
			if(i<wdBufSize)
			{
				mx_encripted[i]=pbyTXBuf[i];
			}
			else
			{
				mx_encripted[i]=0;
			}
		}

		AES_init_ctx(&ctx, abyKeyEn);
```

```
              AES_ECB_encrypt(&ctx, mx_encripted);

#ifdef TEST_AES_TX

              for(uint16_t i=0;i<KEY_SIZE; i++)
              {
                     mx_decripted[i]=mx_encripted[i];
              }
              AES_ECB_decrypt(&ctx, mx_decrypted);
#endif
              //fill buffer to send
              for(uint16_t i=0;i<KEY_SIZE; i++)
              {
#ifdef TEST_AES_TX
                     abyTXBuf[i] = mx_decripted[i];
#else
                     abyTXBuf[i] = mx_encripted[i];
#endif
                     abyTXBuf[i+KEY_SIZE] = abyKeyEn[i];
              }

              HAL_UART_Transmit_IT(pmyhuart,abyTXBuf,TX_BUF_SIZE);
#ifndef CLEAR_PSW
              for(uint8_t i=0; i<KEY_SIZE;i++)
              {
                     abyKeyEn[i]=rand()%UINT8_MAX;
              }
#endif
              return UART_COM_OK;
#endif
```

Code block 10 - Implementation of the test of AES transmission in UART_COM_Send


RealTerm: Serial Capture Program 2.0.0.70

```
SECURITY_TEST8␊SECURITY_TEST9␊SECURITY_TEST0␊SECURITY_TEST1␊SECURITY_TEST2␊SECUR
ITY_TEST3␊SECURITY_TEST4␊
```

Baud 115200 Port 4
Parity: None (selected); Data Bits: 8 bits (selected); Stop Bits: 1 bit (selected); Hardware Flow Control: None (selected)
Software Flow Control: Receive Xon Char: 17; Transmit Xoff Char: 19
Winsock is: Telnet (selected)

You can use ActiveX automation to control me! | Char Count:210 | CPS:0 | Port: 4 115200 8N1 Non

Code block 11 - Serial monitor of the AES transmission test

The results are satisfactory, so decryption will be implemented on the receiving module.

Similar to transmission, the received message and its key are stored in a specific buffer. Following this, the algorithm is initialized, and decryption is performed using the dedicated function.

```
#ifdef AES_CONF
		for(uint16_t i=0;i<KEY_SIZE; i++)
		{
			abyKeyDe[i] = pbyRXBuf[i+KEY_SIZE];
			mx_decripted[i] = pbyRXBuf[i];
		}

		AES_init_ctx(&ctx, abyKeyEn);

		AES_ECB_decrypt(&ctx, mx_decrypted);
	#ifdef TEST_AES_RX
		HAL_UART_Transmit_IT(pmyhuart,mx_decripted,KEY_SIZE);
	#endif
#endif
```
Code block 12 - Implementation of the AES reception in UART_COM_Receive

To verify the functionality of decryption, the decrypted message received is sent back to the sender, and its match with the original message is examined. This verification process is facilitated using the serial monitor, as done previously.
The message to send to the microcontroller is derived from the previously received messages.

Brute-Force Attack

To test the robustness of a new system, we will simulate a hacker and attempt to break it.

The attack strategy will begin with the Brute-Force method. This approach involves systematically testing every possible combination of characters for the encryption key, exhausting all potential permutations to uncover the original message.

For python there is a special library, called pycriptodome, in which there are everything we need to go and encrypt our messages.

After installing the package via terminal, it is imported into the project. Assuming that the correct configuration of the UART port has already been determined, the port is initialized with the parameters considered most likely, and communication is initiated. Since the length of the message is unknown, the most common configurations are tested: 64, 32 and 16 bytes. This allows for testing cases where a single message is sent followed by the key, as well as cases where a message consisting of half key and half data is sent. It is important to note that this is only an assumption about message formatting; the key and message could also be alternated, with one byte of the encrypted message followed by one byte of the key, and so on until the end. Another clever idea is to use a predefined mask or multiple masks to identify in advance which bytes correspond to the message and which to the key. Message formatting is flexible, and it is advisable to make it as complicated as possible in order to make it difficult for anyone to guess.

Regarding the possible characters that the password can assume, it was decided to limit ourselves only to alphanumeric characters in order to evaluate the effectiveness of the algorithm. These characters are divided into 10 numeric digits, 26 uppercase letters, and 26 lowercase letters.

Considering that the minimum key length is 16 characters, our program must evaluate a total of 62^{16} keys or even more if the size of key increase. This yields a staggering number: 4,767,525,381,634,189,126,649,103,360. This number is so immense that it surpasses the computing power of any existing computer in the world. The computational complexity involved in processing such a vast number of keys is beyond the capabilities of current technology. This underscores the robustness and security of AES encryption, as it poses an insurmountable challenge to brute-force attacks due to the astronomical number of possible key combinations.

A list is created with all these combinations. Each combination is then checked to decrypt the message and find text made only of letters and numbers. If found, the process stops, and the password is shown on the screen.

```
import serial
import time
from Crypto.Cipher import AES
import itertools

class FoundedConf: def __init__(self, baudrate, parity, nbits):
self.baudrate = baudrate self.parity = parity self.nbits = nbits


start_time = time.time()
com_port = 'COM5'
typ_baud_rate = [9600, 19200, 38400, 57600, 115200, 2400]
typ_byte_size = [serial.EIGHTBITS, serial.SEVENBITS]
typ_stop_bits = [serial.STOPBITS_ONE,
serial.STOPBITS_ONE_POINT_FIVE, serial.STOPBITS_TWO]
typ_parity = [serial.PARITY_NONE,
serial.PARITY_ODD,
serial.PARITY_EVEN,
serial.PARITY_MARK, serial.PARITY_SPACE] #5 operations at constant
time O(5)

ser = serial.Serial()
ser.port = com_port
ser.timeout = 10
ser.baudrate = typ_baud_rate[4]
ser.parity = typ_parity[0]
ser.bytesize = typ_byte_size[0]
ser.open()

 match = False
swapen = True
LenPack = [64, 32, 16]
i = 0

TestNumber = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
TestLowCase = ['a','b','c','d','e','f','g','h','i'
'l','m','n','o','p','q','r','t','u','v','x','z']
TestUpperCase = ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'T', 'U', 'V', 'X', 'Z']

if __name__ == "__main__":

    x = ser.read(LenPack[i]) if swapen:
msg = x[int(LenPack[i]/2):]
key = x[:int(LenPack[i]/2)] else:
msg = x[:int(LenPack[i]/2)]
key = x[int(LenPack[i]/2):] if i<len(LenPack): i=i+1 swapen = not
swapen
    PossibleCharacters = TestNumber+TestLowCase+TestUpperCase
combinations = list(itertools.permutations(PossibleCharacters,
len(key)))

    for i_KeyBF in combinations:
cipher = AES.new(i_KeyBF, AES.MODE_CBC, msg) decrypted_message =
cipher.decrypt(encrypted_message) unpadded_message =
unpad(decrypted_message, AES.block_size)
        for i_cryp_msg in unpadded_message: if 33 <= i_cryp_msg <
126:
match = True
else: if match is True and i_cryp_msg == 0: break
else:
```

```
match = False
break
    end_time = time.time() compromise_time = end_time - start_time
if not match: print("Nothing founded") else: print("Matched
Password:", key.decode("utf-8")) print("Compromise time:",
compromise_time, "seconds")
```

Code block 13 - Python code of the AES brute-force attack

As expected, the program needs to use a significant amount of computer RAM, exceeding the 16 GB capacity of this PC. Consequently, this leads to crashing the program and causing this error:

```
C:\Users\antop\PycharmProjects\Scripts\python.exe
C:\Users\antop\PycharmProjects\PythonProject\AES_BF.py
Traceback (most recent call last):
  File
"C:\Users\antop\PycharmProjects\pythonProject\AES_BF.py"
, line 68, in <module>.
    combinations =
list(itertools.permutations(PossibleCharacters,
len(key)))
MemoryError

Process finished with exit code 1
```

Code block 14 - error message of AES brute-force attack

Surprisingly, this can actually make the program more secure against brute-force attacks. Trying to guess the time it would take to crack the system becomes nearly impossible due to the added complexity.

Analysis of Timing and Computational Cost

Now focusing on the computational cost, the dominant factor in the code's computational complexity is the nested loop iterating over possible key combinations using itertools.permutations.
This loop has a time complexity of **O(n!)** where n is the length of the key.

Analyzing the code reveals that the overall cost of the program is composed as follows:

- Initialization (including FoundedConf class and variable assignments): O(1) - Constant time
- Serial port setup (opening ser): O(1) - Constant time (assuming negligible overhead)
- Reading data from serial port (single read call): O(1) - Constant time (for a single read)
- String manipulation (slicing for msg and key): O(n) - Linear time with respect to the length of the read data (n)
- Checking characters (inner loop): O(n) - Linear time with respect to the length of the decrypted message (n)
- Printing results: O(1) - Constant time

Moving on to the compromise time, the worst-case execution time for compromising the system is calculated under the assumption that the correct password is positioned at the end of the list of combinations. Since the program encountered an error due to the high computational cost, it was not possible to determine an experimental execution time. However, by conducting a theoretical calculation based on the available data, we can conclude that:

$$T_{compromise} = \frac{N}{f_{clock}} = \frac{62^{16}}{2.5\ Ghz} = \frac{62^{16}}{2.5 * 10^{9}} \approx\ 8.8 \times 10^{11} \quad \approx 27.86\ years.$$

This benchmark reassures us since no computer, even the most powerful, could wait that long to detect a single message password. It would certainly not be worth it. But if we want to be even more protected, we can see that the compromise time is proportional to the key length, so if the key length increases, the compromise time also increases. By introducing a key length of 32 bits, the $T_{compromise}$ becomes $3.504 \times 10^{34} years$, an astronomical number that could be approximated to infinity.

With this satisfactory result, the next step is the testing of another type of attack, characterized by greater intelligence and efficiency.

Dictionary Attack

The Dictionary Attack is a technique commonly used in computer hacking to uncover an account's password. It involves using a predefined list of words or character combinations to exploit the tendency of many individuals to use weak or easily guessable passwords. The fundamental principle underlying the Dictionary Attack is to systematically test each word or character combination in the dictionary against the target account until a matching password is discovered.

In implementing a Dictionary Attack, it is necessary to first compile a dictionary containing a wide range of common words, character combinations, names, and other possible passwords used by users. This dictionary can be created manually or downloaded from web as in our case. [8]

The files of the dictionary used in our experiment specifically contain about:

- 11000 Italian words with proper names
- 38000 Italian surnames
- 400 compound words
- 33500 verb conjugations

So many more could be added, but given the experimental purpose of the research it can be limited to these.

Once the dictionary is obtained, the attack process involves iterating through each entry in the dictionary and attempting to access the account with that password.

In our case we set as sample password:

Auto-assemblaggio

and write it on the bottom of the list, in order to test the maximum compromise time.

In this case, there is a specific package called pandas, which need to be installed. This package allows us to read from a txt file all the possible wors and save it as object.

The code for this attack is similar to the previous one, with the only difference being that the key used to decrypt the message is now in the dictionary instead of the list of possible combinations as before.

```
import serial
import pandas as pd
from Crypto.Cipher import AES

class FoundedConf: def __init__(self, baudrate, parity, nbits):
self.baudrate = baudrate self.parity = parity self.nbits = nbits

 dictionary = pd.read_csv("password_dictionary.txt",
names=['passwords'])

com_port = 'COM4'
typ_baud_rate = [2400, 9600, 19200, 38400, 115200, 57600]
typ_byte_size = [serial.SEVENBITS, serial.EIGHTBITS]
typ_stop_bits = [serial.STOPBITS_ONE,
serial.STOPBITS_ONE_POINT_FIVE, serial.STOPBITS_TWO]
typ_parity = [serial.PARITY_NONE,
serial.PARITY_ODD,
serial.PARITY_EVEN, serial.PARITY_MARK, serial.PARITY_SPACE]


 ser = serial.Serial()
ser.port = com_port
ser.timeout = 10
ser.baudrate = typ_baud_rate[4]
ser.parity = typ_parity[0]
ser.bytesize = typ_byte_size[1]
ser.open()

 valid = False
match = False
swapen = True
LenPack = [64, 32]
i = 0

while not valid:
valid = False x = ser.read(LenPack[i]) if swapen:
msg = x[int(LenPack[i]/2):]
key = x[:int(LenPack[i]/2)] else:
msg = x[:int(LenPack[i]/2)]
key = x[int(LenPack[i]/2):] if i<len(LenPack): i=i+1 swapen = not
swapen
    for i_KeyDICT in dictionary["passwords"]:
cipher = AES.new(i_KeyDICT, AES.MODE_CBC, msg) decrypted_message =
cipher.decrypt(encrypted_message) unpadded_message =
unpad(decrypted_message, AES.block_size)
        for i_cryp_msg in unpadded_message: if 33 <= i_cryp_msg <
126:
                match = True
else: if match is True and i_cryp_msg == 0: break
else:
match = False
break
    end_time = time.time() compromise_time = end_time - start_time
if not match: print("Nothing founded") else: print("Matched
Password:", key.decode("utf-8"))
```

Code block 15 - Python code of the AES dictionary attack

Analysis of Timing and Computational Cost

Considering a dictionary containing 490738 words and all possible combinations of segments from a message format are being explored, the estimated computational cost can be determined by evaluating the number of "for" loops required to assess all dictionary combinations.

Given the dictionary's 4,812,915 words and the examination of all possible combinations for each message segment, the total iterations can be calculated as follows:

$Let\ \ W\ \ be\ the\ number\ of\ words\ in\ the\ dictionary\ (4812915),$
$and\ (\ S\ )\ be\ the\ number\ of\ message\ segments\ (4).$

$The\ number\ of\ iterations\ for\ each\ message\ segment\ can\ be\ represented\ as$

$$Number\ of\ iterations\ for\ each\ message\ segment = W$$

$The\ total\ iterations\ can\ be\ calculated\ as:$
$Total\ iterations = Number\ of\ iterations\ for\ each\ message\ segment * Number\ of\ message\ segments = W \times S = 4812915 \times 4 = 19251660$

This results in approximately 19251660 iterations in the "for" loops. However, for a more precise computation cost, additional factors such as the complexity of the algorithm used for word comparison in the dictionary and the time required for other operations within the loop must also be considered.

In terms of compromise time, the worst-case execution time can be calculated when the correct password is located at the end of the list. Using this assumption, the tested time is determined to be:

$$T_{Compromise} = 257{,}24\ s$$

This is perfectly achievable by any type of computer even in a reasonable amount of time. However, this bad news informs us that the system can be breached in a short time, necessitating an increase in the level of security.

Chapter 4: RSA

Introduction

In response to the vulnerabilities identified within the AES system, an additional security layer has been integrated.
Unlike AES, RSA offers a unique approach to encryption by utilizing large prime numbers, which enhances its robustness.
This makes factoring impractical for most computing systems.
Its strength is also based on the different visibility of the keys:
In contrast to AES, where the encryption and decryption keys are the same and public, RSA operates asymmetrically by using a public-private key pair.
The public key is can be read from anyone and utilized for encrypting data transmitted to the system, while the private key remains confidential to the receiver, enabling decryption of incoming messages.

This architecture simplifies communication by transmitting only the encryption key, reducing a lot he risk since, if the key is compromised, decryption requires the corresponding private key.

While AES may be more immediately intuitive than RSA in terms of operational complexity, the latter offers substantially elevated security, making it an ideal choice for safeguarding sensitive data within our applications.

Furthermore, in this thesis, a comprehensive assessment of the upgraded system has been conducted, subjecting it to the same attacks previously applied to the AES system. Remarkably, all attempted attacks were unsuccessful, underscoring the robustness and effectiveness of the enhanced security measures in the system.

RSA Algorithm

To gain a better understanding of the RSA algorithm, a practical example can effectively illustrate its steps.
Consider the following scenario in which Bob wishes to send a message to Alice.
The following steps must be followed:

1. Key Generation:
- Alice selects two prime numbers, $p = 11$ and $q = 13$.
- She calculates the modulus, $n = p \times q = 143$, and the function $\phi(n) = (p-1) \times (q-1) = 120$.
- Alice chooses her RSA public key, $e = 7$, ensuring $1 < e < \phi(n)$ and the greatest common divider is $\gcd\big(e, \phi(n)\big) = 1$.
- Using the Extended Euclidean algorithm, she computes her RSA private key, $d = 103$.

2. Encryption by Bob:
- Bob obtains Alice's RSA public key (n, e), i.e., (143, 7).
- His plaintext message, M, is the number 9.
- Bob encrypts M into ciphertext, C, using the RSA encryption formula:

$$C \equiv M^e \, mod \, n$$

- So, $M^e mod \, n = 9^7 mod \, 143 = 48 \equiv C$.

3. Decryption by Alice:
- Alice receives Bob's ciphertext, C.
- She decrypts it using her RSA private key (d, n) with the formula:

$$M \equiv C^d \, mod \, n$$

- Thus, $C^d mod \, n = 48^{103} mod \, 143 = 9 \equiv M$

Consequently, Alice can reply to Bob following the same steps.

This encryption method can also guarantee the origin of a message.
Considering the situation where Alice, before encrypting the message with Bob's public key, encrypts it with her private key and then encrypts it again with Bob's public key. When Bob receives the message and decrypts it with his private key, he receives a message that is still encrypted. Decrypting this message requires Alice's public key, which confirms that the message was sent by Alice alone, since she alone has the private key that was used to encrypt the message in the first place.

In other words, by using this double encryption method, Alice can send messages to anyone while guaranteeing their origin. In fact, by encrypting the message with her private key, anyone can read it by decrypting it with her public key, thus ensuring that the sender is indeed Alice. [9]

RSA Implementation

The microcontroller being used lacks a dedicated cryptography library, necessitating the creation of a custom one. The library found at the provided link[https://github.com/kevhou/RSA/blob/master/RSA.c] [10] will serve as a starting point, which will then be modified to function on the microcontroller and interface with the UART communication library.

This provides a console application that uses the algorithm steps discussed in the previous section and integrates with the Windows terminal.

Examining the code, firstly, the two prime numbers and the public exponent are requested to generate the keys. Subsequently, the parameters are checked, and the appropriate function is called to perform the necessary mathematical operations to return both the public and private keys.
After entering the parameters, a printout is made of the generated keys and entered parameters.

For encryption and decryption, the user is only prompted to input the encrypted and decrypted messages, respectively. These are then processed by the appropriate function, which applies the correct mathematical formula to obtain the message in the desired configuration.

Our current task is to tailor this library to suit our needs. The changes to be made primarily concern the interface.
The main difference lies in the fact that the microcontroller lacks an interactive shell. The messages to encrypt and decrypt are data received from external sources through a dedicated function, and once received, they are allocated in memory.

The new library has been named MicroRSA and includes the following public functions:

```
MICRO_RSA MicroRSA_GenerateKeys(uint8_t p, uint8_t q, uint8_t e);

MICRO_RSA MicroRSA_Encrypt(uint8_t* abyEnBuf, size_t LenEnBuf);

MICRO_RSA MicroRSA_Decrypt(uint8_t* abyDeBuf, size_t LenDeBuf);

struct PubKey MicroRSA_GetPubKey(void);
```

Code block 16 - Prototypes of the public functions of MicroRSA module

The names are very self-explanatory: the first one takes the two prime numbers and the public exponent as parameters, checks them to ensure they meet the necessary criteria, generates the public and private keys, and saves them in two privately allocated structures. The encryption and decryption functions simply apply the formula to the buffer passed by reference. The last one have the only purpose to call the public key structure from other modules.
To send an encrypted message, the only action to do is enabling the RSA_CONF macro at the beginning of the UART_Comm.c file.

Let's focus for a moment on the implementation of functions.
The first function, named `MicroRSA_GenerateKeys`, has the purpose to generate both the public and private keys, which are used for encrypting and decrypting messages.

Initially, three parameters are passed: p, q, and e, representing two prime numbers and a public exponent, respectively. Firstly, the function verifies whether both numbers p and q are prime through the check_prime function. If both numbers are prime, the modulo of the public key N is calculated by multiplying the two prime numbers p and q.

Subsequently, the value of Euler's Phi (dwPhi) is calculated as the product of (p - 1) and (q - 1). This value is crucial for calculating the private exponent.

After checking the validity of the public exponent e using the check_e function, the public exponent is assigned to the public key EnKey.PubE, while the private exponent d is calculated using the modular inverse of e with respect to dwPhi.

Finally, if all conditions are met and the keys are successfully generated, the function returns MICRO_RSA_OK. Otherwise, if any of the checks fail, MICRO_RSA_PAR_ERR is returned, indicating an error in the input parameters.

```
// Function to generate RSA keys
MICRO_RSA MicroRSA_GenerateKeys(uint16_t p, uint16_t q, uint16_t e)
{
    uint16_t dwPhi;

    if ((check_prime(p) == true) && ((check_prime(q) == true)))
    {
        EnKey.PubN = p * q;
        DeKey.PrivN = EnKey.PubN;

        dwPhi = (p - 1) * (q - 1);

        if (check_e(e, dwPhi))
        {
            EnKey.PubE = e;
            DeKey.PrivD = mod_inverse(e, dwPhi);
            return MICRO_RSA_OK;
        }
        else
        {
            return MICRO_RSA_PAR_ERR;
        }
    }
    else
    {
        return MICRO_RSA_PAR_ERR;
    }
}
```

Code block 17 - Implementation of MicroRSA_GenerateKeys function

The second function is designed to encrypt data using the RSA algorithm. It takes as input a pointer to an array of unsigned 16-bit integers (uint16_t *abyEnBuf) and the size of the array (size_t LenEnBuf), representing the length of the data to be encrypted.

The function begins by checking if the pointer to the data array (abyEnBuf) is not NULL, ensuring that the data array is not empty. If the pointer is not NULL, the function proceeds through a for loop that iterates over all elements of the array.

For each element, the data array is encrypted using the public exponent and modulus of the public key (EnKey.PubE and EnKey.PubN) through the MEA function. The result of the encryption overwrites the data array itself.

Once all elements of the array have been successfully encrypted, the function returns MICRO_RSA_OK, indicating that the encryption operation has been completed successfully. If the pointer to the data array is NULL, the function returns MICRO_RSA_PAR_ERR, indicating an error in the input parameters.

In summary, This function accepts an array of data and encrypts it using the RSA algorithm with the specified public key. If the encryption operation succeeds, it returns a success signal; otherwise, it signals an error in the input parameters.

```
// Function to encrypt data using RSA
MICRO_RSA MicroRSA_Encrypt(uint16_t *abyEnBuf, size_t LenEnBuf)
{
    if (abyEnBuf != NULL)
    {
        for (uint16_t i = 0; i < LenEnBuf; i++)
        {
            abyEnBuf[i] = MEA(abyEnBuf[i], EnKey.PubE, EnKey.PubN);
        }
        return MICRO_RSA_OK;
    }
    else
    {
        return MICRO_RSA_PAR_ERR;
    }
}
```

Code block 18 - Implementation of MicroRSA_Encrypt function

The third function, MicroRSA_Decrypt, mirrors the structure of MicroRSA_Encrypt, with the primary distinction being its role in decrypting data. It accepts a pointer to an array of 8-bit unsigned integers (uint8_t *abyDeBuf) and the size of the array (size_t LenDeBuf), which denotes the length of the data to be decrypted.

Similar to MicroRSA_Encrypt, MicroRSA_Decrypt begins by validating the pointer to the data array (abyDeBuf) to ensure it is not NULL. If the pointer is valid, the function proceeds to iterate through each element of the array.

For each element, the function decrypts the data using the private exponent and modulus of the private key (DeKey.PrivD and DeKey.PrivN) via the MEA function. The decrypted result overwrites the original data array, preserving its structure.

Upon successful decryption of all elements, the function returns MICRO_RSA_OK, indicating that the decryption operation has been completed successfully. In the event that the pointer to the data array is NULL, MICRO_RSA_PAR_ERR is returned, signaling an error in the input parameters.

```
// Function to decrypt data using RSA
MICRO_RSA MicroRSA_Decrypt(uint8_t *abyDeBuf, size_t LenDeBuf)
{
    if (abyDeBuf != NULL)
    {
        for (uint8_t i = 0; i < LenDeBuf; i++)
        {
            abyDeBuf[i] = MEA(abyDeBuf[i], DeKey.PrivD, DeKey.PrivN);
        }
        return MICRO_RSA_OK;
    }
    else
    {
        return MICRO_RSA_PAR_ERR;
    }
}
```

Code block 19 - Implementation of MicroRSA_Decrypt function

The last function, MicroRSA_GetPubKey, simply returns a structure containing the modulus and the public exponent of the public key.

As done with the other configurations, a macro is created to activate the feature related to RSA transmission within the UART_Comm.c file. If the RSA_CONFIG macro is activated, the UART_Send function first saves the message to be sent in a dedicated buffer, intended solely to be modified for encryption. The MicroRSA_Encrypt function is then called, passing this buffer by reference along with its size.

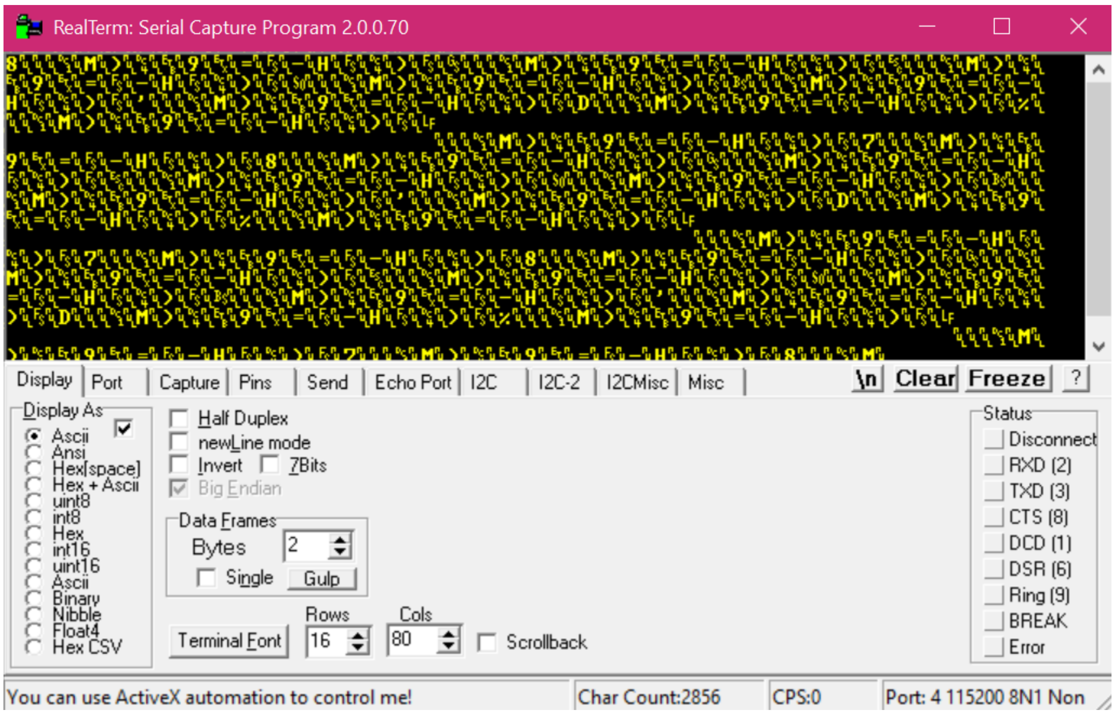
Now, the formatting of the output message needs to be decided. The simplest approach is to append the values of the public exponent and the modulus of the decryption key to the encrypted message. To implement this, a temporary buffer with two additional bytes is created, and it is filled with the encrypted message up to the second-to-last position. The value of the public exponent is then stored in the penultimate position, while the modulus is appended at the end.

Subsequently, the buffer is transmitted externally through the appropriate function of the operating system.

```
#ifdef RSA_CONF
		for(uint16_t i=0;i<KEY_SIZE; i++)
		{
			if(i<wdBufSize)
			{
				mx_encripted[i]=pbyTXBuf[i];
			}
			else
			{
				mx_encripted[i]=0;
			}
		}
		if(MicroRSA_Encrypt (mx_encripted, wdBufSize))
		{
			uint16_t rsa_mx_to_send[wdBufSize+2];
			for(uint16_t i=0;i<(wdBufSize+2); i++)
			{
				if(i<wdBufSize)
				{
					rsa_mx_to_send[i]=mx_encripted[i];
				}
				else if(i==wdBufSize)
				{

	rsa_mx_to_send[i]=MicroRSA_GetPubKey().PubE;
				}
				else
				{

	rsa_mx_to_send[i]=MicroRSA_GetPubKey().PubN;
				}
			}

	HAL_UART_Transmit_IT(pmyhuart,(uint8_t*)rsa_mx_to_send,sizeof(rsa_
mx_to_send));

			return UART_COM_OK;
		}
		else
		{
			return UART_COM_LL_ERROR;
		}
#endif
```

Code block 20 - Implementation of the RSA transmission in UART_COM_Send

To test if the function works, the message is first encrypted and then decrypted. The serial monitor is checked to confirm that it matches the original message.



```
Display As: Ascii
Half Duplex [ ]  newLine mode [ ]  Invert [ ]  7Bits [ ]  Big Endian [✓]
Data Frames  Bytes 2  Single [ ]
Rows 16  Cols 80  Scrollback [ ]
You can use ActiveX automation to control me! | Char Count:2856 | CPS:0 | Port: 4 115200 8N1 Non
```

Code block 21 - Serial monitor of the RSA transmission test

Satisfied with the achieved results, the implementation of the decryption at the receiving end goes smoothly.

As with transmission, the received message is stored in a dedicated buffer. However, there's no need to store the public key, as it's solely utilized for encryption. Therefore, when saving the vector, I stop two bytes before the end of the buffer.

Decryption is performed using a dedicated function, which applies the mathematical formula to the message using the internally allocated private key in memory.

```
#ifdef RSA_CONF

             HAL_UART_Receive_IT (pmyhuart,pbyRXBuf,wdBufSize);

             for(uint16_t i=0;i<(wdBufSize-2); i++)
             {
                    mx_decripted[i]=pbyRXBuf[i];
             }
             if(MicroRSA_Decrypt(mx_decripted,(wdBufSize-2)) ==
UART_COM_OK)
             {

       HAL_UART_Transmit_IT(pmyhuart,mx_decripted,(wdBufSize-2));
                    return UART_COM_OK;
             }
             else
             {
                    return MICRO_RSA_LL_ERROR;
             }

#endif
```

Code block 22- - Implementation of the RSA reception in UART_COM_Receive

In testing the decryption process, the decrypted message is resent to the sender for verification. This involves comparing the decrypted message with the original one to ensure they match.

Brute-Force Attack

Following the methodology used in the previous section to assess the robustness of an RSA system, a simulated attack will be conducted to evaluate its security.

The initial attack to be attempted is the Brute-Force attack, which entails systematically testing all conceivable combinations of the key in an attempt to uncover the original message.

Given the previous discovery of the port configuration and the use of RSA encryption for serial messages, the identical library used on the microcontroller is adapted to the Python code. This adaptation process is relatively straightforward, facilitated by Python's own shell, and requires only syntax changes compared to C.

Once the port is configured and opened, as long as the key is not found, the program receives the message and extracts the encoded message, the modulus, and the public exponent. However, since the decryption key consists of the modulus and the private exponent, the only value to be found is the latter.

Knowing that the exponent is a uint8_t data, with values ranging from 0 to 255, a for loop is instantiated to iterate over all these values. For each value, the program attempts to decrypt the message and analyzes it to see if it is composed of alphanumeric characters. If the key is found, the program prints the decryption key and the decrypted message on the screen.

```
while not valid:
    valid = False
    RxMsg = ser.read(LenPack[i])
    n = RxMsg[LenPack[i]-1]
    e = RxMsg[LenPack[i]-2]
    EnMsg = RxMsg[0:(LenPack[i]-2)]
    i=(i+1)%2
    values = range(256)
    # Find all combinations of n and e
    n=34
    for test_d in range(2**8):
        DeKey.PrivD = test_d
        DeKey.PrivN = n
        DeMsg = MicroRSA_Decrypt(EnMsg, DeKey)

        for i_cryp_msg in DeMsg:
            if 33 <= i_cryp_msg < 126: # printable range
                valid = True
            else:
                if valid is True and i_cryp_msg == 0:
                    break
                else:
                    valid = False
                    break
    end_time = time.time()
    compromise_time = end_time - start_time
    print("Compromise time:", compromise_time, "seconds")
    if valid:
            print("Decription key:","n  =", DeKey.PrivN,"d
=",DeKey.PrivD, )
            ascii_string = ''.join(chr(num) for num in DeMsg)
            print("Cleartext msg:", ascii_string)
            print("Compromise time:", compromise_time, "seconds")
            break
```

Code block 23 - Python code snippet of the RSA brute-force attack

Analysis of Timing and Computational Cost

Following the discovery that this method easily compromises the system, especially when using a data type with a limited range of representation, such as a byte, an analysis of compromise time was undertaken. To accomplish this, the time library included in Python was utilized.

The code sets the initial time at the beginning and then prints the elapsed time on screen.
On the basis of our experiment, our conclusion is that the worst-case scenario is that the compromise time is:

$$T_{Compromise} = 0.475\ s$$

The low value indicates that our system is highly susceptible to compromise. The short period suggests that anyone could exploit it to hack into the system. This vulnerability is due to the use of excessively small numbers. For example, transitioning to a number that can be represented on 16 bits results in an exponential increase in the compromise time. With only 20 bits, the compromise time extends to 195 seconds, significantly accelerating the system's vulnerability.

Typically, a large number of at least 617 digits is employed to ensure the required level of security. In this case, a small number is used merely as an illustrative example due to the limited computing power of the microcontroller. It is important to note that this encryption is only one layer of security among others.

The dominant factor influencing its computational cost arises from the nested loop structure. The inner loop iterates through all possible values of the private key exponent (d), resulting in exponential complexity (O(2^d)). This exhaustive search, while effective for small key sizes, becomes computationally infeasible as the key size (d) increases. The analysis assumes constant time for other operations within the loop, such as reading data and the decryption function (MicroRSA_Decrypt). However, the complexity of the decryption function, if not constant, would further contribute to the overall cost.

The conclusion is that to enhance protection, the RSA key must be sufficiently large to make computational cost and compromise time unachievable.

Dictionary Attack

In this scenario, the dictionary consists only of number combinations representing 'd', the range of which, of course, depends on the size of the chosen prime number. The level of security increases with the complexity of the 'd' value, making it more resistant to decryption attempts.
Therefore, a dictionary search becomes a meticulous brute-force attack strategy. The decryption process involves systematically iterating through a file to exhaustively test every possible combination of 'd'. The goal is to uncover the decrypted message while ensuring that it retains meaningful content despite the encryption. This exhaustive approach underscores the intensive computational effort required for decryption.

Taking a look at the code, the first step to take is to save us the encrypted message and the n form placed in the penultimate position. A for loop that runs through all the words in the dictionary changes the decryption key and tries to decrypt the message looking for a message consisting of only printable characters. When found, the loop stops and prints the decryption key and the decrypted message.

```
valid = False
RxMsg = ser.read(LenPack)
n = RxMsg[LenPack-1]
e = RxMsg[LenPack-2]
EnMsg = RxMsg[0:(LenPack-2)]
i=(i+1)%2
values = range(256)
# Find all combinations of n and e
combinations = itertools.combinations(values, 2)
for test_d in dictionary["passwords"]: #only numbers between 0 and
255
    DeKey.PrivD = test_d
    DeKey.PrivN = n
    DeMsg = MicroRSA_Decrypt(EnMsg, DeKey)
    for i_cryp_msg in DeMsg:
        if 33 <= i_cryp_msg < 126:
            valid = True
        else:
            if valid is True and i_cryp_msg == 0:
                break
            else:
                valid = False
                break
end_time = time.time()
compromise_time = end_time - start_time
if valid:
    print("Decription key:","n  =", DeKey.PrivN,"d =",DeKey.PrivD,
)
    ascii_string = ''.join(chr(num) for num in DeMsg)
    print("Cleartext msg:", ascii_string)
```

Code block 24 - Python code snippet of the dictionary attack on RSA system

Analysis of Timing and Computational Cost

The computational cost of this algorithm depends heavily on the size and complexity of the dictionary it uses. As the number of words in the dictionary grows, the algorithm needs to perform comparisons against each entry, leading to a significant increase in time and resources required. This complexity is typically measured in Big O notation, and for certain algorithms, it might be proportional to the product of dictionary size (n) and word length (w).

However, techniques like hashing can help mitigate this by improving search efficiency. Ultimately, considering this relationship between dictionary size, word length, and computational cost is essential for selecting the most suitable algorithm for a specific task. For instance, if real-time performance is a priority, a smaller dictionary with shorter words might be a better choice.

In our case, if the length of d is a byte, the computational cost is:

$$O(n) = 2^8 = 256$$

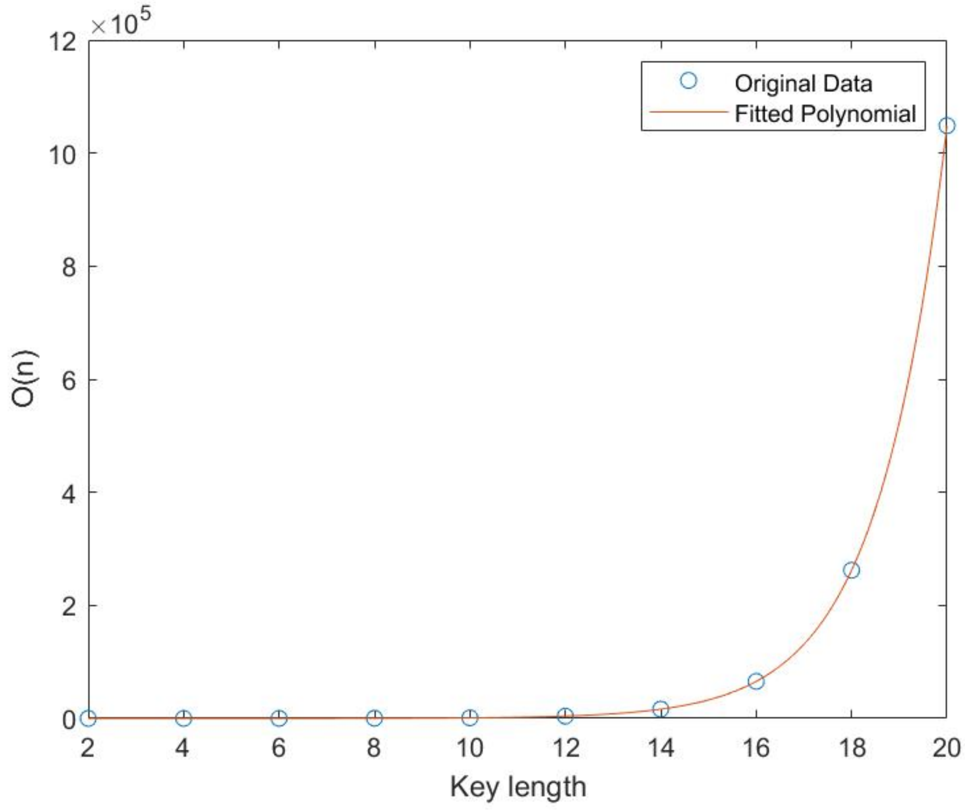


Figure 22 - Trend of computational cost respect to the key length

Analyzing the program's execution times when the for loop completes reveals an exponential trend. This is a critical observation in understanding how key length impacts security. An exponential increase in execution time signifies a dramatic rise in the effort required to crack the code. The attacks, which systematically try every possible key combination, become significantly slower and computationally expensive as the key length increases.

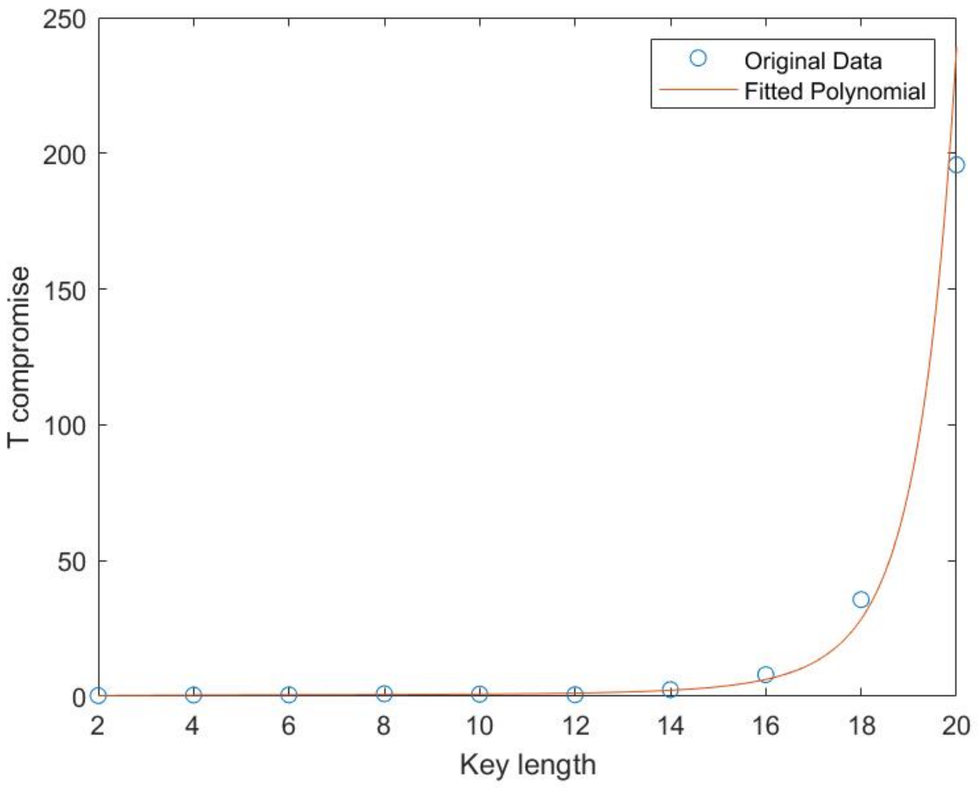


Figure 23 - Trend of compromise time respect to key length

For today's systems, RSA recommends using at least a 1024-bit key to maintain document security through 2010, and upgrading to a 2048-bit key for security through 2030. For documents requiring security beyond 2030, a 3072-bit key is recommended.

Conclusions

In conclusion, our research has highlighted the vulnerabilities present in serial communication for mechatronic systems and the critical need for enhanced security measures. The initial exploration revealed how easily the system could be compromised without any protection, underscoring the importance of implementing robust encryption algorithms.

The introduction of the AES encryption algorithm significantly improved the system's security, demonstrating resilience against brute force attacks and offering substantial protection. However, it showed some susceptibility to dictionary attacks, particularly when using commonly used keys.

Furthermore, the integration of the RSA algorithm provided an additional layer of security, especially against dictionary and brute force attacks when employing sufficiently large key sizes. Combining RSA and AES to create a double encryption of the message enhances the overall security posture of the system, making it significantly more challenging for malicious actors to decrypt intercepted messages.

Moreover, considering the lightweight nature of the software implementations, the potential applications of these security measures are vast and varied. From industrial automation to IoT devices, the ability to secure serial communication opens up a multitude of possibilities for safeguarding sensitive data and protecting critical systems.

In summary, by implementing advanced encryption algorithms such as AES and RSA and exploring the potential synergies between them, it is possible to significantly enhance the security of serial communication in mechatronic systems, ensuring the integrity and confidentiality of transmitted data across various applications and environments.

Bibliography

[1] Digital4, «Attacchi informatici in aumento nel 2023: l’Italia al centro dell’attenzione,» 14 November 2023. [Online]. Available: https://www.digital4.biz/pmi/cyber-security/attacchi-informatici-2023-italia-al-centro-dell-attenzione/.

[2] B. Kashyap, «Frame structure of UART protocol,» December 2020. [Online]. Available: https://www.researchgate.net/figure/Frame-structure-of-UART-protocol_fig1_348452220.

[3] codeurdudimanche, «No data signal on USART_Tx pin on my STM32-Nucleo,» 9 May 2023. [Online]. Available: https://stackoverflow.com/questions/76211058/no-data-signal-on-usart-tx-pin-on-my-stm32-nucleo.

[4] B. Kirstein, «RS232 9 Pin Pinout: Here’s What You Need to Know,» 20 July 2016. [Online].

[5] STMicroelectronics, «RM0368 User Manual,» 2018. [Online]. Available: https://www.st.com/en/microcontrollers-microprocessors/stm32f401/documentation.html.

[6] «Pyserial,» [Online]. Available: https://pythonhosted.org/pyserial/pyserial.html#overview.

[7] Xilinc Inc, «AES Encryption Algorithms,» 2019. [Online]. Available: https://xilinx.github.io/Vitis_Libraries/security/2019.2/guide_L1/internals/aes.html.

[8] kokke, "Tiny AES in C," [Online]. Available: https://github.com/kokke/tiny-AES-c?tab=readme-ov-file.

[9] napolux, «Liste di parole italiane,» [Online]. Available: https://github.com/napolux/paroleitaliane.

[10] A. F. Francesca Coppa, «ALGORITMO DI CIFRATURA RSA,» 2 July 2018. [Online]. Available: https://www.mat.uniroma1.it/sites/default/files/PLINIOSENIORE-CrittografiaRSA.pdf.

[11] K. Hou, «RSA Encryption/Decryption Program,» [Online]. Available: https://github.com/kevhou/RSA.