UNIVERSITÉ DE NANTES

RESEARCH PROJECT

MASTER 2 WIRELESS EMBEDDED TECHNOLOGIES (WET)

ANNEE UNIVERSITAIRE 2019/2020

COMPATIBILITY BETWEEN SECURITY AND RELIABILITY MECHANISMS IN IOT SYSTEMS

Bogdan GORELKIN

Institut d'Électronique et de Télécommunications de Rennes UMR CNRS 6164

Encadrants du projet :
- Sébastien PILLEMENT
- Maria MENDEZ REAL

Dates du stage du 03/02/2020 au 03/07/2020

Declaration of Authorship

I, Bogdan Gorelkin, declare that this thesis titled, «COMPATIBILITY BETWEEN SECURITY AND RELIABILITY MECHANISMS IN IOT SYSTEMS» and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date: 23/06/2020

Résumé

Un aspect important de la mise en œuvre matérielle des dispositifs cryptographiques est la protection contre les fuites d'informations via des canaux tiers. Du côté du canal de température, les recherches sont actuellement insuffisantes et il est nécessaire d'étudier la possibilité d'obtenir des données utiles sur la température du dispositif cryptographique. La protection contre les attaques par canal latéral est actuellement un domaine de recherche prioritaire dans le domaine de la sécurité de l'information.

L'objectif de ce travail est d'analyser les données obtenues à partir du capteur de température intégré lors du décryptage sur le microcontrôleur STM32F070Rb. Pour obtenir cet objectif, il faut effectuer les tâches suivantes:

1. Examiner les dangers/les points noirs de la sécurité.
2. Démonter le principe de l'algorithme RSA.
3. Faire la recherche sur des attentes
4. Implémentez une configuration expérimentale pour recevoir des données du capteur de température intégré du processeur STM32.
5. Analysez les résultats.

Auparavant, l'extraction de données utiles sur le canal de température n'était effectuée qu'à l'aide d'équipements supplémentaires. Dans ce travail, le capteur de température intégré du microcontrôleur STM32F070Rb est utilisé. Les résultats de ces travaux contribuent à l'étude de la sécurité RSA dans l'implémentation matérielle et peuvent affecter le choix des processeurs utilisés pour créer des appareils dans l'industrie de l'Internet des objets.

Les principaux résultats de la thèse ont été obtenus personnellement par l'auteur. La méthodologie de travail a été discutée avec le responsable scientifique de Sébastien PILLEMENT et Maria MENDEZ REAL, les aspects de la mise en place de l'installation expérimentale ont été discutés avec Safouane NOUBIR et Evgeny Rogozhnikov. La modélisation et le traitement des données ont été réalisés directement par l'auteur de cet ouvrage.

Contents

Introduction

Every day there are more and more devices with the ability to access the Internet, and therefore, according to Metcalfe's law (the usefulness of any system equals the square of the elements of this system), the Internet of Things is becoming more promising every day. The Internet of things is not just connecting billions of devices into one network, as the Internet once connected all computers. The real innovation and potential of the Internet of things is to transform business models, enable companies to sell products, bringing new benefits to both the company and the client. The scope of the Internet of things is illustrated in figure 1.

Currently, the Internet of Things (IoT) is changing dramatically depending on many factors in our world, on how we drive, how we make decisions, and even on how we get energy. The Internet of things consists of complex sensors, actuators, and microchips embedded in physical objects around us, making them smarter than ever. All these things are connected together and exchange huge data between them and with other digital components without any human intervention [2].

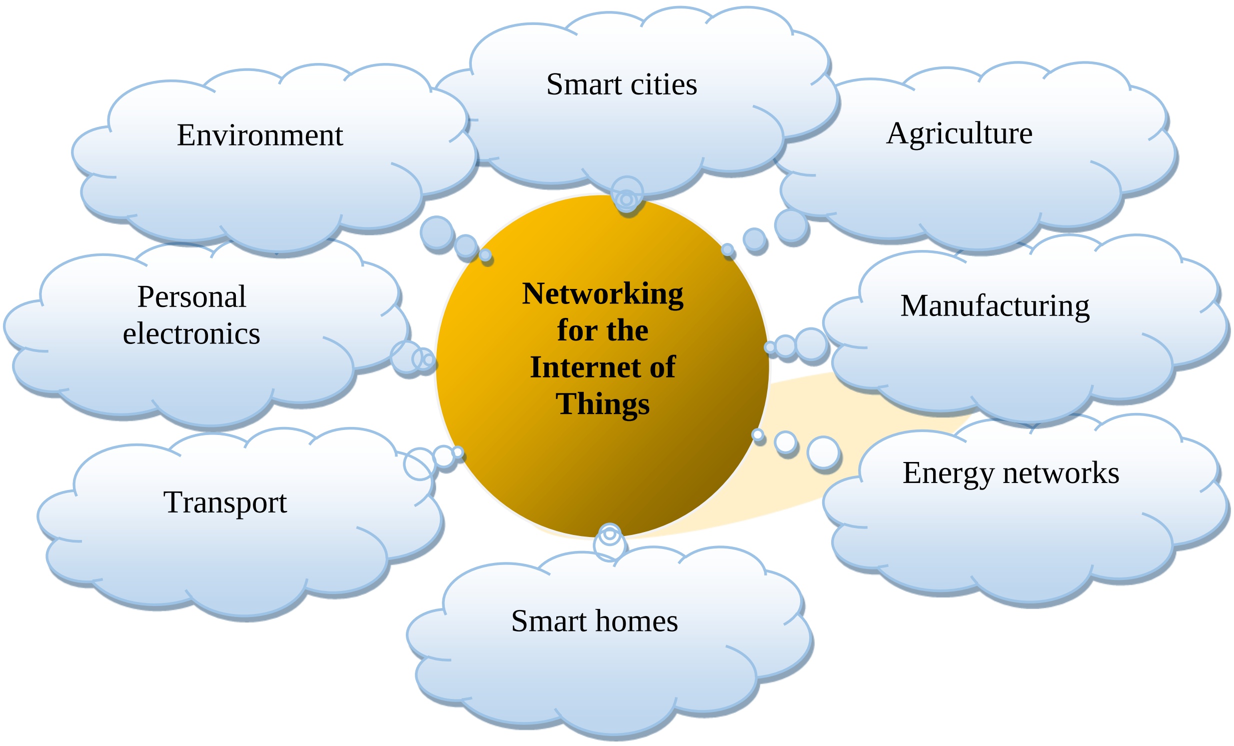


Figure 1 – Networking for the Internet of Things

Examples of using IoT are given below according to the order in the figure

1. Smart cities – Parking, urban transport, lighting, etc.
2. Agriculture – Monitoring environmental parameters, inventory management, process automation, etc.
3. Manufacturing – Monitoring and control of technical processes
4. Energy networks – Accounting and monitoring, management of intelligent networks
5. Smart homes – Fire / security alarm, etc.
6. Transport – Fleet management, cargo tracking, etc.
7. Personal electronics – Health monitoring, location monitoring systems, etc.
8. Environment – Monitoring of air condition, water bodies, etc.

Since every day the amount of data collected / transmitted increases - an important aspect is their security. When implementing encryption devices, it is important to consider vulnerabilities and prevent possible attacks on cryptosystems. In addition to the basic information from the encryption/decryption device, you can also get additional information received through third-party channels.

The aim of this work is to study the poorly studied side channel attack on the RSA cryptographic algorithm using data from the built-in temperature sensor during decryption on the STM32F070Rb microcontroller. To achieve this goal, you must complete the following tasks:

1. To consider threats to the security of the transmitted data.
2. Parse the operation of the RSA algorithm.
3. Explore the attacks through third-party channels.
4. Perform a RSA attack test through the temperature channel
5. Implement the experimental setup.
6. Analyze the results.

1. Security and confidentiality of transmitted / collected data

IoT makes a significant contribution to improving our daily lives over many applications in various sectors such as smart cities, smart construction, healthcare, smart networks, industrial manufacturing and much more. Nowadays, one of the problems potentially threatening Internet of Things devices is the security and confidentiality of transmitted / collected data, which are often deeply connected with people's lives.

Many organizations and enterprises are currently in the process of implementing security solutions to protect their IoT devices [2]. Today, security issues in IoT are more complex than existing security issues on our familiar Internet. It is important to note that IoT devices are often very limited in resources of computing power, memory and energy. These restrictions mean that many existing solutions that ensure the security and confidentiality of the transmitted data are absolutely not applicable.

To ensure the confidentiality (information becomes illegible with unauthorized access) [3] of data, messages and characters are encrypted. In general, encryption can be divided into two subgroups: symmetric and asymmetric. Examples of these mechanisms ensuring the confidentiality of the transmitted data are presented in table 1.1.

Table 1.1 – Privacy Algorithms

	Examples	Algorithm basis on
Symmetric cryptographic mechanisms	AES	the difficulty of solving certain types of equations[4].
	DES	on a combination of non-linear (S-blocks) and linear (permutations of E, IP, IP-1) converters.
	GOST block cipher (Magma)	Fiestel network [5]
Asymmetric mechanisms	RSA	the complexity of factorizing semisimple numbers
	Elgamal	the difficulties of computing discrete logarithms in a finite field
	DSA	computational complexity of taking logarithms in finite fields.

Cyber attacks on IoT systems create fears and hinder the development of the Internet of things, as they can steal confidential data, cause physical damage and even threaten people's lives. The complexity of these systems and the potential impact of cyberattacks pose new threats. Cryptography is one of the methods to ensure security.

When using symmetric ciphers, such as AES, which was used by the US National Security Agency to protect state secret information. Up to the SECRET level, keys with a length of 128 bits were allowed, for the TOP SECRET level, keys with a length of 192 and 256 bits[6], or GOST block cipher (Magma), (created by a group of KGB cryptographers of the 8th control) [7] were required.

The main problem arises: when transmitting a signal, a common encryption key is used for all devices. This means that if an attacker receives a key, then all devices using this type of encryption may be under his control. In order to organize a secure channel and securely transmit a symmetric key, asymmetric encryption methods, also called one-way, are often used.

2. Public key cryptosystems

Public key cryptosystems can be used for three purposes:

1. As independent means of protection of transmitted and stored data.
2. As a means for distributing keys.
3. Means of user authentication.

The most popular public key cryptosystem is the RSA algorithm, named by the first letters of their inventors' names - Rivest, Shamir, and Adleman [8] RSA cryptosystem - the first practical implementation of public key cryptography based on the concept of one-way function with a secret proposed by Diffie and Hellman [9,10].

The basic principle of RSA encryption is built on large primes () and mathematical operations on them. In asymmetric, unlike symmetric, not one, but a pair of keys is generated: private is used to decrypt messages by the addressee and remains secret, and the public is transmitted to the addressee. Key generation is performed according to the following algorithm [11,12].

1. **Two random large prime numbers are selected** p **and** q**.**

Numbers p and q remain secret and must be in the range from $(2^{2047}-1)$ to $(2^{2048}-1)$ or in binary representation: from *100....001* to *111….111* (2047 bits).

2. **Module of** p **and** q **of** $n=p*q$ **is calculated:**

n is the part of both public and private keys and it is communicated to the addressees through insecure channels. Decomposing of n into prime factors will decrypt the message. To date, the RSA - 232 cryptosystem has been factorized (02.17.2020) using the Lomonosov supercomputer of Lomonosov Moscow State University and the Zhores supercomputer [13].

3. **The Euler function is calculated.**

The Euler function shows the number of natural numbers coprime with

$n>\phi(n)=(p-1)*(q-1)$

The next step is to generate public key $\{e;n\}$. This key is communicated to the opponent via unprotected communication channels.

4. **An integer number** e**, prime with** $\phi(n)$**;** $e \in (1; \phi(n))$; is chosen.

It should be noted that to increase the encryption speed, Fermat numbers are selected more often [14].

5. **The private key** *{d;n}* **is generated. This key remains a secret and works as a decoder.**

Number *d* is selected, it should satisfy the following condition *d*e mod ɸ(n)=1;* For efficient memory operations the algorithm of fast exponentiation is used [14].

If there is a need to transmit not only numbers, but also text, then each character must be transmogrified into an integer representation. This is not difficult to procedure, for example, we can use the serial number in the alphabet, ASCII code characters [16] or in another way, but it is important to remember that the integer representation of the character should not exceed n. However, this will not be enough for the safe operation of the system, as the same character will be encrypted with the same number (Figure 3.1).

```c
#include <assert.h>
#include <stdio.h>
int main ()
{
  int c;
  int a;
  int d;
  while ((c = getchar ()) != EOF)
   {printf (" %3c%s --> ASCII -->", c, (c == "\n" ? '\n' : ""));
    printf (" %3d%s\n", c, (c == '\n' ? "\n" : ""));
    a = c+1;}
  return 0;
}
```

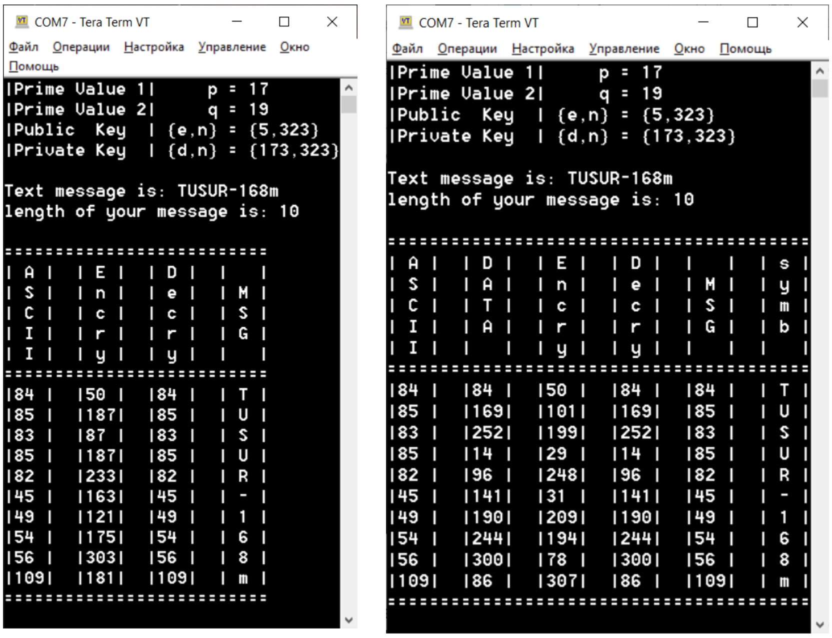
```
TUSUR
   T --> ASCII -->  84
   U --> ASCII -->  85
   S --> ASCII -->  83
   U --> ASCII -->  85
   R --> ASCII -->  82

 --> ASCII -->  10
```

Figure 3.1 – Receiving ASCII of symbols.

With a large selection of encrypted data, it will be possible to determine the original message without using keys (Figure 2a). To prevent this, it is necessary to use a reversible additional algorithm, where each block of encrypted data depends on the previous one, as in the Fibonacci sequence, or, for example, to add the previous one to each subsequent part of the transmitted message and find the remainder of the division by nn[17] (Figure 2b).

$$data[i] = (ASCII[i] + ASCII[i-1])\ mod\ (n).$$

To receive the original message, it is necessary to perform the reverse operation.



a) without block cipher б) with block cipher

Figure 3.2 – Encryption results

The encryption algorithm and decryption is carried out according to the formulas:

$$encrypt = data^{e} \bmod n;$$

$$decrypt = encrypt^{d} \bmod n;$$

When choosing the number sent to the input of the encryption device, one very important point must be taken into account. If the encrypted data is less than $\sqrt[e]{n}$, then it will not be difficult for an attacker to find the original message by finding it using a known public key:

$$data = \sqrt[e]{\text{encrypt}}$$

The main goal of this work is to analyze the operation of the RSA algorithm and to find the correlation of the decrypted data on the side channels. Because this is research work and there is no need to create a robust system for attacks, the block cipher will be omitted.

3. Side Channel Attacks

As many information security experts note, a strong cryptographic algorithm cannot automatically guarantee system security. In addition, the fact that every component of the system is protected does not necessarily mean that the entire system is safe.

Since the late 1990s, it has been known that the implementation of cryptographic algorithms is impossible without information leakage from different side channels. The first article that demonstrates the exploitation of the lateral canal was published by P. Kocher [18] in 1996. He emphasized that hardware implementations of cryptosystems can provide synchronization characteristics that leak private key information.

Side-channel attacks is a class of attacks on the cryptosystem that uses physical processes in the device and is aimed at the vulnerabilities of the cryptosystem in its practical implementation. As a result, cryptographic implementations must be evaluated for their resistance to such attacks, and various countermeasures must be taken into account. Side channel attacks are shown in the figure 4.1.

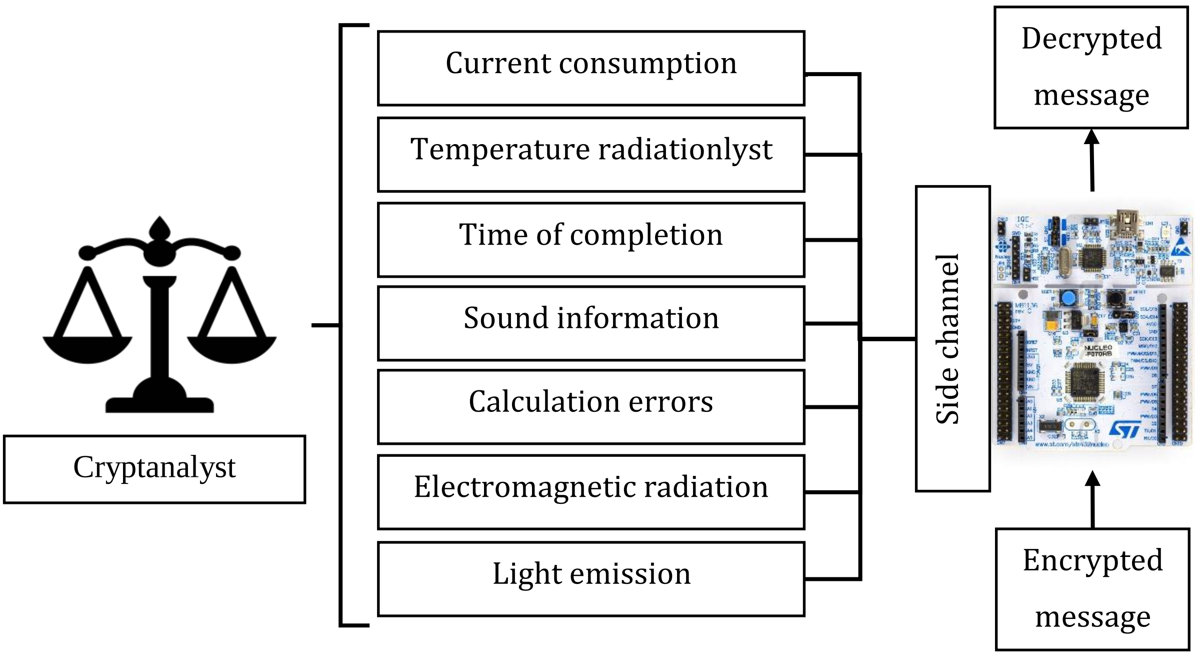


Figure 4.1 – Side channel attacks

Depending on the methods used in the analysis of the sampled data, all attacks can be divided into simple side channel attacks (SSCA) and differential side channel attacks (DSCA).

In SSCA, the attack uses the output of the side channel depending on the operations performed. The secret key in this case can be directly calculated from the side channel data trace. The side channel signal should be more relevant than noise [19].

On the other hand, when a simple attack on third-party channels is not feasible due to too much noise in the measurements, a differential side channel attack is used using statistical methods. Side channel differential attacks use the correlation between the data and the side channel output of the cryptographic device. Since this ratio is usually very small, it is necessary to use statistical methods to use it effectively. In a differential attack on the side channel, the attacker uses a hypothetical model of the attacking device, which is used to predict the output of the side channel of the device, and the quality of this model depends on the capabilities of the attacker.

At the moment, a fairly large number of attacks on side channels have been investigated. [20] So, the following types of attacks are distinguished through third-party channels:

1. Time Attacks

Based on statistical measurement results. [21]

2. Attacks on electromagnetic radiation (attacks using electromagnetic analysis)

This is a passive attack, consisting in the assessment of electromagnetic radiation.

3. Attacks on energy consumption (attack with power analysis)

a passive attack that allows you to measure energy consumption and obtain information about operations performed during the encryption process.

4. Attacks by apparent radiation (attack in visible light)

This is a passive attack based on recording and evaluating the intensity of light scattered from a monitor or encoders with light indicators.

5. Acoustic attacks (acoustic attack)

that study acoustic cryptanalysis and which are based on passive analysis of sound signals. [22] Acoustic side channels have been first introduced by A. Shamir and E. Tromer in 2004.

6. Attacks by calculation errors (“error-induction” attack)

This is an active type of side attack, based on the effect of influence on information, its active searches and analysis of results at different stages of work. This type of side attack can be carried out in various ways, such as changing power consumption, cryptosystems, encoder designs, voltage power systems, device clock frequencies, raising the temperature of some part of the encoder. The calculation can be related to the classification according to the type of errors received: they can be constant or variable, errors that require application at a certain time (for example, tactical frequency) and are independent of the moment of exposure.

A successful attack on cryptographic modules based on the calculation results involves two stages: implementation of errors and error handling. The first step is to enter an error. Clock, temperature, radiation, light, etc. The second step is to process the results. Work with a malfunction depends on the development and implementation of the software.

Thus, an attack on cryptographic modules through third-party channels uses characteristic information extracted

7. Temperature attacks (“error-induction” attack)

Although information leakage through temperature-based attacks is mentioned in modern literature, it has not been considered in sufficient detail.

Shown [23], that the cooling fan can transmit information about the processed data indirectly through a change in processor temperature. The experiment demonstrated how to extract bits from a possible RSA key (assuming a low-frequency bit leak, that is, a bit leak in three minutes). In addition, it is known that IP cores embedded in FPGAs can transmit information to other IP cores in the system through temperature readings.

Active temperature attacks that directly affect the temperature of the device (cooling or heating) are also investigated. Most of them demonstrate the effectiveness of low-temperature attacks, for example, as S. Skorobogatov reports [24] and D. Samyde et al. [25] in 2002.

They showed that by cooling SRAM devices to -50 ° C, they could freeze data and restore memory even a few seconds after turning off the power (using the SRAM cell data storage property). T. Muller et al. Used the same idea [26], who introduced a tool called FROST

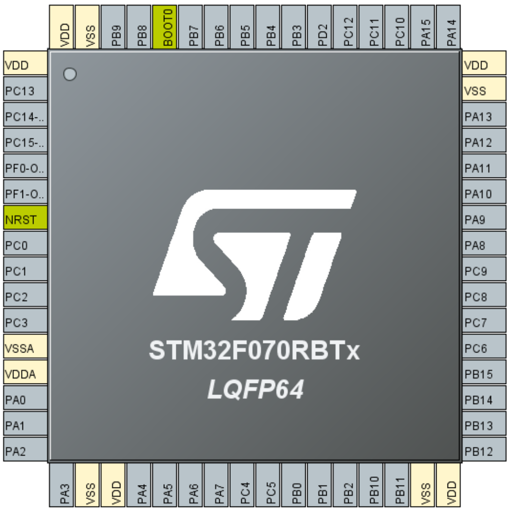
A temperature attack correlates with a power attack, but this does not make this type of attack simple. The temperature attack is quite difficult to analyze because temperature change is an inertial process that cannot change instantly. This means that with this type of attack it is important to consider a large number of factors. This type of attack requires further deeper study, which is what was done in this work. It is desirable to carry out an attack on the cryptosystem through the temperature channel together with other, additional third-party information. The most detailed attack using the temperature channel is described in [27].

This threat, in fact, refers to passive types of attacks, one of the most difficult to detect. Passive attacks - these are attacks in which Eve, a passive attacker[20], does not affect the cryptosystem, but only reads data from it. With such an attack, the cryptosystem continues its work without any changes [28].

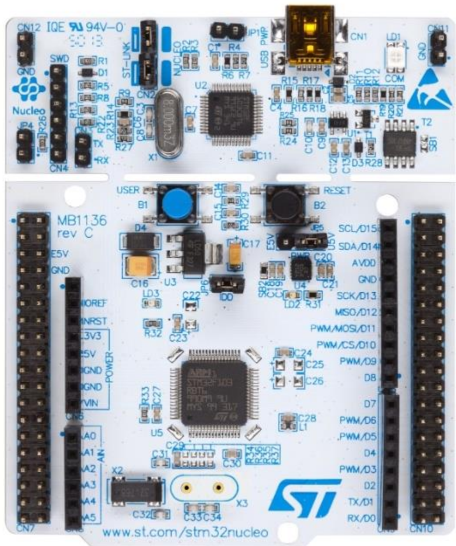
In order to counteract side channel attacks, the following methods can be used: shielding added noise in the side channel, balancing the execution time of the operations, leveling the power consumption system, masking and creating blindly calculated ones. [28]

4. The process of obtaining experimental data.

In the present study, the security of the RSA cryptosystem implemented on the STM 32 F 070 RB microcontroller (Figure 5.1-a) on the STM 32 Nucleo -64 development board (Figure 5.1-b) is checked. It also checks the possibility of extracting useful data from the readings of the built-in temperature sensor during decryption of various messages.



a) microcontroller



b) development board

Figure 5.1 - Equipment Used

For the purity of the experiment, the processor temperature is measured using a DMA controller (direct memory access controller). Direct memory access (DMA) is used in order to provide high-speed data transfer betweenperipherals and memory as well as memory to memory. Data can be quickly moved by DMA without any CPU actions. This keeps CPU resources free for other operations [29].

To obtain experimental data, the following tasks were implemented:

1. A public and private key has been calculated. The obtained values of the public key (e; n) were recorded in the code of the program that encrypted the message on the PC. Secret key values (d; n) - in the program code of the microcontroller that performed the message decryption (Appendix 1).
2. An encrypted message was received using the program (Appendix 2). Based on the received encrypted message, decryption was performed on the stm32 microcontroller.

3. The decryption of the same message was carried out sequentially many times to get enough ADC measurements for analysis from the internal temperature sensor of the microcontroller using direct memory access (DMA) (Appendix 3).
4. The temperature values using a USB cable via a UART connection were transferred to the virtual COM port of the computer for subsequent processing and finding a correlation between the decrypted message and the temperature of the stm32 processor. Instead of the usb cable, there can also be Bluetooth, Wi-Fi or any other module connected to the corresponding outputs of the microcontroller

Calculation of encryption and decryption keys on a PC.

The code of the program of preliminary calculations on the PC is presented in the appendix 1. The result of this algorithm is shown in Figure 5.2.

1. Entering numbers $p$and q is expected, they are checked for simplicity $p = 10007$ and $q = 399989$, were chosen such that their product n did not exceed $\sqrt[2]{\text{unsigned long long}}$ (n<*4294967296*). If you select n close to 18446744073709551615 (unsigned long long), then encryption and decryption can cause overflow of the variable and this will lead to an incorrect result of the algorithm.
2. n is calculated as well as the Euler function $\varphi(n)$.
3. A number e satisfying the conditions is selected:
   - $e < \varphi(n)$;
   - e is mutually simple with c $\varphi(n)$;
   - $e \in (1; \varphi(n))$.

   Number d is computed using the extended Euclidean algorithm[32] such that: $(d * e) mod\big(\varphi(n)\big)$=1

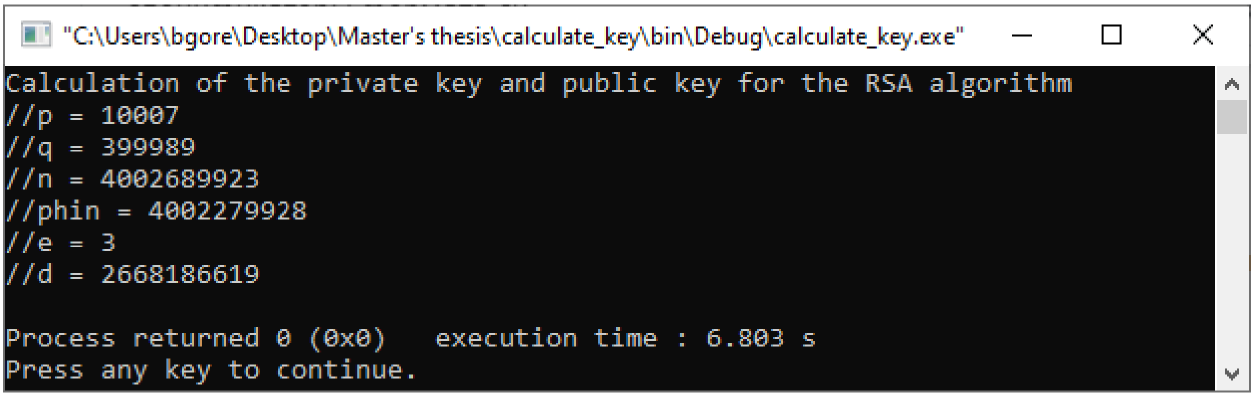


Figure 5.2 – search of keys

The maximum key size (n) cannot exceed 64bit, which in decimal notation corresponds to 18446744073709551615 (0xFFFFFFFFFFFFFFFF in hexadecimal). This limitation arises from working with integer values. If you continue to work with variables of type int, then you need to consider that when generating the private key, encryption and decryption, the remainder of the division by n is found, i.e. the size of the quotient should also not exceed the maximum size of the integer variable "unsigned long long".

Choosing the right key size for real cryptosystems is an important task. For this, it is necessary to know the estimates of the complexity of decomposition of primes of various lengths made by Shroppel[32].

Table 5.1 - estimates of the complexity of the decomposition of primes[32]

lg n	Number of operations	Notes
50	$1.4*10^{10}$	Disclosed on supercomputers
100	$2.3*10^{15}$	At the limit of modern technology
200	$1.2*10^{23}$	Beyond Modern Technology
400	$2.7*10^{34}$	Requires significant technology changes
800	$1.3*10^{51}$	Not disclosed

According to the recommendations of the authors of the RSA algorithm, the sizes of module n are selected according to the following criteria:

768 bits - for individuals;

1024 bits - for commercial information;

2048 bits - for classified information.

For an example, we will use received from February 17, 2020. p and q, in the decomposition of a semisimple number n - RSA-232 (768 bit) [13].

An integer type of variables is not suitable for working with values of such sizes. A solution that can help solve this problem can be implemented using arrays. Figure 5.3 shows the result of executing the program from Appendix 4, where two numbers are multiplied. This code works on the principle of column multiplication. If you continue to work with arrays further, then you need to implement the remaining necessary algorithms for mathematical operations, which in essence is a time-consuming task and is similar to writing your own library for long arithmetic.

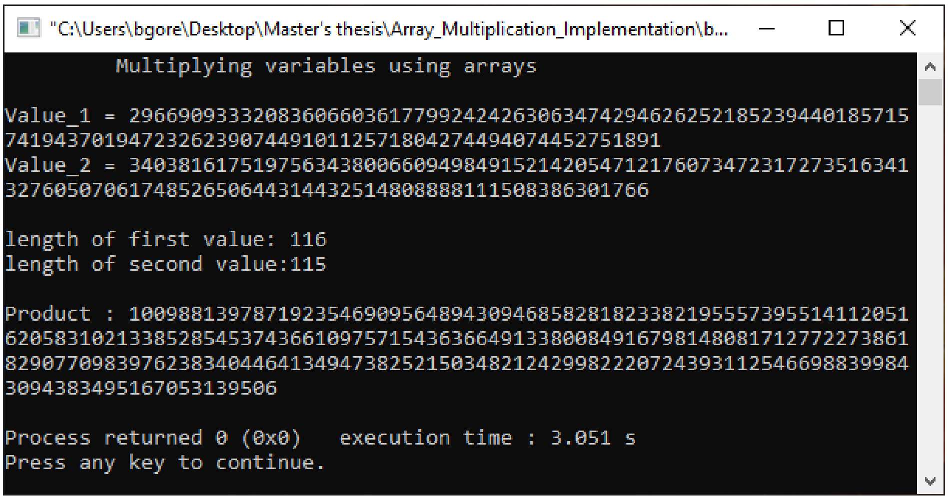


Figure 5.3 – Array multiplications

There are also alternative solutions to this problem. One option is to use off-the-shelf libraries like GMP, MPIR, or Mini-GMP. Another solution is to use modular arithmetic of a proprietary data type with a fixed size (int1024_t).

Note an important computational aspect of RSA implementation. For correct operation, you have to use a long arithmetic apparatus. If a key with a length of k bits is used, then $\mathbf{O(k^2)}$ operations are required for public key operations, $\mathbf{O(k^3)}$ operations for a private key, and $\mathbf{O(k^4)}$ operations are required to generate new keys. Compared to the same DES algorithm, RSA requires thousands and tens of thousands of times more time.

Receiving an encrypted message

In order to more likely see differences in temperature dependencies, it was decided to decrypt messages with different values. Also, to analyze the results, encrypted values were obtained for the messages “1024” and “133964360”, the encrypted values of which in the binary number system differ by 30 unit bits. Table 5.2 shows the messages and their corresponding encrypted values. For the effective operation of encryption - the function of raising to a power modulo is used.

Table 5.2 – Encrypted data

№	Data	Encrypted
1	2	8
2	3	27
3	100	1000000
4	1500	375000000
5	4002689921	4002689915
6	4002689920	4002689896
7	133964360	2147483647
8	1024	1073741824

Obtaining temperature data during decryption

To obtain a sufficient number of temperature measurements for analysis, the same data were decrypted sequentially (2500 times). Because DMA temperature reading is slower than decryption.

The DMA temperature readings were stored in the array during decryption, as soon as the last data was decrypted, the temperature reading stopped and the value of the built-in temperature sensor from the ADC was sent via UART to the computer's Com port.

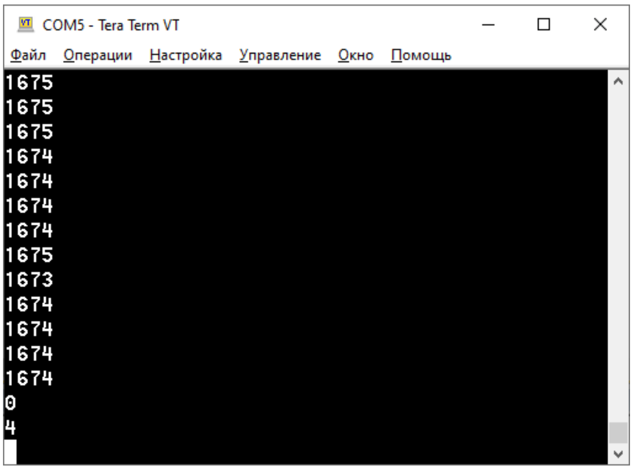


Figure 5.4 – Temperature Sensor Data

The data obtained is not difficult to translate into degrees Celsius using the following formulas[29]:

$$\text{Temperature (in °C)} = \frac{V_{30} - V_{SENSE}}{Avg\_Slope} + 30\ °C$$

$$V_{SENSE} = \frac{TS\_DATA}{4095} \times V_{DD}$$

5. Data analysis

Temperature leakage is linearly related to the power leakage model, but is limited by the physical properties of thermal conductivity and capacity.

The general scheme for obtaining experimental data is shown in Figure 6.1.

Deciphering different data takes different time to execute an algorithm. Thus, by the number of measurements obtained by DMA, it is possible to determine the operating time of the algorithm. Table 6.1 presents the characters from 0 to 9, their ASCII values, the result of their encryption, as well as the number of measurements obtained during decryption. Each character was also decrypted 2500 times. During decryption, values were obtained from the ADC of the temperature sensor for each symbol. The number of results for each symbol is different and constant for all experiments, because each character requires a different decryption time.

Table 6.1 – The number of measurements obtained for various ASCII characters

Symbol	ASCII	Encrypted	Measurements
0	48	110592	2211
1	49	117649	2198
2	50	125000	2222
3	51	132651	2213
4	52	140608	2207
5	53	148877	2202
6	54	157464	2204
7	55	166375	2229
8	56	175616	2219
9	57	185193	2223

By the number of measurements, it is possible to get data about quantity of the message decryption. When decrypting 1250 times the message with the symbol "1" and 1250 times the message with the symbol "2", the total number of measurements was 2210. For example, if we encrypt 2500 times the “1” symbol on the microcontroller, we can only get 168 measurements, because for encryption, an open exponent (e = 3) is used, in contrast to decryption (d = 2668186619). If the encrypted message is brought closer to n = 4002689923, then in the process of decryption the algorithm will follow the division instructions modulo and encrypting message MSG = 4002689920 2500 times, 175 measurements were obtained.

To obtain temperature measurements during decryption, the encrypted data was written in advance to the code of the executable program on stm32. Each data was decrypted separately and sequentially 2500 times. This was necessary in order to obtain about 2000 temperature measurements, because decryption of one message is faster than the RAP.

In order to see undistorted data, the readings were taken with the Keil uVision program closed, since when the debugger is on, the microcontroller consumes more current, and this directly affects the temperature of the processor. The decrypted message remained in the memory of Nucleo - STM32f070Rb. At the end of all UART decryption, the stored temperature data was sent using the RAP during the execution of the entire code. After that, the data obtained using auxiliary programs were analyzed. The general scheme for obtaining experimental data is shown in Figure 6.1.

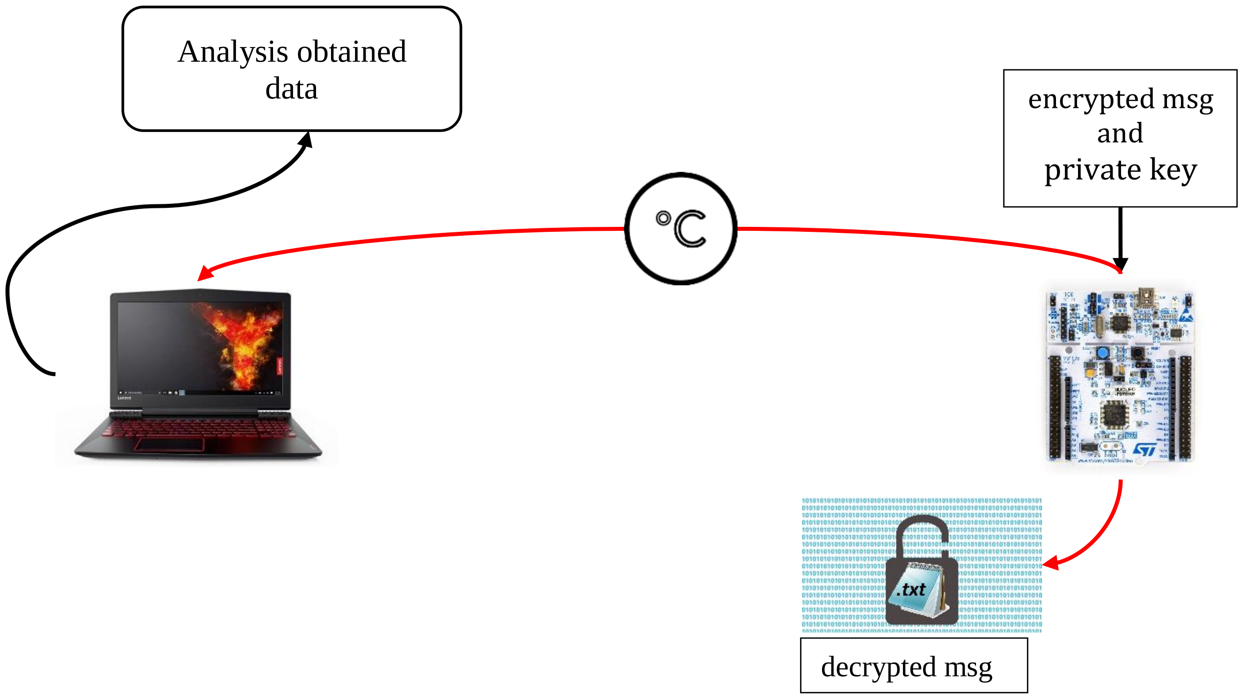


Figure 6.1 - Scheme for obtaining experimental data

The experiments were carried out in room conditions. The environmental conditions were obtained using a DHT11 sensor connected to the ATmega328P-AU microcontroller installed on the UNO R3 development board [Atmega 328P-

AU + CH340G] on a free computer COM port. The temperature in the room was 24.3 ° C (Figure 6.2)

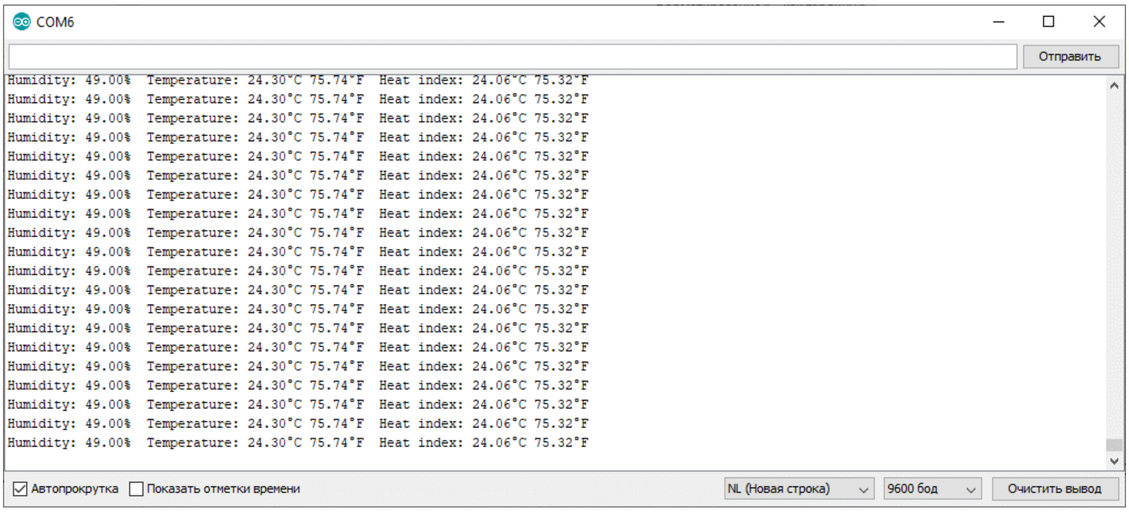


Figure 6.2 – Room temperature

Similar experiments were conducted to decrypt the data from table 5.1. Figure 6.2 shows the graphs of temperature changes for messages 1-4 from table 5.1. For each message, measurements were carried out 4 times and, based on the results obtained, the average temperature was constructed. The initial temperature for all experiments was artificially set at 44.38 ° C. The main difference between these graphs in the number of measurements obtained. For the “1500” message, it took more time to decrypt, and therefore more temperature measurements were obtained (2216).

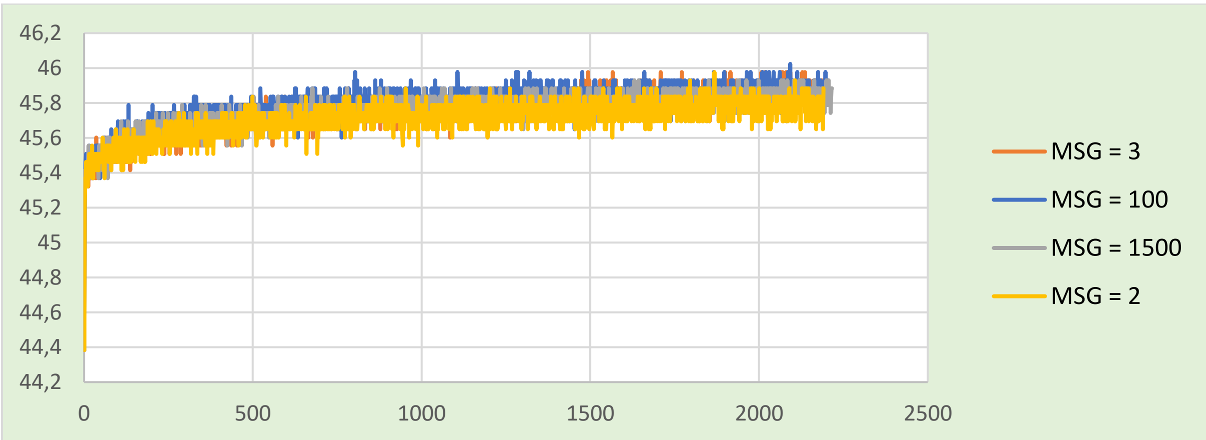


Figure 6.3 – CPU temperature for small messages

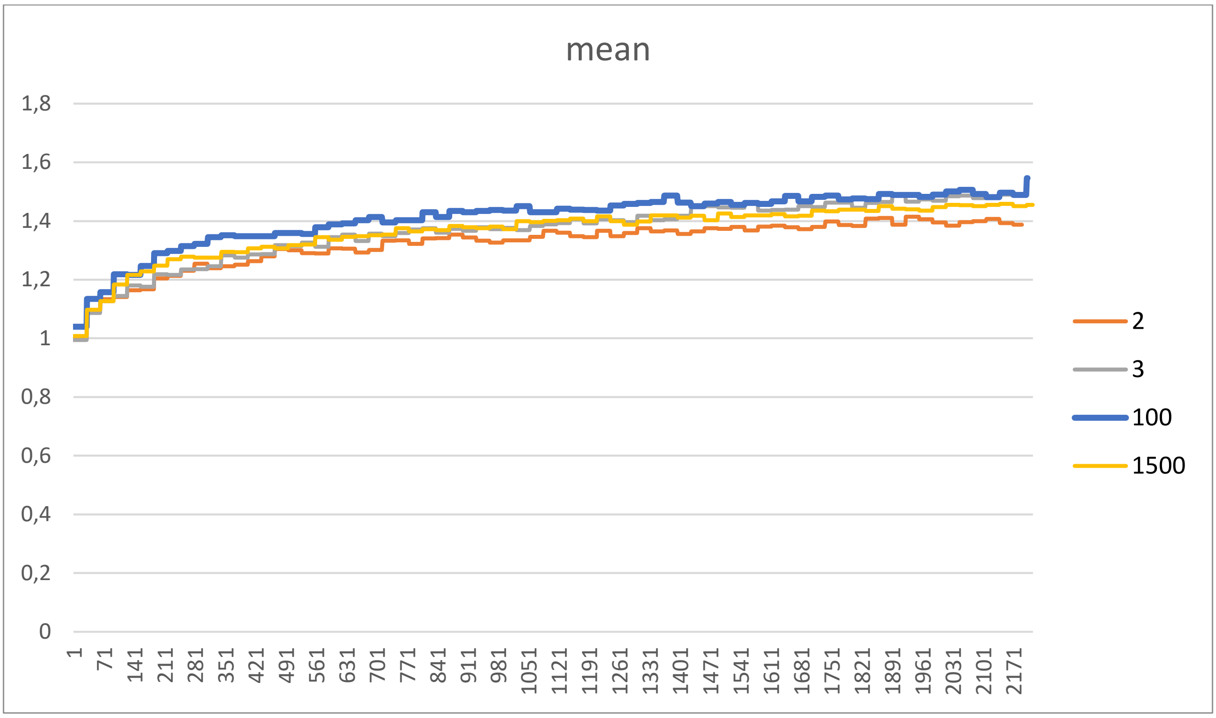


Figure 6.3 – Approximated CPU temperature for small messages

The initialization temperature for these measurements ranged from 44.5 to 44.3 ° C. This experiment was also performed for values 5 - 8 (Figure 6.4).

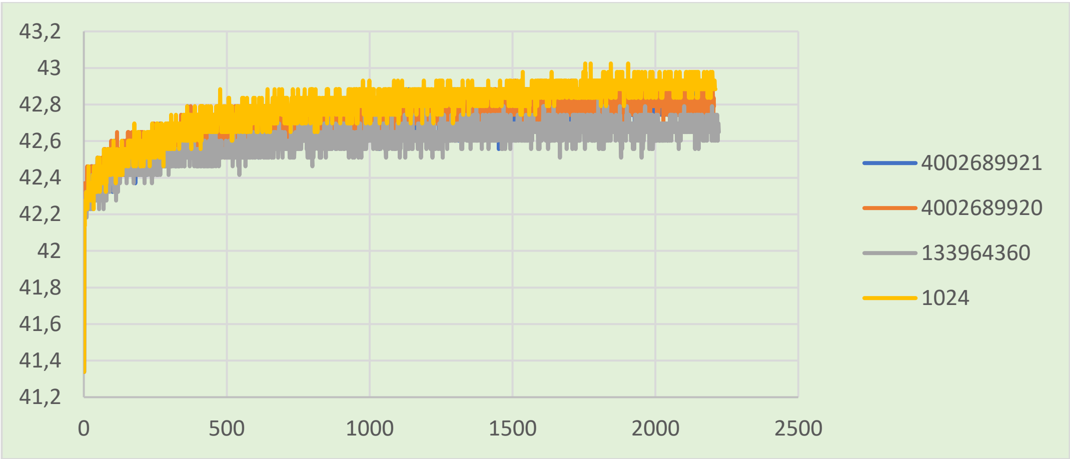
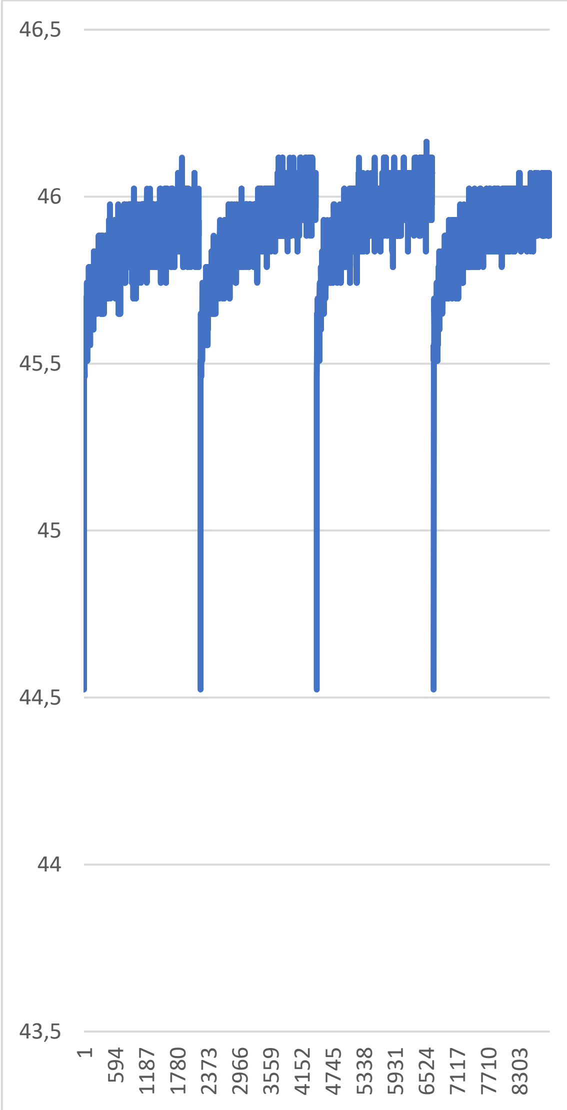


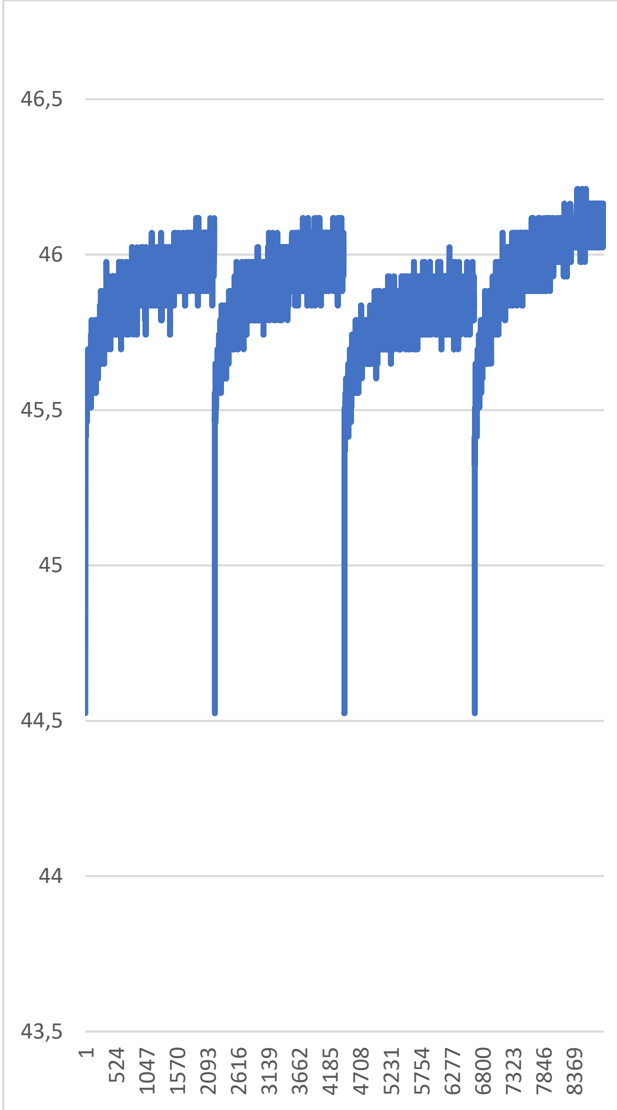
Figure 6.4 – CPU temperature for big messages

From the pictures of slow heating of the processor it can be seen that the temperature rises very slowly, by 1 ° C for the first 2 seconds, after which 0.05 ° C for every second. Based on the data obtained, we can conclude: an attack using a temperature side channel is possible in the event of a data leak for several milliseconds or seconds.

Figure 6.5 shows graphs of the temperature rise for each message. Each message was decrypted 2500 times. From this figure it can be seen that the processor temperature depends on which message is decrypted.



a



b

Figure 6.5 – Temperature rise for various messages

Figure 6.5 (a) presents graphs of temperature rise for messages № 1-4 of table 5.2 in Figure 6.5 (b) for messages № 5-8.

Also, this experiment was conducted for identical messages (table 6.2) but with different keys. Each message was decrypted 2500 times, and due to the different sizes of the keys, it was possible to obtain a different number of temperature measurements, since when decrypting, a different number of iterations is performed in the function of raising to a power modulo (Figure 6.6)

```
/* USER CODE BEGIN PFP */
int powMod(unsigned long long a, unsigned long long b, unsigned long long n)
 {
        long long x = 1, y = a;
        while (b > 0) {
                if (b % 2 == 1)
                        x = (x%n * y) % n;
                y = (y%n * y) % n; // (a*b) mod n = (a*(b mod n) mod n)
                b /= 2;
        }
        return x % n;
        /**<  MSG = powMod(Encrypted, d, n); */
}

/* USER CODE END PFP */
```

Figure 6.6 – Modular exponentiation

Table 6.2 – Message encryption with various keys

Experiment		1	2	3	4
p		11	103	4993	10007
q		13	107	4999	10009
n		143	11021	24960007	100160063
φ(n)		120	10812	24950016	100140048
e		7	5	5	5
d		103	4325	19960013	60084029
Symbol	ASCII	Encrypted data			
0	48	126	9469	5203898	54483842
1	49	36	7019	7915172	82155123
2	50	41	10566	12979916	12019811
3	51	116	1825	20545160	44545062
4	52	13	1574	5803927	79723843
5	53	92	3648	18835381	17555241
6	54	76	8122	9884898	58524772
7	55	55	10410	4084235	2484060
8	56	56	1385	1611622	49931461
9	57	73	562	2651889	731679

In the process of decrypting each message, temperature values were obtained for each message, with different parameters of the keys. Graphs of temperature changes are shown in figures 6.7-6.11. In order to display all graphs on the same level, the first measurement was taken as 0.

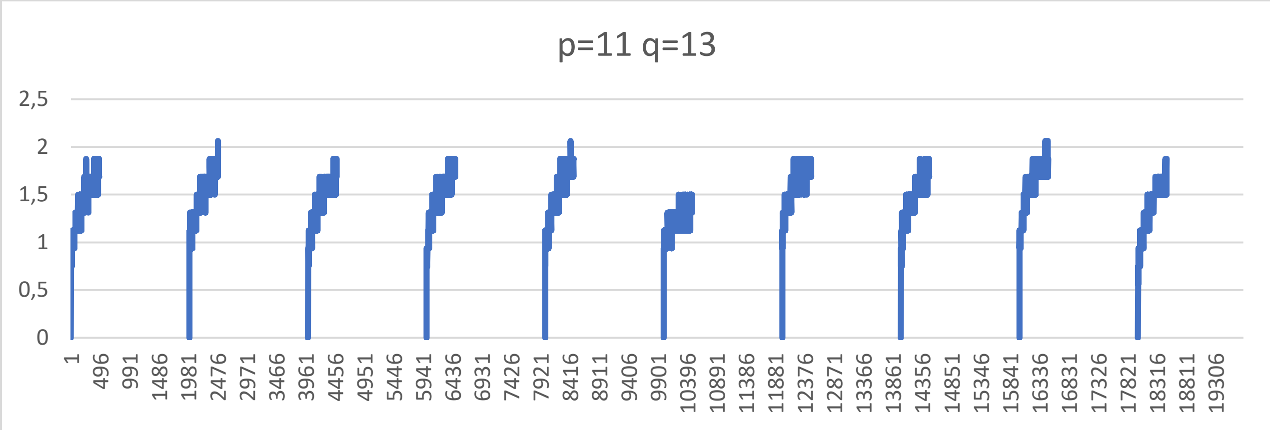


Figure 6.7 – Temperature dependence at p=11 q=13

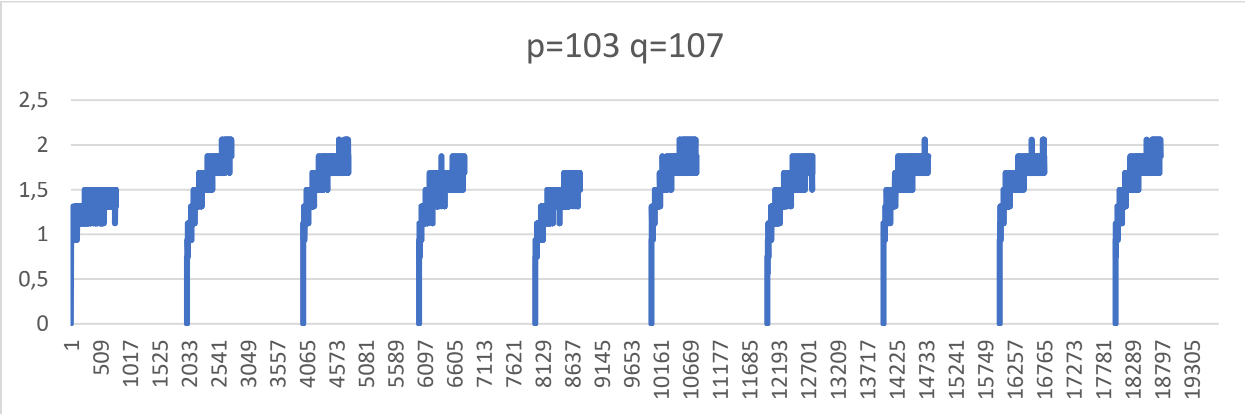


Figure 6.7 – Temperature dependence at p=103 q=107

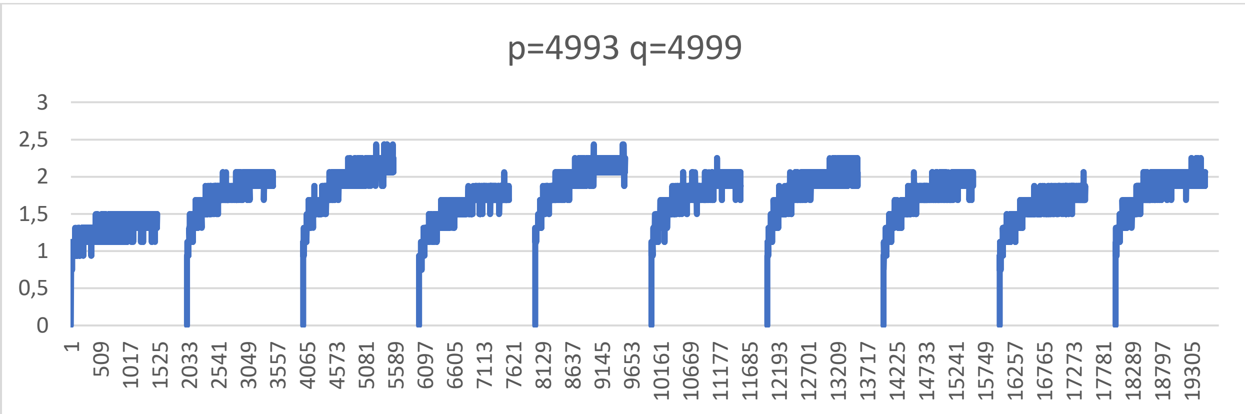


Figure 6.7 – Temperature dependence at p=4993 q=4999

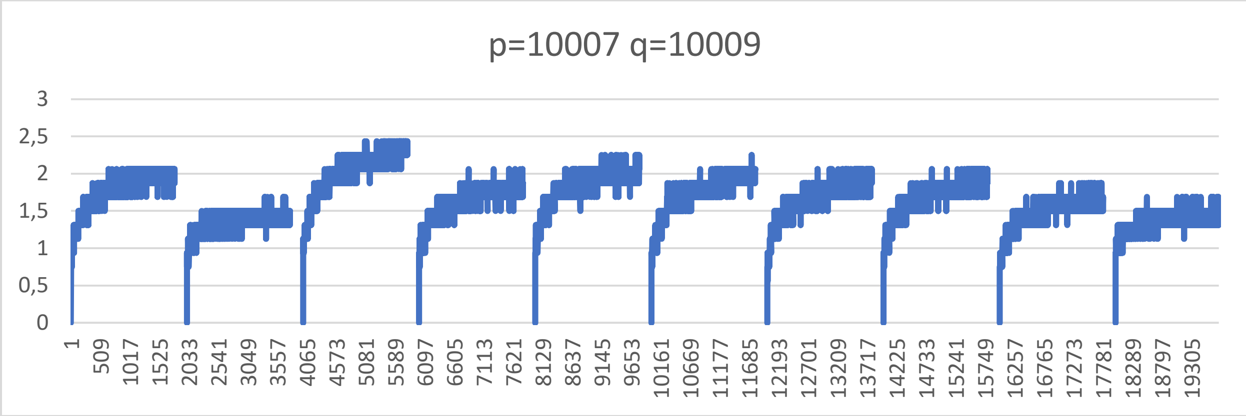


Figure 6.7 – Temperature dependence at p=10007 q=10009

Since the encrypted message was the same for all experiments, we can conclude that the processor temperature depends not only on the decrypted message, but also on the keys used. This is due to the fact that for messages in which the key size was larger, more time was required to execute the entire algorithm.

Current consumed by the microcircuit

Since the processor temperature directly depends on the current consumed by the microcircuit, it was decided to measure the current consumed by the processor. In standby mode, the microcircuit consumes less current (figure 6.6), in contrast to the consumption during program code execution

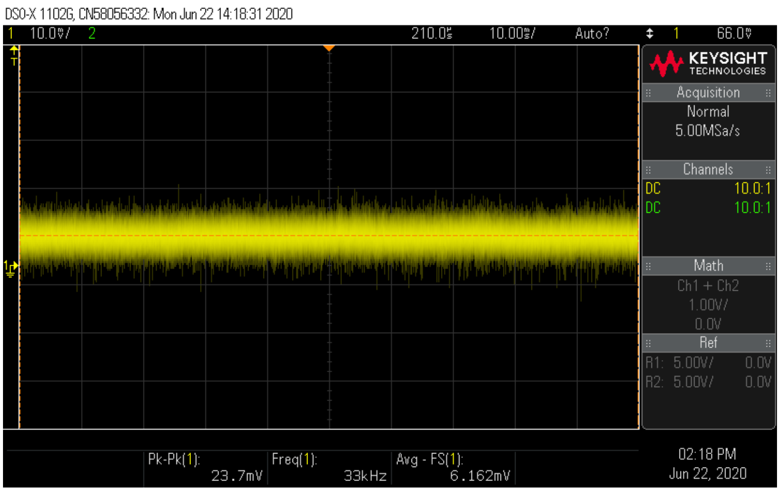


Figure 6.6 – Standby Current Consumption

Measurements of current consumption were taken with JP6 using a keysight oscilloscope through a 1Ω resistor as shown in the figure 6.7.

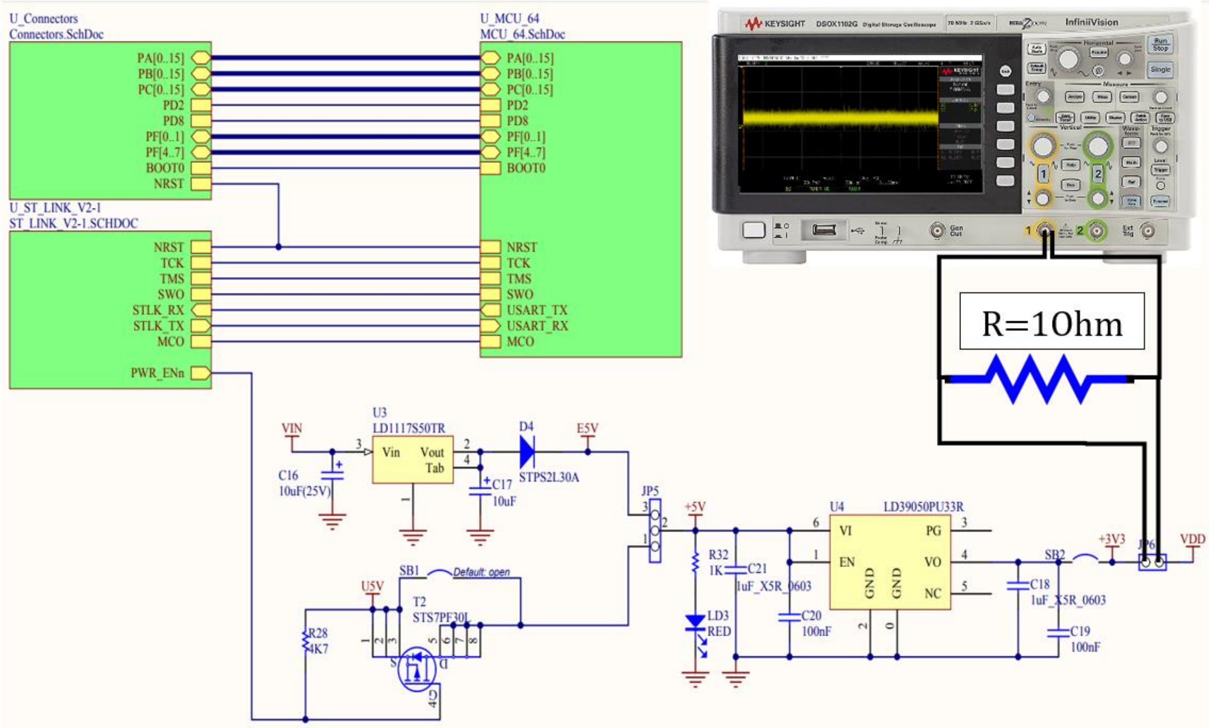


Figure 6.7 –Connection diagram for current measurement

In order to verify that the theory was working, a code was implemented that decrypted one message in an endless cycle, and when the button was pressed, the decryption of another began. The results obtained differed by 700 µA, but after analyzing the results obtained, the reason for this difference was determined. The reason for the increase in current was the button B1. This button is connected to the pull-up resistor R30 (Figure 6.8). The value of the resistor R30 is 4.7k. Dividing according to Ohm's law 4.7 / 3.3, we get the desired 700 µA.

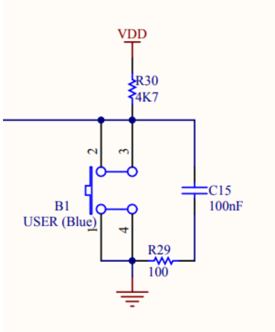


Figure 6.8 – Button b1

After making the corrections, it was found out that during the decryption process, the current consumption of the processor is around 20 µA (Figure 6.9).

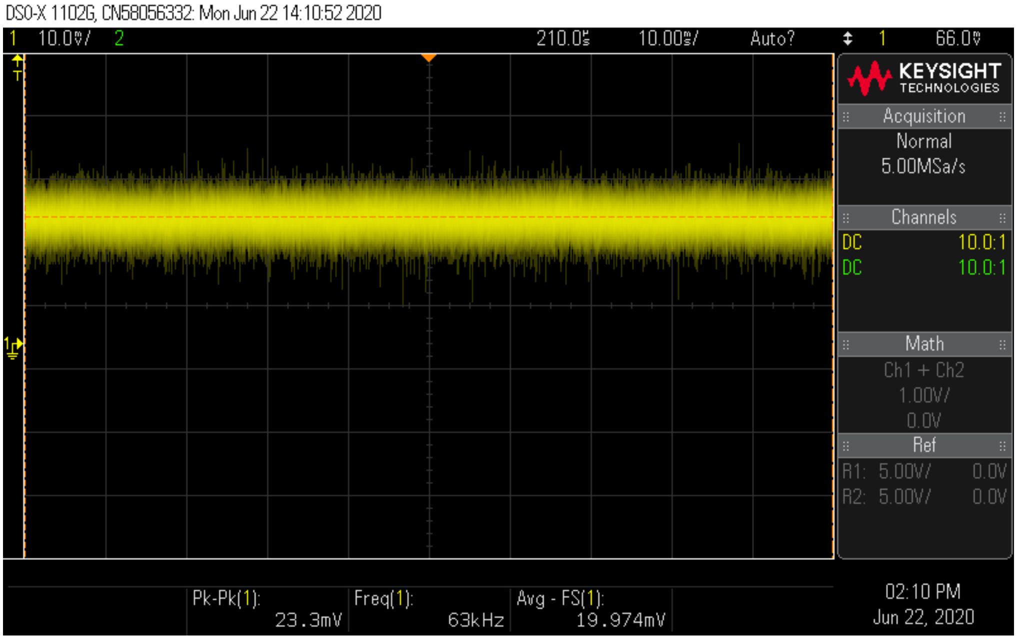


Figure 6.9 – Current consumption during decryption

6. Conclusion

With the hardware implementation of any cryptographic device, even with the most mathematically complex cryptographic algorithm, it is impossible to avoid information leakage through side channels. However, this information can be made illegible and prevent unauthorized access. So, the built-in temperature sensor can be used not only for personal gain, but also to prevent unauthorized access. You can add a security algorithm to the cryptographic device, which works when the temperature changes, when the probability of extracting encryption keys increases.

The vulnerability of RSA, at the moment, is caused not by the algorithm, but by its practical implementation. When using large keys, from a mathematical point of view, a large amount of effort and resources is required to factorize a semisimple number. However, the use of third-party data that occurs during the execution of the algorithm on the device can allow an attacker to extract useful information and obtain encryption keys of even the most complex algorithm.

With a passive attack on the temperature channel of a cryptographic device, it is not possible to extract encryption keys, however, temperature analysis of the device in conjunction with other third-party channels can help to extract encryption keys. So, for example, during a time attack of calculations in conjunction with a temperature channel, you can find out whether a special computation delay was introduced to protect against time attacks or not.

Active attacks on the temperature channel have already been investigated [27]. With active attacks, it was possible to successfully retrieve the RSA encryption keys based on erroneous calculations. A 2014 study by Michael Hutter and Jorn-Marc Schmidt used an external temperature sensor that compromised processor integrity. Using built-in temperature sensors for such attacks can be more promising, as visually the integrity of the device will not be compromised.

In general, it must be emphasized that security solutions for Internet of Things are not yet fully understood. With the development of technology and its role in the modern world, it is necessary to continue scientific research and propose modern solutions that can improve the situation in the field of security of the Internet of things.

References

1. Al-Fuqaha A., Guizani M., Mohammadi M., Aledhari M., Ayyash M. (2015) Internet of things: a survey on enabling technologies, protocols, and applications, *IEEE Commun. Surv. Tutorials*, 17(4), 2347–2376.
2. Lu U., Da Xu L. (2018). Internet of things (IoT) Cybersecurity Research: A review of current research topics. *IEEE Internet of Things Journal.*, 1-19. DOI:10.1109/JIOT.2018.2869847
3. Kalkan K., Zeadally S. (2017). Securing internet of things with software defined networking. *IEEE Communications Magazine*, 56(9), 186-192. DOI:10.1109/MCOM.2017.1700714
4. Ferguson N., Schroeppel R., Whiting D. (2001). A simple algebraic representation of Rijndael. *Selected Areas in Cryptography*, Proc. SAC, Lecture Notes in Computer, 2259, 103-111.
5. Feistel cipher. (2020, May 22). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Feistel_cipher
6. Advanced Encryption Standard. (2020, May 10). [Online resource]. Accessed at https://ru.wikipedia.org/wiki/Advanced_Encryption_Standard
7. Sukhoparov M.E., Lebedev I.S. (2016). Realization symmetric encryption on algorithm. *Information Security Problems. Computer Systems*, 4, 140-145. [Online resource]. Accessed at http://jisp.ru/en/volume/aspekty-informacionnoj-bezopasnosti-4/
8. Rivest R. L., Shamir A., Adleman L. (1978). A method for obtaining digital signatures and public-key cryptosystems. *Commun ACM*, 120–126. DOI:10.1145/359340.359342
9. Dime W., Hellman M. (1976). Multiuser cryptographic techniques. In *Proceedings of AFIPS 1976 NCC*. AFIPS Press, Montvale, New Jersey, 109-112.
10. Dime W., Hellman M.E. (1976). New directions in cryptography. *IEEE Trans. Info. Theory, IT*, 22(6), 644-654. (2020, March 17). [Online resource] – Accessed at https://ee.stanford.edu/~hellman/publications/24.pdf
11. Coutinho S.C. (1999). The Mathematics of Ciphers. *Number Theory and RSA Cryptography*. CRC Press, 1 edition, p. 323.
12. Menezes A. J., Oorschot P. V., Vanstone S. A. (1996). *Handbook of Applied Cryptography*. CRC Press, p. 816.
13. The expansion of the number. Institute of Computational Mathematics G.I. Marchuk Russian Academy of Sciences. (2020, February 24). [Online resource]. Accessed at https://www.inm.ras.ru/math_center/в-ивм-ран-получено-разложение-числа-rsa-232/

14. Fermat number (2020, February 26). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Fermat_number

15. Exponentiation by squaring (2020, February 27). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Exponentiation_by_squaring

16. ASCII (2020, February 27). [Online resource]. Accessed at https://ru.wikipedia.org/wiki/ASCII

17. Block cipher mode of operation. (2020, March 1). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Electronic_codebook_.28ECB.29

18. Kocher P.C. (1996). Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In N. Koblitz, editor, *Advances in Cryptology - CRYPTO '96*, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, Proceedings, 1109, p. 104–113.

19. Blake I.F., Serroussi G., Smart N.P. (2005). Advances in elliptic curve cryptography. *Cambridge University Press*.

20. Zhou Y., DengGuo Fenguo. (2006). Side-Channel Attacks: Ten Years After Its Publication and the Impacts on Cryptographic Module Security Testing. *Information Security Seminar WS*, 0607.

21. Brumley D., Boneh D. (2003). Remote timing attacks are practical. *CRYPTO'99, LNCS*, 1666, 388-397.

22. Shamir A., Trommer E. (2011). Acoustic cryptanalysis: on noisy people and noisy cars. A preliminary description of the concept. (2020, May 20). [Online resource]. Accessed at https://www.tau.ac.il/~tromer/acoustic/

23. Brouchier J., Dabbous N., Kean T., Marsh C., Naccache D. (2009). Thermocommunication. Temperature attacks. *IEEE Secur. Priv*., 7(2), 79–82.

24. Skorobogatov S. (2002). Low temperature data remanence in static RAM. *Technical report, University of Cambridge Computer Laboratory*.

25. Samyde D., Skorobogatov S.P., Anderson R.J., Quisquater J.-J. (2002). On a new way to read data from memory. *In: IEEE Security in Storage Workshop (SISW02)*, 65–69.

26. Muller T., Spreitzenbarth M. (2013). FROST. In: *Jacobson, M., Locasto, M., Mohassel, P., Safavi-Naini, R. (eds.) ACNS 2013. LNCS*, 7954, 373-388. Springer, Heidelberg.

27. Hutter M., Schmidt M. (2014). The Temperature Side Channel and Heating Fault Attacks. *Cryptography ePrint Archive*, 190. https://eprint.iacr.org/2014/190.pdf

28. Alice and Bob. (2020, April 1). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Alice_and_Bob

29. Boneh D., DeMillo R.A., Lipton R.J. (1997). On the importance of checking cryptographic protocols for faults. *EUROCRYPT '97*, *LNCS*, 1233, 37-51.

30. RM0360. *Reference manual.* STM32F030x4/6/8/C and STM32F070x6/B advanced ARM. (2020, May 21). [Online resource]. Accessed at https://www.st.com/resource/en/reference_manual/dm00091010-stm32f030x4-x6-x8-xc-and-stm32f070x6-xb-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

31. Boneh D., DeMillo R.A., Lipton R.J. (2001). On the Importance of Eliminating Errors in Cryptographic Computations. *Journal of Cryptology*, 14(2), 101-119.

32. Barichev S. (2002). *Cryptography without secrets*, p. 25. (2020, May 21). [Online resource]. Accessed at http://www.bnti.ru/dbtexts/ipks/old/analmat/1_2002/crypto4.pdf

33. Euclidean algorithm. (2020, February 21). [Online resource]. Accessed at https://en.wikipedia.org/wiki/Euclidean_algorithm

Appendix 1

```c
#include <stdio.h>
int checkPrime(unsigned long long n) {
        int i;
        unsigned long long m = n/2;
        for (i = 2; i <= m; i++) {
                if (n % i == 0) {
                        return 0; // Not Prime
                }}
        return 1;                       // Prime
}

int findGCD(unsigned long long n1,unsigned long long n2) {
        unsigned long long i, gcd;
        for(i = 1; i <= n1 && i <= n2; ++i) {
                if(n1 % i == 0 && n2 % i == 0)
                        gcd = i;}
        return gcd;
}

        unsigned long p, q;
        unsigned long long n;
        unsigned long long phin;
        unsigned long e, d;


int main() {
    p = 11;
    q = 13;
                printf("Calculation of the private key and public key for the RSA algorithm\n");
        while (1) {

                printf("//p = %u\n",p);//scanf("%u",&p);
                printf("//q = %u\n",q);//scanf("%u",&q);

                if (!(checkPrime(p) && checkPrime(q)))
                        printf("Both numbers are not prime. Please enter prime numbers only...\n");
                else if (!checkPrime(p))
                        printf("The first prime number you entered is not prime, please try again...\n");
                else if (!checkPrime(q))
                        printf("The second prime number you entered is not prime, please try again...\n");
                else
                        break;
        }
        n = p * q;
        phin = (p - 1) * (q - 1);
        printf("//n = %u\n",n);
        printf("//phin = %u\n",phin);
//======================================
//=========== Public key ===========
//======================================
        int i;
        i = 0;
        for (e = 3; e < phin; i++) {
            if (findGCD(phin, e) == 1)
                break;
            if (e<phin)
                e = pow(2,(pow(2,i)))+1;
                if (findGCD(phin, e) == 1)
                        break;
            else
                for (e = 3; e<phin; e++){
                    if (findGCD(phin, e) == 1)
                        break;
        }
        }
        printf("//e = %u\n",e);
//======================================
//=========== Private key =============
//======================================
//https://en.wikipedia.org/wiki/Euclidean_algorithm
    unsigned long long a, b, p=1, q=0, r=0, s=1, x, y;
    a = phin;
    b = e;
    while (a && b) {
        if (a>=b) {
            a = a - b;
            p = p - r;
```

```
            q = q - s;
        } else
        {
            b = b - a;
            r = r - p;
            s = s - q;
        }
    }
    if (a) {
        x = p;
        y = q;
    }else
    {
        x = r;
        y = s;
    }
    d = a*x+b*y;
/*
        for (d = e + 1; d < n; d++) {
                if ( (((d % phin)* e) % phin) == 1)
                        break;
        }*/
        printf("//d = %lu\n",d);
        if ((((d % phin)* e) % phin) != 1){
            printf("d is wrong!");
            return 1;}

        return 0;
}
```

Appendix 2

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int powMod(unsigned long long a, unsigned long long b, unsigned long long n) {
        long long x = 1, y = a;

        while (b > 0) {
                if (b % 2 == 1)
                        x = (x%n * y) % n;
                y = (y%n * y) % n; // (a*b) mod n = (a*(b mod n) mod n)
                b /= 2;
        }

        return x % n;
}
int main()
{
unsigned long long cipher;
unsigned long long decrypt;
unsigned long long e;
unsigned long long d;
unsigned long long n;

unsigned long data;

//p = 10007
//q = 399989
n = 4002689923;
//phin = 4002279928
e = 3;
d = 2668186619;

//data = 48;
//        printf("data: %u\n", data);

 /**< encrypt */
/*        for(int i=1;i<=e;i++)
        {cipher=(data*(cipher))%n;}*/
        cipher = powMod(data, e, n);
        printf("The cipher data is: %u\n", cipher);

 /**< decrypt */
/*

        for (int i =1;i<=d;i++)
        {
            decrypt = (decrypt*cipher)%n;
        }
*///      cipher = 110592;
        decrypt = powMod(cipher, d, n);
        printf("The decrypted data is: %u\n", decrypt);

        return 0;
}
```

Appendix 3

```c
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */


/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc;
DMA_HandleTypeDef hdma_adc;

TIM_HandleTypeDef htim3;

UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;

/* USER CODE BEGIN PV */
//==========================
//=======  RSA Value  =======
//==========================
//p = 10007;
//q = 399989;
//n=4002689923;
//e = 3;
//d=2668186619;

uint32_t tempDMA[2300] = {0};
uint64_t d = 2668186619;
//uint64_t e = 3;
uint64_t n = 4002689923;
unsigned long Encrypted = 110592;
//unsigned long Encrypted2 = 125000;
unsigned long MSG = 0;
unsigned short j = 1*2500;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC_Init(void);
static void MX_TIM3_Init(void);
static void MX_USART2_UART_Init(void);
```

```c
/* USER CODE BEGIN PFP */
int powMod(unsigned long long a, unsigned long long b, unsigned long long n) {
        long long x = 1, y = a;

        while (b > 0) {
                if (b % 2 == 1)
                        x = (x%n * y) % n;
                y = (y%n * y) % n; // (a*b) mod n = (a*(b mod n) mod n)
                b /= 2;
        }

        return x % n;
}
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC_Init();
  MX_TIM3_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */
//start the timer
    HAL_TIM_Base_Start(&htim3);
//start ADC as DMA
     HAL_ADC_Start_DMA(&hadc, tempDMA, 2300);



  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
        while (j)
    {
        j--;
//============= decrypt ===============
//          if (j>1250)
        MSG = powMod(Encrypted, d, n);
//          else
//        MSG = powMod(Encrypted2, d, n);

        if (j==0)
        {
            HAL_ADC_Stop_DMA(&hadc);

                uint8_t tempUART[5] = {0};


//sent temperature by UART
                for (unsigned short z=0; z<2300; z++)
                {
                sprintf(tempUART, "%lu\n" , tempDMA[z]);
                HAL_UART_Transmit(&huart2,tempUART, 5,100);
```

```
                    if (tempDMA[z]==0)
                        return 0;
                    }
                }

//      HAL_ADC_Start(&hadc);
//      tempConvert(int z, tempDMAvar)
/*      for(int z   = 0;z<100;z++){
            HAL_Delay(500);
        tempArray[z] = tempDMA[0];
        }
            if(HAL_ADC_PollForConversion(&hadc, 100) == HAL_OK)
        {
            TS_ADC_DATA = HAL_ADC_GetValue(&hadc);
            Vsense = (float) TS_ADC_DATA/4095*VDDA;
            Temperature = (V_30-Vsense)/Avg_Slope+30;

        }
*/

    }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_HSI14;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSI14State = RCC_HSI14_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.HSI14CalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL6;
  RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC_Init(void)
{

  /* USER CODE BEGIN ADC_Init 0 */

  /* USER CODE END ADC_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC_Init 1 */

  /* USER CODE END ADC_Init 1 */
  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and
number of conversion)
```

```c
  */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO;
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc.Init.DMAContinuousRequests = ENABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC_Init 2 */

  /* USER CODE END ADC_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 4800;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 100;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
```

```c
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
  /* DMA1_Channel4_5_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);

  /*Configure GPIO pin : PC13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pin : PA10 */
  GPIO_InitStruct.Pin = GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(hadc);
```

```c

  /* NOTE : This function should not be modified. When the callback is needed,
            function HAL_ADC_ConvCpltCallback must be implemented in the user file.
   */
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{ /* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Main program body
  ******************************************************************************
  * @attention
  *
  * <h2><center>&copy; Copyright (c) 2020 STMicroelectronics.
  * All rights reserved.</center></h2>
  *
  * This software component is licensed by ST under BSD 3-Clause license,
  * the "License"; You may not use this file except in compliance with the
  * License. You may obtain a copy of the License at:
  *                        opensource.org/licenses/BSD-3-Clause
  *
  ******************************************************************************
  */
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */


/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
ADC_HandleTypeDef hadc;
DMA_HandleTypeDef hdma_adc;

TIM_HandleTypeDef htim3;

UART_HandleTypeDef huart2;
DMA_HandleTypeDef hdma_usart2_rx;

/* USER CODE BEGIN PV */
//==========================
//=======  RSA Value  =======
```

```c
//===========================
//p = 10007;
//q = 399989;
//n=4002689923;
//e = 3;
//d=2668186619;

uint32_t tempDMA[2300] = {0};
uint64_t d = 2668186619;
//uint64_t e = 3;
uint64_t n = 4002689923;
unsigned long Encrypted = 110592;
//unsigned long Encrypted2 = 125000;
unsigned long MSG = 0;
unsigned short j = 1*2500;

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_DMA_Init(void);
static void MX_ADC_Init(void);
static void MX_TIM3_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */
int powMod(unsigned long long a, unsigned long long b, unsigned long long n) {
	long long x = 1, y = a;

	while (b > 0) {
		if (b % 2 == 1)
			x = (x%n * y) % n;
		y = (y%n * y) % n; // (a*b) mod n = (a*(b mod n) mod n)
		b /= 2;
	}

	return x % n;
}
/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* USER CODE BEGIN Init */

  /* USER CODE END Init */

  /* Configure the system clock */
  SystemClock_Config();

  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */

  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  MX_DMA_Init();
  MX_ADC_Init();
  MX_TIM3_Init();
  MX_USART2_UART_Init();
  /* USER CODE BEGIN 2 */
//start the timer
    HAL_TIM_Base_Start(&htim3);
//start ADC as DMA
     HAL_ADC_Start_DMA(&hadc, tempDMA, 2300);
```

```c
  /* USER CODE END 2 */

  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
        while (j)
    {
        j--;
//============== decrypt ===============
//          if (j>1250)
        MSG = powMod(Encrypted, d, n);
//          else
//        MSG = powMod(Encrypted2, d, n);

        if (j==0)
        {
            HAL_ADC_Stop_DMA(&hadc);

                uint8_t tempUART[5] = {0};


//sent temperature by UART
                for (unsigned short z=0; z<2300; z++)
                {
                sprintf(tempUART, "%lu\n" , tempDMA[z]);
                HAL_UART_Transmit(&huart2,tempUART, 5,100);
                if (tempDMA[z]==0)
                    return 0;
                }
            }

//      HAL_ADC_Start(&hadc);
//      tempConvert(int z, tempDMAvar)
/*      for(int z   = 0;z<100;z++){
            HAL_Delay(500);
        tempArray[z] = tempDMA[0];
        }
            if(HAL_ADC_PollForConversion(&hadc, 100) == HAL_OK)
        {
            TS_ADC_DATA = HAL_ADC_GetValue(&hadc);
            Vsense = (float) TS_ADC_DATA/4095*VDDA;
            Temperature = (V_30-Vsense)/Avg_Slope+30;

        }
*/

    }
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */

  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_HSI14;
  RCC_OscInitStruct.HSIState = RCC_HSI_ON;
  RCC_OscInitStruct.HSI14State = RCC_HSI14_ON;
  RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
  RCC_OscInitStruct.HSI14CalibrationValue = 16;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
  RCC_OscInitStruct.PLL.PLLMUL = RCC_PLL_MUL6;
  RCC_OscInitStruct.PLL.PREDIV = RCC_PREDIV_DIV1;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }
  /** Initializes the CPU, AHB and APB busses clocks
  */
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                              |RCC_CLOCKTYPE_PCLK1;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;

```

```c
  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
  {
    Error_Handler();
  }
}

/**
  * @brief ADC Initialization Function
  * @param None
  * @retval None
  */
static void MX_ADC_Init(void)
{

  /* USER CODE BEGIN ADC_Init 0 */

  /* USER CODE END ADC_Init 0 */

  ADC_ChannelConfTypeDef sConfig = {0};

  /* USER CODE BEGIN ADC_Init 1 */

  /* USER CODE END ADC_Init 1 */
  /** Configure the global features of the ADC (Clock, Resolution, Data Alignment and number of conversion)
  */
  hadc.Instance = ADC1;
  hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;
  hadc.Init.Resolution = ADC_RESOLUTION_12B;
  hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
  hadc.Init.ScanConvMode = ADC_SCAN_DIRECTION_FORWARD;
  hadc.Init.EOCSelection = ADC_EOC_SEQ_CONV;
  hadc.Init.LowPowerAutoWait = DISABLE;
  hadc.Init.LowPowerAutoPowerOff = DISABLE;
  hadc.Init.ContinuousConvMode = DISABLE;
  hadc.Init.DiscontinuousConvMode = DISABLE;
  hadc.Init.ExternalTrigConv = ADC_EXTERNALTRIGCONV_T3_TRGO;
  hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_RISING;
  hadc.Init.DMAContinuousRequests = ENABLE;
  hadc.Init.Overrun = ADC_OVR_DATA_PRESERVED;
  if (HAL_ADC_Init(&hadc) != HAL_OK)
  {
    Error_Handler();
  }
  /** Configure for the selected ADC regular channel to be converted.
  */
  sConfig.Channel = ADC_CHANNEL_TEMPSENSOR;
  sConfig.Rank = ADC_RANK_CHANNEL_NUMBER;
  sConfig.SamplingTime = ADC_SAMPLETIME_239CYCLES_5;
  if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN ADC_Init 2 */

  /* USER CODE END ADC_Init 2 */

}

/**
  * @brief TIM3 Initialization Function
  * @param None
  * @retval None
  */
static void MX_TIM3_Init(void)
{

  /* USER CODE BEGIN TIM3_Init 0 */

  /* USER CODE END TIM3_Init 0 */

  TIM_ClockConfigTypeDef sClockSourceConfig = {0};
  TIM_MasterConfigTypeDef sMasterConfig = {0};

  /* USER CODE BEGIN TIM3_Init 1 */

  /* USER CODE END TIM3_Init 1 */
  htim3.Instance = TIM3;
  htim3.Init.Prescaler = 4800;
  htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim3.Init.Period = 100;
  htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
  {
```

```
    Error_Handler();
  }
  sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
  if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
  {
    Error_Handler();
  }
  sMasterConfig.MasterOutputTrigger = TIM_TRGO_UPDATE;
  sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
  if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM3_Init 2 */

  /* USER CODE END TIM3_Init 2 */

}

/**
  * @brief USART2 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART2_UART_Init(void)
{

  /* USER CODE BEGIN USART2_Init 0 */

  /* USER CODE END USART2_Init 0 */

  /* USER CODE BEGIN USART2_Init 1 */

  /* USER CODE END USART2_Init 1 */
  huart2.Instance = USART2;
  huart2.Init.BaudRate = 115200;
  huart2.Init.WordLength = UART_WORDLENGTH_8B;
  huart2.Init.StopBits = UART_STOPBITS_1;
  huart2.Init.Parity = UART_PARITY_NONE;
  huart2.Init.Mode = UART_MODE_TX_RX;
  huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart2.Init.OverSampling = UART_OVERSAMPLING_16;
  huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart2) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART2_Init 2 */

  /* USER CODE END USART2_Init 2 */

}

/**
  * Enable DMA controller clock
  */
static void MX_DMA_Init(void)
{

  /* DMA controller clock enable */
  __HAL_RCC_DMA1_CLK_ENABLE();

  /* DMA interrupt init */
  /* DMA1_Channel1_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel1_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel1_IRQn);
  /* DMA1_Channel4_5_IRQn interrupt configuration */
  HAL_NVIC_SetPriority(DMA1_Channel4_5_IRQn, 0, 0);
  HAL_NVIC_EnableIRQ(DMA1_Channel4_5_IRQn);

}

/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
```

```
  __HAL_RCC_GPIOA_CLK_ENABLE();

  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOA, GPIO_PIN_10, GPIO_PIN_RESET);

  /*Configure GPIO pin : PC13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  /*Configure GPIO pin : PA10 */
  GPIO_InitStruct.Pin = GPIO_PIN_10;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

}

/* USER CODE BEGIN 4 */
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef* hadc)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(hadc);

  /* NOTE : This function should not be modified. When the callback is needed,
            function HAL_ADC_ConvCpltCallback must be implemented in the user file.
   */
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}
/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
void Error_Handler(void)
{
  /* USER CODE BEGIN Error_Handler_Debug */
  /* User can add his own implementation to report the HAL error return state */

  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/

  /* USER CODE BEGIN 6 */
  /* User can add his own implementation to report the file name and line number,
     tex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

/************************ (C) COPYRIGHT STMicroelectronics *****END OF FILE****/
```

Appendix 4

```c
#include <stdio.h>
#include <string.h>
//p = 29669093332083606603617799242426306347429462625218523944018571574194370194723262390744910112571804274494074452751891
//q = 34038161751975634380066094984915214205471217607347231727351634132760507061748526506443144325148088881115083863O1766

int main()
{
    int a[116],b[116]; //size in decimal of your p and q
    int RSA_size = 232;
    int ans[233]={0};
    int i,j,tmp;
    char s1[117],s2[117];
    printf("         Multiplying variables using arrays\n\n");
    printf("Value_1 = ");scanf(" %s",s1);
    printf("Value_2 = ");scanf(" %s",s2);
    int l1 = strlen(s1);
    int l2 = strlen(s2);

    printf("\nlength of first value: %d\nlength of second value:%d\n",l1,l2);

//preobrozovanie simvola v chislo i zapis zadom na pered
    for(i = l1-1,j=0;i>=0;i--,j++)
    {
        a[j] = s1[i]-'0';
    }
    for(i = l2-1,j=0;i>=0;i--,j++)
    {
        b[j] = s2[i]-'0';
    }
//umnozenie v stolbik
    for(i = 0;i < l2;i++)
    {
        for(j = 0;j < l1;j++)
        {
            ans[i+j] += b[i]*a[j];
        }
    }
    for(i = 0;i < l1+l2;i++)
    {
        tmp = ans[i]/10;
        ans[i] = ans[i]%10;
        ans[i+1] = ans[i+1] + tmp;
    }

    //nahodim dlinu proizvedeniya
    for(i = l1+l2; i>= 0;i--)
    {
        if(ans[i] > 0)
            break;
    }
    printf("\nProduct : ");

    //pechataem proizvedenie v privichnoi cheloveku forme
    for(;i >= 0;i--)
    {
        printf("%d",ans[i]);
    }
    printf("\n");
    return 0;
}
```