A hybrid encryption system with a Token device to avoid leak of information in corporate email

Author:

Luis Hernando Santamaría Bernales

Director:

Cesar Leonardo Niño, Ph.D.

Xaverian Pontifical University
School of Electronic Engineering
Bogotá, Colombia
May, 2014
Abstract

In this degree project, an implementation of both symmetric and asymmetric encryption algorithms for sending information through the SMTP protocol with a token device for authentication is proposed for an enterprise email application. The proposed system satisfies most security requirements such as confidentiality, integrity, non-repudiation, and authentication. This software implementation combines the advantages of both symmetric-key algorithms and public-key algorithms that allow to have greater security and a fast implementation, therefore achieving better performance.
Contents

Abstract i

Contents ii

List of Figures iv

List of Tables v

1 Introduction 1

2 Background Information 3
  2.1 Overview of AES 3
  2.2 Overview of RSA 4
    2.2.1 Key Generation, Encryption and Decryption 5
    2.2.2 Confidentiality, Authentication and no repudiation 6
  2.3 SMTP Protocol 8
  2.4 PGP 8

3 Specifications 10
  3.1 Key management server 10
  3.2 Client System 10
  3.3 USB token 11
  3.4 Mail Server 11
  3.5 Communication Protocols 12
    3.5.1 Sending an Email Operation 12
    3.5.2 Receiving and Email Operation 13
    3.5.3 Key Management Server Operation 13

4 System Development 15

5 Analysis of Results 17
  5.1 Analysis of the proposed system against PGP 20
  5.2 System Performance 21

6 Conclusions 25
Bibliography 26

Annex A. Source Code 29
List of Figures

2.1 AES-128 Encryption and Decryption Algorithm ............... 5
2.2 Scheme that provides confidentiality. ..................... 6
2.3 Scheme that provides authentication and non-repudiation. .... 7
2.4 Public Key scheme that provides confidentiality, authentication and
non-repudiation. ............................................ 7
2.5 SMTP Protocol. ........................................... 8
2.6 How PGP works .......................................... 9

3.1 SDL for USB Token Verification. ............................. 11
3.2 SDL client for sending email. ................................. 12
3.3 SDL client for receiving email. ............................... 13
3.4 SDL for Key Management Server. .............................. 14

4.1 Architecture of the Secure Email System with an Authentication
Token and a Hybrid Crypto System implemented. .................. 16

5.1 Encryption operation example with AES. ...................... 18
5.2 Encryption operation example with RSA. ..................... 18
5.3 Sending a message in Email Client. .......................... 19
5.4 Email encrypted received. .................................. 19
5.5 Plain-text after decryption process. .......................... 20
5.6 Encryption and Decryption time with a 512 bit RSA Key. ..... 22
5.7 Encryption and Decryption time with a 1024 bit RSA Key. .... 22
5.8 Encryption and Decryption time with a 2048 bit RSA Key. .... 23
5.9 Time comparison between encryption and decryption operation. .. 24
List of Tables

2.1 AES Key Block Round Combinations . . . . . . . . . . . . . . . . . . 4

5.1 System Characteristics . . . . . . . . . . . . . . . . . . . . . . . . . 21
5.2 Mean, standard deviation and coefficient of variation for encryption
  and decryption process with different sizes of RSA key. . . . . . . . 23
5.3 Time to encrypt AES-128 Key with RSA . . . . . . . . . . . . . . . 23
5.4 Time to decrypt AES-128 Key with RSA . . . . . . . . . . . . . . . 23
Chapter 1

Introduction

Information is one of the most important assets on an enterprise and e-mail is one of the essential tools to communicate and transmit messages. The advances in communications in recent decades have enabled organizations and enterprises to interconnect their networks, making them accessible from virtually anywhere in the world, but this also leaves the possibility of sensitive information being stolen. One of the most used applications is Email, unfortunately messages are not protected most of the time. Email Messages can be intercepted and read by unauthorized people. Messages can also be modified creating the impression that a person made a statement that he did not do[1].

Ordinary email does not provide techniques to assure integrity, privacy or authentication, therefore cryptography techniques are needed to encrypt a message so that it is not intelligible. The intention is that an encrypted message can only be decrypted by authorized personnel. Cryptography refers to the art of secret writing, which enables someone to secure the content of a message from unauthorized recipients. The two types of cryptography are symmetric cryptography and asymmetric cryptography. Symmetric key algorithms use the same key to encrypt and decrypt, such as the DES algorithm (Data Encryption Standard), Triple DES [2] and the AES (Advanced Encryption Standard) [3].

The advantage of symmetric-key algorithms is that they are faster in execution because of the use of straightforward cryptographic transformations, which can be pipelined to give better performance[4]. However, with such systems is necessary to distribute the key between the two parties. Sharing the key involves the risk that it may be stolen by an intruder and if the key is compromised all the security
is lost. In 1974, Diffie and Hellman first proposed a class of public key systems [5]. Essentially, the key is used by all the people to encrypt messages to a given user. The system also has a private key used by the final user to decrypt the message. Public key algorithms have the principle that it is too difficult to calculate the private key from the public key, usually based on the difficulty of factoring large numbers and computing discrete logarithms. One of the most popular public key algorithms is the RSA (Rivest, Shamir, Adleman) [6].

Security in email has been approached from different perspectives, for example in [7] an email architecture is proposed using TLS/SSL which provides protection against viruses as well as spam email. Available email standards such as S/MIME [8] and OpenPGP [9] focus on content protection using cryptographic algorithms. One drawback of security email features is that a lot of them are not used because the users do not know how to configure them, leaving the system exposed to the loss of important information. Another solution in [10] describes a certificate-based solution to check the authenticity of users, where the receiver has to show his certificate before getting the message. Other solutions make use of a smart card to store user credentials, as it is proposed in [11]. There have been proposals on hybrid encryption that make use of both symmetric-key and public-key cryptographic algorithms like [4] using DES and AES combined with RSA to cypher plain text.

Although cryptography prevents encrypted information of being understandable, there is still a problem when an intruder has access within the private network to one of the computers [12] and could read messages from an user or sends messages on its behalf. Also, authentication based on name and password is considered weak as an intruder could break it through a dictionary attack. Since cryptography alone is not sufficient, it must be accompanied by an authentication method to ensure that indeed the right person is going to send or receive mail. This paper presents and hybrid crypto system using AES to cipher content of an email message, RSA to share the private Key along with a Token USB for authentication purposes. This implementation shows interesting results in terms of security and performance.
Chapter 2

Background Information

2.1 Overview of AES

AES is based on the Rijndael block cipher and it has become the successor of DES being chosen by the US National Institute of Standards and Technology (NIST), which specifies the cryptographic algorithm to protect sensitive information by the U.S. Government. Rijndael Algorithm was chosen to be the AES in a quite demanding contest that had the following rules:

- Must be a symmetric block cipher.
- Keys with size of 128, 196 and 256 bits.
- The design must be public.
- Hardware and Software implementations must be possible.

The algorithm uses permutations and substitutions in several rounds (10, 12 or 14 according to the key size) that are SubBytes, ShiftRows, MixColumns, and AddRoundKey. These are mathematical operations that convert a plain text block into a cipher block when it is doing encryption, and converts a cipher block into a plain text block when it is doing decryption. The number of rounds \(Nr\) that the algorithm executes depends on the key length \(Nk\), which is the number of key 32-bit words. This can be seen in Table 2.1.
Chapter 3. Background Information

Table 2.1: AES Key Block Round Combinations

<table>
<thead>
<tr>
<th></th>
<th>Key Length (Nk words)</th>
<th>Block Size (Nb words)</th>
<th>Number of Rounds (Nr)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AES-128</td>
<td>4</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>AES-192</td>
<td>6</td>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>AES-256</td>
<td>8</td>
<td>4</td>
<td>14</td>
</tr>
</tbody>
</table>

The AES algorithm is summarized in Figure 2.1. The first step is to copy the plain text in the array state in order to process it during successive rounds. Then, it goes to a first round of AddRoundKey. The cycle runs 10 iterations, one per round, and transforms the array state in each iteration.

- **SubBytes Transformation**: A mono alphabetic substitution is done using a unique substitution table (S-Box).

- **ShiftRows Transformation**: This step shifts left each of the four rows. The 0th row is rotated 0 bytes (no change), row 1 is rotated 1 byte, and so on.

- **MixColumns Transformation**: Mixes each column independently of the other columns. Mixing is performed using a multiplication matrix where each new column is the product of the old column and a constant matrix, and the multiplication is performed by the finite Galois field, $GF(2^8)$.

- **Add Round Key**: Apply XOR to this round key within (state).

### 2.2 Overview of RSA

RSA (Rivest, Shamir, Adleman) is a public (asymmetric) key cryptosystem based on Number theory, it was developed on 1977 and its strength is based on the difficulty that represents the factorization of a large prime number. In the algorithm the plain-text and the cipher text are considered integers between 0 and $n - 1$, where $n$ is the modulus. The typical size of $n$ is 1024 bits, 512 also used when a fast implementation is needed. However, for secure applications a length of 2048 bits is recommended [13].

The RSA algorithm key generation process, encryption and decryption is explained below in Section 2.2.1.
2.2.1 Key Generation, Encryption and Decryption

Two large pseudo-random prime numbers are selected, that are called $p$ and $q$ which must be kept in secret. "Large" in cryptography usually means 512 bits or more [14]. After selecting the prime numbers the algorithm works as follows:

- Compute modulus $n = p.q$
- Calculate totient, $(n) = (p - 1).(q - 1)$
- Choose an integer $e$, such that $gcd(e', (n)) = 1$. 

---

**Figure 2.1**: AES-128 Encryption and Decryption Algorithm
• Compute the secret exponent \( d \), such that \( d \cdot e \equiv 1(\text{mod } n) \).

The public key is composed by the pair \((n, e)\) and the private key is the pair \((n, d)\). The encryption is done by equation 2.1 where the input \( M \) is the message (represented by a positive integer) and the output \( C \) is the cipher-text.

\[
C = M^e \text{mod}(n) \tag{2.1}
\]

The message is decrypted by using equation 2.2, and extracting the plain-text from the integer representative \( M \). Encryption and decryption process are inverse because of the relationship of exponents \( e \) and \( d \).

\[
M = C^d \text{mod}(n) \tag{2.2}
\]

### 2.2.2 Confidentiality, Authentication and no repudiation

Public cryptography give us the possibility of having confidentiality, authentication and no repudiation assured. Confidentiality is the process of protecting the data from disclosure to unauthorized individuals during a communication. If user A wants to send a message to user B he would need to encrypt it using the Public Key of B \( (K_{PB}) \) so that the only user capable of decrypting this message is B because he is the only one who have access to the secret Key of B \( (K_{SB}) \). This can be seen in Figure 2.2.

![Figure 2.2: Scheme that provides confidentiality.](image-url)
Authentication [18] is the process of identifying the sender and/or the receiver and no repudiation is the process of preventing a sender from denying transmitted data. Both attributes will be complied if the sender A uses its own secret key ($K_{SA}$) when encrypting the message. The recipient will use the Public Key of A for decrypting ($K_{PA}$) so he can be sure that the message comes from user A. This has a property of a Digital Signature where anyone can verify the signature, but only the originating part can generate it [14]. This can be seen in Figure 2.3.

![Figure 2.3: Scheme that provides authentication and non-repudiation.](image1)

The three attributes can be achieved if there is a double RSA encryption, each user with a different pair of keys. This can be seen in Figure 2.4.

![Figure 2.4: Public Key scheme that provides confidentiality, authentication and non-repudiation.](image2)
2.3 SMTP Protocol

SMTP (Simple Mail Transfer Protocol) was originally defined in RFC 821 to transfer emails between stations. Currently, the SMTP protocol is defined in RFC 5321 [15]. The protocol focuses on how messages are transmitted.

When sending a message the SMTP protocol makes use of port 25 to establish a TCP connection to the SMTP server. The SMTP server indicates whether it is willing to receive the e-mail, and it starts by sending the source and destination address, and then the message. If the operation is complete the server replies with an acknowledgement of received (ACK). The SMTP operating scheme can be seen in Figure 2.5.

![Figure 2.5: SMTP Protocol.](image)

2.4 PGP

A solution that is worth analyzing is PGP (Pretty Good Privacy). It is an application and protocol RFC 4880 [16] that have been in use since 1991. The main purpose of PGP is encryption and decryption of messages for secure e-mail. It makes use of symmetric cryptography to cipher a plain text, asymmetric cryptography to share the session key and digital signature to guarantee authentication and non repudiation.

In Figure 2.6 the encryption and decryption process for PGP is shown. It first take a plain text and calculate the Message Digest using MD5. Then it creates a random session key generated from movements of the keyboard and mouse to encrypt the message using symmetric algorithms IDEA or CAST. Finally it ciphers
the session key using public cryptography with RSA or DSA with the public key of the receiver. Both encrypted message and encrypted session key are send to the receiver by E-mail. One main advantage of PGP is that the list of support algorithms have been permanently growing.

For decryption the receiver have to known which algorithms were used to encrypted, this information needs to be included within the information received [17]. The receiver uses its own private key to obtain the one time session key. It then uses the session key and gets the message and signature which can be validate with the public key of the sender.
Chapter 3

Specifications

The system consists of a key management server, the client system, the USB token and e-mail server to exchange messages. There are several protocols for communication between the client and server key management developed. All parts are implemented using Java programming language.

3.1 Key management server

This server manages and stores all public keys of the users of the system, they may be requested by a user when decrypting a message. The server is managed by a single administrator who is responsible for the management of public keys of the users of the company. The server will also store encrypted symmetric keys sent by users when sending an email with an ID to identify the message.

3.2 Client System

It is a client module to be run on the user PC and has built-in security features. The customer must ensure access control using a USB Token and a secret key to properly identify the user. The client must encrypt the contents of email messages with a 128 AES key and sharing this key through the key management server using asymmetric cryptography.
3.3 USB token

Each user of the system must have a USB Token assigned internally which must contain private login information of their email account. When the email client starts, the system verifies if there is a USB token plugged into the system. If the USB is not connected, the system asks for it through a message. Once there is a valid Token within the system it takes its email account information (user and key) in order to send or receive messages. The client executes a loop to verify that the USB remain connected at all times, at the time it is removed the user is automatically logout of the system as seen on Figure 3.1.

![Figure 3.1: SDL for USB Token Verification.](image)

3.4 Mail Server

The email server is used by clients to exchange messages. Mail server relays all messages using Simple Mail Transport Protocol.
3.5 Communication Protocols

Communication protocols are also implemented between the client and the server key management to send and receive messages.

3.5.1 Sending an Email Operation

The user makes use of the client interface to send an email. An AES-128 Key is generated and once the user press the Send button the content is encrypted with this key and connection with the Key Management Server starts. The client asks for the Public Key of the receiver and then it begins the double encryption process of the AES-128 Key. Finally the client sends this encrypted key to the database server and the email is send as seen on Figure 3.2.

![SDL client for sending email.](image-url)
3.5.2 Receiving and Email Operation

When an encrypted message is receive the client starts the communication with the Key Management Server. The client asks for the encrypted AES-128 Key and the sender Public Key. Then it begins the asymmetric decryption process to obtain the AES-128 Key which is then use to decrypt the content by the client. This can be seen on Figure 3.3

![Diagram](image)

**Figure 3.3:** SDL client for receiving email.

3.5.3 Key Management Server Operation

The key management server keeps running on listening state waiting for a connection request from a client. Once the connection request the client identifies and
confirms that the connection was established is received. There are three kind requests that the server can receive from the client. If a 001 request ID is received, it means that the client is sending encrypted secret AES Key. The server verifies the values of sender and recipient and stores this information. If a 010 request ID is received, it means that the client is asking for a Public Key of a user. The server must verify if this key exists within its database and send it to the client. If a 011 request ID is received, it means that a client is asking for a symmetric AES key encrypted. The server verifies the sender and recipient address values, and it must send this information to the client. This can be seen on Figure 3.4

Figure 3.4: SDL for Key Management Server.
Chapter 4

System Development

AES have proven to be a highly secure algorithm against many attacks such brute-force, differential and linear attacks [19], and continues to be the best single key algorithm because of security, cost and implementation. AES has less implementation complexity and can easily encrypt large data, in comparison with the asymmetric algorithm RSA, which uses large keys and so it requires more computational resources for encryption. However, the secret key distribution remains an issue of symmetric algorithms. In order to solve this issue it is proposed the use of a hybrid encryption system that uses RSA for distributing the secret key of AES. The system also uses a token device for physical authentication that helps preventing the leak of sensitive information.

The token device stores information such name and password of a user (email account) and the requirement is that it must be connected at all times in order to use the application. This authentication token have an important role in the system since it is assuring that a valid user is the one using the system. The token is a small device in this case simulated by means of a USB. This is a major advantage since there is no need of a reader, because just about every computer on the planet has one or more USB connector [20]. If the USB token is unplugged, then the system is automatically blocked preventing that any other user seeing any information or from attempting to use the email application.

The system also makes use of a database server to store the public key of the users and the AES encrypted keys that were generated when an email was sent. This database server must be a trusted third party inside the organization so it
can assert that a public key belongs to a particular user. All components of the system and its scheme can be seen in Figure 4.1.

Figure 4.1: Architecture of the Secure Email System with an Authentication Token and a Hybrid Crypto System implemented.

When a message is created, an AES 128-bit key is automatically generated which is then used to encrypt the content of the message, the message is sent to the mail server through SMTP protocol. The 128 key provides an space of $2^{128} \approx 10^{38}$ combinations which is very secure. If there was a machine with a thousand million processors in parallel it would take approximately $10^{10}$ years to complete the key space [5]. The mail server is then responsible for delivering the message to the recipient mail server, so that the user can receive the encrypted message via POP3 or IMAP protocols.

The selected 128-bit key ($K$) which was used to encrypt the message, is then encrypted twice using the RSA algorithm. The user A who wants to send a message to user B, uses its own secret RSA key ($K_{SA}$) to encrypt $K$ becoming $K^*$ and then uses the public RSA key of B ($K_{PB}$) to encrypt one more time the previous result. This double encryption guarantees the confidentiality, authentication and no repudiation in the communications as it was seen in the previews section. When the user B gets connected, it receives the encrypted AES email and the encrypted AES key ($K^{**}$). Then, it starts the process of deciphering the $K^{**}$ with the RSA algorithm, first with its own secret key ($K_{SB}$) and then with the public key of A ($K_{PA}$) to finally obtain $K$. $K$ is then used to decipher the encrypted email (AES algorithm).
Chapter 5

Analysis of Results

In order to evaluate the crypto-system a proof of concept was developed using JAVA on the Netbeans platform. A random AES-128 key was generated to encrypt text messages. This selected key was then double cipher with two different pair of keys (public and private respectively) and stored on a database server. Encrypted message was then send to the Mail Box using email information from an USB simulated token. To get the message, the recipient of the mail must use its own token to login and view the database. A decryption process is done using two times RSA and then uses the decrypted AES key to obtain the original plain-text.

The application was tested by parts. First, the AES encryption and decryption process was proven against verified examples such the ones in Advanced Encryption Standard (AES) (FIPS PUB 197) [21] and The Advanced Encryption Standard Algorithm Validation Suite (AESAVS) [22]. An example of the encryption process is shown in Figure 5.1 the inputs Plain-text and Key where selected according to the example C.1 in [21] and the output matches correctly.

The RSA key generation, encryption and decryption operation were also tested. In Figure 5.2 the values 47 and 59 were chosen for $p$ and $q$ respectively. The $e$ value was selected to be 17 to match the example in [23] chapter 8. The rest of the values were properly calculated, $n = p * q = 47 * 59 = 2773$, $d = 157$, $\text{totient}(n) = 46 * 58 = 2668$. The characters were substitute using two digits for each letter: blank = 00, A = 01, B = 02, . . . , Z = 26. The plain-text ”ITS ALL GREEK TO ME” was then encrypted and the output matches correctly as in [23].
Chapter 6. Analysis of Results

Figure 5.1: Encryption operation example with AES.

Figure 5.2: Encryption operation example with RSA.
Once the two algorithms were tested successfully, they were implemented on the email client. In Figure 5.3 it can be seen that the plain-text "Hello World." is send.

![Figure 5.3: Sending a message in Email Client.](image)

In Figure 5.4 the message is receive and the text is in fact encrypted by an AES-128 Key and in Figure 5.5 it can be seen the message decrypted when using the USB Token and executing the RSA and AES decryption process.

![Figure 5.4: Email encrypted received.](image)
5.1 Analysis of the proposed system against PGP

The proposed system and PGP solution provide security for e-mail application using a hybrid cryptographic system that consists of symmetric cryptography and public key cryptography. PGP does use some of the existing algorithms such as MD5, RSA, and IDEA integrated with the application for e-mail security [24] as so it is the proposed system. The way that the encryption and decryption is done is slightly different. The proposed system uses the RSA algorithm to encrypt twice using two pairs of keys. In the other hand, PGP uses RSA in one way to protect the secret key, but it also make use of MD5 for signing the message. Accordingly, both proposed solutions can provide confidentiality, authentication and non-repudiation.

However the PGP solution lacks of security when multiple users make use of the same computer according to Phil Zimmermann creator of PGP [25]. There are several multi risk associated on multi user system such plain text can not be protect it while it is still being plain text. For example if the user leaves the system it could be exposed for someone else to access its private information. Also having a single private key to log on the application is very risky. If somehow it gets compromised all the security will be lost.

The proposed system is more secure because it uses two mechanisms for controlling access. The key that only the user knows to enter the system and a security token that validates the user’s identity. So that if the user is leaves the computer with the Token, the application automatically closes and does not allow a different user to see your information. Also if your secret key is compromised, the attacker would still need the USB Token to access the system.
5.2 System Performance

The system was tested by measuring the time it takes for the RSA algorithm to encrypt the AES secret key that consists of 128 bits for all cases. Tests were made using RSA keys that were previously calculated from $P$ and $Q$ values of 512, 1024 and 2048 bits. The tests were run on a computer with the characteristics of the table 5.1.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Windows 7 64 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processador</td>
<td>Intel Core i5-520M (3M Cache, 2.4 GHz)</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>2</td>
</tr>
<tr>
<td>RAM</td>
<td>4.00 Gb</td>
</tr>
<tr>
<td>SDK</td>
<td>Netbeans 7.3</td>
</tr>
</tbody>
</table>

**Table 5.1: System Characteristics**

For each key size, a total of 50 measurements were performed for encryption and decryption using NanoTime Java method which measures the elapsed time with a nanosecond precision [26].

Using a 512 bit key for RSA algorithm, the average time to encrypt a 128 AES key is 80,313 ms. The average time for decryption is 100,944 ms. The standard deviation for the encryption process is 6,434 ms and the decryption process is 11,17 ms. It can be seen that decryption process shows a coefficient of variation of 11.06% which is slightly high, however the impact is not noticeable in the use of the application. The measured times for encryption and decryption in the test can be seen in Figure 5.6.

The average time to encrypt with a 1024 RSA key is 538,361 ms and the average time for decryption is 645,299 ms. The standard deviation for the encryption process is 28,024 ms and the decryption process is 22,399 ms. The coefficients of variation are 5.2% and 3.4% respectively showing an acceptable result for the tests. The measured times for encryption and decryption in the test can be seen in Figure 5.7.

Finally using a 2048 bit key for RSA algorithm, the average time to encrypt a 128 AES key is 3913,414 ms. The average time for decryption is 4654,309 ms. The standard deviation for the encryption process is 141,214 ms and the decryption process is 130,676 ms. It can be seen that encryption process shows a coefficient of variation of 3.6% and decryption process 2.8%, which implies less dispersion.
Figure 5.6: Encryption and Decryption time with a 512 bit RSA Key.

Figure 5.7: Encryption and Decryption time with a 1024 bit RSA Key.

compared to the average. The measured times for encryption and decryption in the test can be seen in Figure 5.8.

In Table 5.2 it can be seen the values for the average, standard deviation and coefficient of variation calculated from a sample of 50 test for each RSA key size encryption and decryption process.

It is observed from Table 5.3 the comparison of elapsed time using different sizes of RSA pair of keys to encrypt the secret shared key of 128 bits.

In Table 5.4 it shows the time to decrypt the secret shared key, and it can be seen that it is higher than the encryption time for all the key sizes. This times are
Figure 5.8: Encryption and Decryption time with a 2048 bit RSA Key.

<table>
<thead>
<tr>
<th>RSA Key Size (bits)</th>
<th>Av. Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>80,31329434</td>
</tr>
<tr>
<td>1024</td>
<td>538,3611531</td>
</tr>
<tr>
<td>2048</td>
<td>3913,41448</td>
</tr>
</tbody>
</table>

Table 5.2: Mean, standard deviation and coefficient of variation for encryption and decryption process with different sizes of RSA key.

Table 5.3: Time to encrypt AES-128 Key with RSA

<table>
<thead>
<tr>
<th>RSA Key Size (bits)</th>
<th>Av. Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>100,9449672</td>
</tr>
<tr>
<td>1024</td>
<td>645,299484</td>
</tr>
<tr>
<td>2048</td>
<td>4654,30903</td>
</tr>
</tbody>
</table>

tolerable for an email application but the most important is the fact that it would not increase since the system is using a unique 128 bit AES key to encrypt and decrypt. CPU performance has a direct impact on the overall results to process the cryptography operations.

<table>
<thead>
<tr>
<th>RSA Key Size (bits)</th>
<th>Av. Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>6,434,211358</td>
</tr>
<tr>
<td>1024</td>
<td>11,17071692</td>
</tr>
<tr>
<td>2048</td>
<td>4654,30903</td>
</tr>
</tbody>
</table>

Table 5.4: Time to decrypt AES-128 Key with RSA
Chapter 6. *Analysis of Results*

The normal key sizes for RSA are 512, 1024 and 2048 bits so if the crypto system is implemented, there is need to have a discussion regarding performances against security priorities. For instance, it is said that 1024-bit keys could be compromised in the near future so a 2048-bit key would be the choice in order to have a more robust implementation.

According to Figure 5.9 where both encryption and decryption times are shown, it can be seen that the time is increased by a ratio of 6 or 7 times each time the key size is incremented.

![Figure 5.9: Time comparison between encryption and decryption operation.](image)
Chapter 6

Conclusions

In this project degree it is presented a hybrid encryption scheme with the AES and RSA algorithms using an external Token device for authentication for enterprise email purposes.

RSA is an asymmetric algorithm than can satisfy most security concerns such confidentiality, authentication and non-repudiation, when the RSA is implemented in both communication ways. However the main drawback of RSA is the computational overhead because of the key size.

AES have less implementation complexity and it is one of the most strongest single-key algorithms. In this case, the key distribution is considered to be the major drawback.

An optimal solution was implemented using the AES 128-bit to encrypt the messages, taking advantage of its speed, and using RSA for sharing the secret key safely. The token USB increases the security of the system by authenticating users when using the application.
Bibliography


[26] JavaTM 2 Platform Standard Ed. 5.0, Class System.
   http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/System.html
Annex A. Source Code

The RSA.java is the class that implements the logic of the RSA algorithm. It contains the key generation process of the RSA algorithm. The class java.util.Random is used for the prime number generation $P$ and $Q$. All variables are calculated in order to get the Pair of keys.

For encryption and decryption of RSA the method modPow is used which allows us to get the BigInteger whose value is (this exponent mod m).

```java
package logica.rsa;

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Random;
import logica.util.Constantes;
import logica.util.MetodosUtiles;

/**
 * @author Luis H Santamaria
 */
public class RSA {

    private BigInteger p;
    private BigInteger q;
    private ParDeClaves claves;
    private String desc = "";

    /**
     * Constructor
     *
     * @param privadaD
     * @param publicaE
     * @param n
     */
```
public RSA(BigInteger privadaD, BigInteger publicaE, BigInteger n) {
    claves = new ParDeClaves(privadaD, publicaE, n);
}

/∗∗
 ∗ Constructor
 ∗/
public RSA() {
    generaPrimos();
    generaClaves();

    String s = "P: " + p + "\t" + "Q: " + q + "\t" + "N: " + claves.getN() + "\t" + "D: " + claves.getPrivadaD().getD() + "\t" + "E: \n" + claves.getPublicaE().getE();
    addDesc(s);
}

/∗∗
 ∗ Method in charge of generate primes P and Q to calculate the key.
 ∗/
private void generaPrimos() {
    p = new BigInteger(Constantes.BIT_LONGITUD, Constantes.LIM_PRIMO_RSA, new Random());

    do { //That they are not the same
        q = new BigInteger(Constantes.BIT_LONGITUD, Constantes.LIM_PRIMO_RSA, new Random());
    } while (q.compareTo(p) == 0); //While they are equal, then generate it again.
}

/∗∗
 ∗ Generate the keys
 ∗/
private void generaClaves() {
    BigInteger e;
    BigInteger d;
    BigInteger n;
    BigInteger toTient;

    // n = p * q
Annex A. Source Code

```java
n = p.multiply(q);
// totient = (p-1)*(q-1)
toTient = p.subtract(BigInteger.valueOf(1));
toTient = toTient.multiply(q.subtract(BigInteger.valueOf(1)));

// We select a coprime number E of Totient
do {
    e = new BigInteger(2 * Constantes.BIT_LONGITUD, new Random());
} while (e.compareTo(toTient) != -1 || e.gcd(toTient).compareTo(BigInteger.valueOf(1)) != 0);

// We found the value of d. d = e^{-1} mod toTient
d = e.modInverse(toTient);

claves = new ParDeClaves(n, toTient, d, e);
}

/**
 * @return
 */
public BigInteger getP() {
    return p;
}

/**
 * @return
 */
public BigInteger getQ() {
    return q;
}

/**
 * @return
 */
public ParDeClaves getClaves() {
    return claves;
}

/**
 * Encrypts the AES key with the private key D
 */
```
Annex A. Source Code

* @param secretKey
* @return
*/

public ArrayList<byte[]> cifrarConPrivada(byte[] secretKey) {
    addDesc("Clave en bytes: ");
    addDesc(MetodosUtiles.enBytes(secretKey));

    long startTime = System.nanoTime();

    BigInteger[] numero = MetodosUtiles.byteToInt(secretKey);
    addDesc("Clave en enteros: ");
    addDesc(MetodosUtiles.enEnteros(numero));

    BigInteger[] cifrado = new BigInteger[numero.length];

    for (int i = 0; i < numero.length; i++) {
        Constantes.cont++;
        cifrado[i] = numero[i].modPow(claves.getPrivadaD().getD(), claves.getN());
    }

    long end = System.nanoTime();
    long elapsedTimeNs = System.nanoTime() - startTime;
    System.out.println("*************************");
    System.out.println(elapsedTimeNs + " ns
");
    System.out.println("*************************");

    addDesc("Cifrado en enteros :");
    addDesc(MetodosUtiles.enEnteros(cifrado));

    return MetodosUtiles.intToByteArray(cifrado);
}

/**
 * Encrypt the AES Key with the public key E
 *
 * @param secretKey
 * @return
 */

public ArrayList<byte[]> cifrarConPublica(ArrayList<byte[]> secretKey) {

addDesc("Clave en bytes Array: ");
addDesc(MetodosUtiles.enByteArray(secretKey));

long startTime = System.nanoTime();

BigInteger[] texto = MetodosUtiles.byteArrayToInt(secretKey);
addDesc("Clave en enteros: ");
addDesc(MetodosUtiles.enEnteros(texto));

BigInteger[] cifrado = new BigInteger[texto.length];

for (int i = 0; i < texto.length; i++) {
    Constantes.cont++;
    cifrado[i] = texto[i].modPow(claves.getPublicaE().getE(), claves.getN());
}

long end = System.nanoTime();
long elapsedTimeNs = System.nanoTime() - startTime;
System.out.println("***************************");
System.out.println(elapsedTimeNs + " ns\n");
System.out.println("***************************");

addDesc("Cifrado en enteros: ");
addDesc(MetodosUtiles.enEnteros(cifrado));
return MetodosUtiles.intToByteArray(cifrado);

/**
 * Decrypts with the Private D to obtain the AES 128 bit key
 * @param claveCifrada2
 * @return
 */
public ArrayList<byte[]> decifraConPrivada(ArrayList<byte[]> claveCifrada2) {

    addDesc("Cifrada2 en bytes Array: ");
    addDesc(MetodosUtiles.enByteArray(claveCifrada2));

    long startTime = System.nanoTime();

    BigInteger[] cifrada2 = MetodosUtiles.byteArrayToInt(claveCifrada2);
    addDesc("Cifrada2 en enteros: ");
addDesc(MetodosUtiles.enEnteros(cifrada2));

System.out.println(desc());

BigInteger[] cifrada = new BigInteger[claveCifrada2.size()];

for (int i = 0; i < cifrada2.length; i++) {
    Constantes.cont++;
    cifrada[i] = cifrada2[i].modPow(claves.getPrivadaD().getD(), claves.getN());
}

long end = System.nanoTime();
long elapsedTimeNs = System.nanoTime() - startTime;

System.out.println("\n∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗∗\n");
//System.out.println(res);
System.out.println(elapsedTimeNs + " ns\n");
addDesc("Decifrado en enteros : ");
addDesc(MetodosUtiles.enEnteros(cifrada));

return MetodosUtiles.intToByteArray(cifrada);
}

/**
 * Decrypt with Public E, to obtaing the AES 128 bit key.
 * @param claveCifrada
 * @return
 */
public byte[] decifraConPublica(ArrayList<byte[]> claveCifrada) {

    long startTime = System.nanoTime();

    BigInteger[] cifrada = MetodosUtiles.byteArrayToInt(claveCifrada);
    addDesc("Cifrada en enteros: ");
    addDesc(MetodosUtiles.enEnteros(cifrada));

    BigInteger[] secretKey = new BigInteger[claveCifrada.size()];

    for (int i = 0; i < cifrada.length; i++) {
        Constantes.cont++;
    }
The class ClavePrivada.java is used for generating RSA private key algorithm.

```java
package logica.rsa;

import java.math.BigInteger;

/**
 * Generating RSA private key algorithm
 * @author Luis H Santamaria
 */
public class ClavePrivada {

    secretKey[i] = cifrada[i].modPow(claves.getPublicaE().getE(), claves.getN());

    long end = System.nanoTime();
    long elapsedTimeNs = System.nanoTime() - startTime;

    System.out.println("************************");
    System.out.println(elapsedTimeNs + " ns\n");
    System.out.println("************************");

    addDesc("Decifrado en enteros: ");
    addDesc(MetodosUtiles.enEnteros(secretKey));

    byte[] texto = MetodosUtiles.intToByte(secretKey);

    addDesc("Decifrado en bytes: ");
    addDesc(MetodosUtiles.enBytes(texto));

    return texto;
}

public String getDesc() {
    return desc;
}

public void addDesc(String s) {
    this.desc += s;
}

The class ClavePrivada.java is used for generating RSA private key algorithm.
```

```java
```
The class ClavePublica.java is used for generating RSA public key algorithm.

```java
package logica.rsa;

import java.math.BigInteger;

/**
 * Generating RSA public key algorithm
 * @author Luis H Santamaria
 */
public class ClavePublica {

    private BigInteger e;

    public ClavePublica(BigInteger e) {
        this.e = e;
    }

    public BigInteger getE() {
        return e;
    }

    public byte[] getEncoded() {
        return e.toByteArray();
    }
}
```

ParDeClaves.java contains private and public key of RSA algorithm
package logica rsa;

import java.math.BigInteger;

/**
 * Contains private and public key of RSA algorithm
 *
 * @author Luis H Santamaria
 */
public class ParDeClaves {

    private final BigInteger n;
    private final BigInteger toTient;
    private ClavePrivada privadaD;
    private ClavePublica publicaE;

    public ParDeClaves(BigInteger privadaD, BigInteger publicaE, BigInteger n) {
        this.n = n;
        this.toTient = new BigInteger("0");
        this.privadaD = new ClavePrivada(privadaD);
        this.publicaE = new ClavePublica(publicaE);
        privadaD.toByteArray();
    }

    public ParDeClaves(BigInteger n, BigInteger toTient, BigInteger privadaD, BigInteger publicaE) {
        this.n = n;
        this.toTient = toTient;
        this.privadaD = new ClavePrivada(privadaD);
        this.publicaE = new ClavePublica(publicaE);
        privadaD.toByteArray();
    }

    public BigInteger getN() {
        return n;
    }

    public BigInteger getToTient() {
        return toTient;
    }

    public ClavePrivada getPrivadaD() {

The AES.java class is responsible for implementing all the logic of the AES algorithm. The Package javax.crypto is used which provides the classes and interfaces for cryptographic operations. We make use of Cipher and SecretKeySpec JAVA classes to encrypt and decrypt with AES.

```java
package logica.aes;

import java.security.InvalidKeyException;
import java.security.NoSuchAlgorithmException;
import javax.crypto.BadPaddingException;
import javax.crypto.Cipher;
import javax.crypto.IllegalBlockSizeException;
import javax.crypto.KeyGenerator;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.spec.SecretKeySpec;
import logica.nucleo.entidades.Mensaje;
import logica.util.Constantes;
import logica.util.MetodosUtiles;

/**
 * @author Luis H Santamaria
 */
public class AES {
    private byte[] clave;
}
```
/**
 * @param n
 */
public AES(int n) {
}

/**
 * @throws Exception
 */
public AES() throws Exception {
    generarClave();
}

/**
 * Generates the AES 128-bit Key
 * @throws Exception
 */
private void generarClave() throws Exception {
    try {
        KeyGenerator kgen = KeyGenerator.getInstance(Constantes.ALGORITMO_AES);
        kgen.init(Constantes.LONGITUD_CLAVE_AES);
        clave = MetodosUtiles.eliminarNegativos(kgen.generateKey().getEncoded());
    }
    catch (NoSuchAlgorithmException ex) {
        throw new Exception("NoSuchAlgorithmException - Error en la generacion de la clave AES - " + ex.getMessage());
    }
}

/**
 * Leave the key empty
 */
public void limpiarClave() {
    clave = new byte[0];
}

/**
 * Encrypt the message to send
 */
```java
public void cifrar(Mensaje textoACifrar) throws Exception {
    try {
        byte[] raw = getClave();
        SecretKeySpec skeySpec = new SecretKeySpec(raw, Constantes.ALGORITMO_AES); // Create an instance of the class SecretKeySpec java and associates to the algorithm.
        Cipher cipher = Cipher.getInstance(Constantes.ALGORITMO_AES); // This class is responsible for encrypting and decrypting with AES
        cipher.init(Cipher.ENCRYPT_MODE, skeySpec); // Encrypt mode is selected
        byte[] encrypted = cipher.doFinal(textoACifrar.getMensajeclearByte()); // RETURN THE RESULT: Settlement of bytes encrypted
        textoACifrar.setMensajeCifradoByte(encrypted);
    } catch (NoSuchAlgorithmException ex) {
        throw new Exception("NoSuchAlgorithmException " + ex.getMessage());
    } catch (NoSuchPaddingException ex) {
        throw new Exception("NoSuchPaddingException " + ex.getMessage());
    } catch (InvalidKeyException ex) {
        throw new Exception("InvalidKeyException " + ex.getMessage());
    } catch (IllegalBlockSizeException ex) {
        throw new Exception("IllegalBlockSizeException " + ex.getMessage());
    } catch (BadPaddingException ex) {
        throw new Exception("BadPaddingException " + ex.getMessage());
    }
}

/**
 * Decrypts the message with a 128-bit AES key
 * @param textoADecifrar
 * @throws NoSuchAlgorithmException
 * @throws NoSuchPaddingException
 */
```
```java
public void decifrar(Mensaje textoADecifrar) throws NoSuchAlgorithmException, NoSuchPaddingException, InvalidKeyException, IllegalBlockSizeException, BadPaddingException {
    byte[] raw = getClave();
    SecretKeySpec skeySpec = new SecretKeySpec(raw, Constantes.ALGORITMO_AES); // Create an instance of the class SecretKeySpec java and associates to the algorithm.
    Cipher cipher = Cipher.getInstance(Constantes.ALGORITMO_AES);//This class is responsible for encrypting and decrypting with AES cipher.init(Cipher.DECRYPT_MODE, skeySpec);//Decrypt mode is selected
    byte[] decifrado = cipher.doFinal(textoADecifrar.getMensajeCifradoByte()); //RETURN THE RESULT: Settlement of bytes decrypted
    textoADecifrar.setMensajeClaroByte(decifrado);
}

/**
 * @param clave
 */
public void setClave(byte[] clave) {
    this.clave = clave;
}

/**
 * @return
 */
public byte[] getClave() {
    if (clave == null) {
        return new byte[0];
    }
    return clave;
}
```
CifradoAES_RSA.java contains all the application logic. All methods for sending and receiving operation with RSA (twice) and AES.

```java
package logica.nucleo.entidades;

import java.util.ArrayList;

/**
 * @author Luis H Santamaria
 */
public class CifradoAES_RSA {

    private Parametros parametros;

    public CifradoAES_RSA(ClavesReceptor receptor) {
        parametros = new Parametros();
        parametros.inicializarParametros(receptor);
    }

    public CifradoAES_RSA() throws Exception {
        parametros = new Parametros();
        parametros.inicializarParametros();
    }

    public String dobleCifrado() {
        String s = "Cifrar con Privada A\n";
        cifrarConPrivada_A();
        s += getParametros().getRsa_ParA().getDesc();

        s += "-------------------\n";
        s += "Cifrar con Publica B\n";
        cifrarConPublica_B();
        s += getParametros().getRsa_ParB().getDesc();
        return s;
    }

    public String dobleDecifrado() {
        String s = "-------------------\n";
        s += "Decifrar con Privada B\n";
        decifraConPrivada_B();
        s += getParametros().getRsa_ParB().getDesc();
        s += "-------------------\n";
        s += "Decifrar con Publica A\n";
```
decifraConPublica_A();
s += getParametros().getRsa.ParA().getDesc();
return s;
}

/**
 * Encrypt with AES
 * @throws Exception
 */
public void cifrarAES() throws Exception {
    parametros.getAes().cifrar(parametros.getMensage());
}

/**
 * Decrypt with AES
 * @throws Exception
 */
public void decifrarAES() throws Exception {
    parametros.getAes().decifrar(parametros.getMensage());
}

/**
 *
 */
public void cifrarConPrivada_A() {
    byte[] clave = parametros.getAes().getClave();
    ArrayList<byte[]> claveCifrada = parametros.getRsa.ParA().cifrarConPrivada(clave);
    System.out.println("Clave cifrada");
    System.out.println(parametros.getRsa.ParA().getDesc());
    mostrar(claveCifrada);
    parametros.getMensage().setClaveCifrada(claveCifrada);
}

/**
 */
public void cifrarConPublica_B() {
    ArrayList<byte[]> claveCifrada = parametros.getMensage().getClaveCifrada();
    claveCifrada2 = parametros.getRsa.ParB().cifrarConPublica(claveCifrada);
System.out.println("Clave cifrada2");
System.out.println(parametros.getRsa_ParB().getDesc());
mostrar(claveCifrada2);
parametros.getMensaje().setClaveCifrada(claveCifrada2);
}

/**

 *
 */
public void decifraConPrivada_B() {
    ArrayList<Byte> claveCifrada2 = parametros.getMensaje().getClaveCifrada();
    ArrayList<Byte> claveCifrada = parametros.getRsa_ParB().decifraConPrivada(claveCifrada2);
    System.out.println("Clave cifrada");
    System.out.println(parametros.getRsa_ParB().getDesc());
    mostrar(claveCifrada);
    parametros.getMensaje().setClaveCifrada(claveCifrada);
}

/**

 *
 */
public void decifraConPublica_A() {
    ArrayList<Byte> claveCifrada = parametros.getMensaje().getClaveCifrada();
    byte[] clave = parametros.getRsa_ParA().decifraConPublica(claveCifrada);
    System.out.println("Clave cifrada");
    System.out.println(parametros.getRsa_ParB().getDesc());
    mostrar(claveCifrada);
    parametros.getAes().setClave(clave);
}

/**

 * @return
 */
public Parametros getParametros() {
    return parametros;
}

/**

 *
In ClavesReceptor.java we can find the keys retrieved from the trusted database, for the message that recipient wants to decipher.

```java
package logica.nucleo.entidades;

import java.math.BigInteger;
import java.util.ArrayList;

/**
 * @author Luis H Santamaria
 */
public class ClavesReceptor {

    private ArrayList<byte[]> claveCif;
    private byte[] mensajeCifradoByte;
    private BigInteger aPublica;
    private BigInteger aPrivada;
    private BigInteger bPublica;
    private BigInteger bPrivada;
    private BigInteger AN;
    private BigInteger bN;
```
public ClavesReceptor (ArrayList<byte[]> clave, byte[] mensajeCifradoByte, BigInteger aPublica, BigInteger aPrivada, BigInteger bPublica, BigInteger bPrivada, BigInteger aN, BigInteger bN) {
    this.claveCif = clave;
    this.mensajeCifradoByte = mensajeCifradoByte;
    this.aPublica = aPublica;
    this.aPrivada = aPrivada;
    this.bPublica = bPublica;
    this.bPrivada = bPrivada;
    this.aN = aN;
    this.bN = bN;
}

public BigInteger getaN() {
    return aN;
}

public void setaN (BigInteger aN) {
    this.aN = aN;
}

public BigInteger getbN() {
    return bN;
}

public void setbN (BigInteger bN) {
    this.bN = bN;
}

public ArrayList<byte[]> getClave () {
    return claveCif;
}

public void setClave (ArrayList<byte[]> clave) {
    this.claveCif = clave;
}

public byte[] getMensajeCifradoByte () {
    return mensajeCifradoByte;
}

public void setMensajeCifradoByte (byte[] mensajeCifradoByte) {
    this.mensajeCifradoByte = mensajeCifradoByte;
}
Token.java class represents the security token application using a USB device.

```java
package logica.token;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.File;
import java.io.IOException;
```
import java.util.NoSuchElementException;
import java.util.StringTokenizer;

/**
 * @author Luis H Santamaria
 */
public class Token {

    private static File URL;
    private static String Email, Password, Names;
    private static String Descr;

    // valida si se encuentra el token
    public Token() throws Exception {
        do {
            while (URL == null) {
                URL = BuscarToken();
            }
        } while (!ValidarToken());
    }

    public String getEmail() {
        return Email;
    }

    // Verifies that the Token is there
    public void verificaToken() throws Exception {
        File url2 = BuscarToken();

        if (url2 != null) {
            if (!ValidarToken()) {
                throw new Exception("El token no es valido");
            }
        } else {
            // Handle null URL case
        }
    }
}
throw new Exception("El token no es valido");
}
}

public String getPassword() {
    return Password;
}

public String getNames() {
    return Names;
}

// This method verify if the Token have the user data need it.
private boolean ValidarToken() throws Exception {
    BufferedReader brtk = null;
    try {
        String line;
        brtk = new BufferedReader(new FileReader(URL));
        while ((line = brtk.readLine()) != null) {
            StringTokenizer LineaToken = new StringTokenizer(line, "|");
            while (LineaToken.hasMoreElements() && LineaToken != null) {
                try {
                    Email = LineaToken.nextElement().toString();
                    Password = LineaToken.nextElement().toString();
                    Names = LineaToken.nextElement().toString();
                } catch (NoSuchElementException eq) {
                    String error = "Token Invalido\n" + "Ejecute de nuevo la aplicacion con un Token Valido\n" + eq.getMessage();
                    throw new Exception(error);
                }
            }
        }
    } catch (FileNotFoundException fnfe) {
        throw new Exception(fnfe.getMessage());
    }
    return false;
}

public String getNames() {
    return Names;
}

// This method verify if the Token have the user data need it.
private boolean ValidarToken() throws Exception {
    BufferedReader brtk = null;
    try {
        String line;
        brtk = new BufferedReader(new FileReader(URL));
        while ((line = brtk.readLine()) != null) {
            StringTokenizer LineaToken = new StringTokenizer(line, "|");
            while (LineaToken.hasMoreElements() && LineaToken != null) {
                try {
                    Email = LineaToken.nextElement().toString();
                    Password = LineaToken.nextElement().toString();
                    Names = LineaToken.nextElement().toString();
                } catch (NoSuchElementException eq) {
                    String error = "Token Invalido\n" + "Ejecute de nuevo la aplicacion con un Token Valido\n" + eq.getMessage();
                    throw new Exception(error);
                }
            }
        }
    } catch (FileNotFoundException fnfe) {
        throw new Exception(fnfe.getMessage());
    }
    return false;
}
if (Email != null && Password != null && Names != null) {
    addDesc("Token Valido\n");
    return true;
} else {
    String error = "Token Invalido\n" + "Ejecute de nuevo la aplicacion con un Token Valido\n";
    throw new Exception(error);
}

} catch (IOException e) {
    throw e;
} finally {
    try {
        if (brtk != null) {
            brtk.close();
        }
    } catch (IOException ex) {
        throw ex;
    }
}

// Looks for the .tk file inside of the USB
public static File BuscarToken() {

    File unidades[] = File.listRoots(), files[] = null;
    for (int i = 0; i < unidades.length; i++) {
        // addDesc(unidades[i]);
        if ("/".equals(unidades[i].toString())) {
            //addDesc("-linux");
            unidades = unidades[i].listFiles();
        } else if ("/media".equals(unidades[i].toString())) { //
            unidades = unidades[i].listFiles();
        }
    }
}
HiloVerificarToken.java is the thread responsible for verifying the token is inside the system. This validates if the user leaves and close the application.

```java
package logica.token;

import gui.Administrador;
import static java.lang.Thread.sleep;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * @author Luis H Santamaria
 */
public class HiloVerificaToken extends Thread {

    Administrador admin;

    public HiloVerificaToken(Administrador admin) {
        this.admin = admin;
    }
```
The Mail.java class represents an electronic email. It make uses of the Java.Mail Package which provides a platform-independent and protocol-independent framework to build mail and messaging applications.

```java
package logica.mail.client.libs;

import javax.mail.Address;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
import logica.util.Constantes;

/**
 * @author Luis H Santamaria
 */
public class Mail {

    private static int Id;
    private static int Id_Bd;
    private static String Asunto, Mensaje, Cifrado;
    private static Address [] De, Para;

    public void run() {
        try {
            do {
                try {
                    admin.getToken().verificaToken();
                } catch (Exception ex) {
                    Logger.getLogger(HiloVerificaToken.class.getName()).log(Level.SEVERE, null, ex);
                    System.exit(0);
                }
                sleep(1 * 1000); // The thread is sleep for 30 seconds
            } while (true);
        } catch (InterruptedException ex) {
            Logger.getLogger(HiloVerificaToken.class.getName()).log(Level.SEVERE, null, ex);
        }
    }
}
```
public Mail(String From, String To, String Subject, String Body, String encrypt) {
    // Constructor Mail to sent.

    try {
        Address[] AFrom = {new InternetAddress(From)};
        Address[] ATo = {new InternetAddress(To)};

        De = AFrom;
        Para = ATo;

        Asunto = Subject;
        Id_Bd = 0;

        Mensaje = Body;
        Cifrado = encrypt;
    } catch (AddressException e) {
    }
}

public Mail(int Number, Address[] From, Address[] To, String Subject, Object Body) {
    // Constructor Mail to read

    Id = Number;
    De = From;
    Para = To;
    Asunto = Subject;
    Id_Bd = -1;
    Mensaje = Body.toString();
    Cifrado = null;
}

public Mail(boolean bol, int Number, Address[] From, Address[] To, String Subject, Object Body) {
    // Constructor Mail to read

    Id = Number;
}
De = From;
Para = To;

try {
    String tokens[] = Subject.split(Constance\nSEPARADOR ASUNTO);
    Asunto = tokens[0];
    Id_Bd = Integer.parseInt(tokens[1]);
} catch (Exception e) {
    System.out.println("Revise el Asunto del Mensaje " + e.
getMessage());
}

Mensaje = Body.toString();
Cifrado = null;

}  

public Message MailtoSend(Session sesion) {
    // Function to retrieve mail sent

    Message mensaje = new MimeMessage(sesion);

    try {
        mensaje.setFrom(De[0]);

        Address[] receptor es = new Address[Para.length];
        i nt j = 0;
        while (j < Para.length) {
            receptor es[j] = Para[j];
            j++;
        }

        mensaje.setRecipients(Message.RecipientType.TO,
receptor es);
        mensaje.setSubject(Asunto);
        mensaje.setText(Mensaje);

    } catch (MessagingException e) {
        throw new RuntimeException(e);
    }

    return mensaje;
```java
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append("\nNumero: 
").append(Id);
    sb.append("\nDe: ").append(De[0].toString());
    sb.append("\nAsunto: ").append(Asunto);
    return sb.toString();
}

/**
 * @return
 */
public Integer getIdMensaje() {
    return Id_Bd;
}

public String getMensaje() {
    return Mensaje;
}

public String getMensajeCif() {
    return Cifrado;
}

public static String getAsunto() {
    return Asunto;
}

public void setId_Bd(int Id_Bd) {
    Mail.Id_Bd = Id_Bd;
}

public void setAsunto(String Asunto) {
    Mail.Asunto = Asunto;
}
```
MetodosUtiles.java is a class that implements the most used methods in the application for different classes.

```java
package logica.util;

import java.math.BigInteger;
import java.util.ArrayList;
import logica.nucleo.entidades.ClavesReceptor;
import persistencia.entities.Clave;

/**
 * @author Luis H Santamaria
 */
public class MetodosUtiles {

    /**
     * Get an array of bytes, each byte and converts it to an integer
     * and integer to a hex. Then returns all hexa in a text. Converts the byte in
     * hexadecimal.
     *
     * @param arregloEncriptado
     * @return
     */
    public static String bytesToHexa(byte[] arregloEncriptado) {

        String textoEncriptado = "";
        for (int i = 0; i < arregloEncriptado.length; i++) {
            int aux = arregloEncriptado[i] & 0xff;
            if (aux < 16) {
                textoEncriptado = textoEncriptado.concat("0");
            }
            textoEncriptado = textoEncriptado.concat(Integer.toHexString(aux));
        }
        return textoEncriptado;
    }

    /**
     * Receive a text with several hexadecimal. Then it remove and convert into
     */
}
an integer, and finally becomes in bytes. Take each byte and stores it in
an array.

@method hexaToBytes

/****

    public static byte[] hexaToBytes(String encriptado) {
        byte[] enBytes = new byte[encriptado.length() / 2];
        for (int i = 0; i < enBytes.length; i++) {
            int index = i * 2;
            String aux = encriptado.substring(index, index + 2);
            int v = Integer.parseInt(aux, 16);
            enBytes[i] = (byte) v;
        }
        return enBytes;
    }

****/

public static BigInteger[] byteToInt(byte[] bytes) {
    // for (int i = 0; i < bytes.length; i++) {
    //     byte[] b = new byte[1];
    //     b[0] = bytes[i];
    //     enteros[i] = new BigInteger(b);
    // }
    return enteros;
}

public static BigInteger[] byteArrayToInt(ArrayList<byte[]> bytes) {
    // for (int i = 0; i < bytes.size(); i++) {
    //     //enteros[i] = new BigInteger(bytes[i]);
    //     enteros[i] = new BigInteger(bytes.get(i));
    // }
    return enteros;
}

public static byte[] intToByte(BigInteger[] enteros) {

byte[] bytes = new byte[enteros.length];

for (int i = 0; i < bytes.length; i++) {
    bytes[i] = enteros[i].byteValue();
}

return bytes;

public static ArrayList<byte[]> intToByteArray(BigInteger[] enteros) {
    ArrayList<byte[]> bytes = new ArrayList<>();

    for (int i = 0; i < enteros.length; i++) {
        byte[] ba = enteros[i].toByteArray();
        bytes.add(ba);
    }

    return bytes;
}

public static ArrayList<byte[]> StringToBytesArray(String claveCif) {
    String[] s = claveCif.split(" ",");
    ArrayList<byte[]> clavecifrada2 = new ArrayList<>();

    for (String string : s) {
        byte[] bs = hexaToBytes(string);
        clavecifrada2.add(bs);
    }

    return clavecifrada2;
}

public static byte[] eliminarNegativos(byte[] bytes) {
    for (int i = 0; i < bytes.length; i++) {
        if (Byte.valueOf(bytes[i]).compareTo(Byte.valueOf("0")) < 0) {
            // si es negativo
            int pos = Byte.valueOf(bytes[i]).intValue() * (-1);
            bytes[i] = Byte.valueOf("" + pos);
        }
    }

    return bytes;
}

public static String enBytes(byte[] bytes) {

```java
String s = "";
for (byte b : bytes) {
    s += b + " ";
}
s += "\n";
return s;
}

public static String enByteArray(ArrayList<byte[]> bytes) {
    String s = "";
    for (byte b[] : bytes) {
        for (byte c : b) {
            s += c + " ";
        }
    }
s += "\t";
    s += "\n";
    return s;
}

public static String enEnteros(BigInteger[] enteros) {
    String s = "";
    for (BigInteger b : enteros) {
        s += b + " ";
    }
s += "\n";
    return s;
}

public static ClavesReceptor convertirClaveRec(Clave clave, String menCifrado) {
    ArrayList<byte[]> clavecif = StringToBytesArray(clave.getAesCif());
    byte[] mensajeCifradoByte = hexaToBytes(menCifrado.trim());
    BigInteger aPublica = new BigInteger(hexaToBytes(clave.getAPublica()));
    BigInteger aPrivada = new BigInteger(hexaToBytes(clave.getAPrivada()));
    BigInteger bPublica = new BigInteger(hexaToBytes(clave.getBPublica()));
    BigInteger bPrivada = new BigInteger(hexaToBytes(clave.getBPrivada()));
    BigInteger an = new BigInteger(hexaToBytes(clave.getAN()));
    BigInteger bn = new BigInteger(hexaToBytes(clave.getBN()));
```
return new ClavesReceptor(claveCif, mensajeCifradoByte, aPublica, aPrivada, bPublica, bPrivada, an, bn);

}