OTP and AES: A historical transition between two systems of cryptography

Valdemar Thanner
Kantonsschule Zug
Supervised by Mr. Bernhard Keller
Linguistic supervision by Ms. Margrit Oetiker

December 20, 2016

Contents

Chapter 1

OTP: The One Time Pad

1.1 What is a "One Time Pad"?

When speaking about OTP, it is important to distinguish between its two meanings: On the one hand, it is a technique used to encrypt information. This technique requires one single key, used both to encrypt and decrypt the information. This key is also referred to as a one time pad; therefore, it is important to distinguish between the one time pad (a cryptographical technique) and a one time pad (a key which is used to encrypt and decrypt information).

The One Time Pad is largely derived from the Vernam cipher, which is named after Gilbert Vernam. The Vernam cipher utilized a perforated tape (one of the earliest types of data storage) as the secret key[1]. Each bit of data was stored in the form of a hole punched into the perforated tape.

Figure 1.1: Perforated tape, utilized to store bits as punched holes

However, this system had a vulnerability which the One-Time Pad solved: In Vernam's original method, the perforated tape was not exchanged after it had completed one cycle; instead, it was looped around continuously, often being used to encrypt multiple different messages.

This made the entire system vulnerable. The re-usage of the key meant that the resulting ciphertext suffered from a so-called known-plaintext vulnerability [2]. This means that, if a plaintext and its corresponding ciphertext are captured, the key utilized to generate the ciphertext can be derived from them. This is not an issue if the key is exchanged each time a new message is encrypted. However, if the key of any Vernam cipher machine was compromised through a known-plaintext attack, any further intercepted ciphertexts could be decrypted.

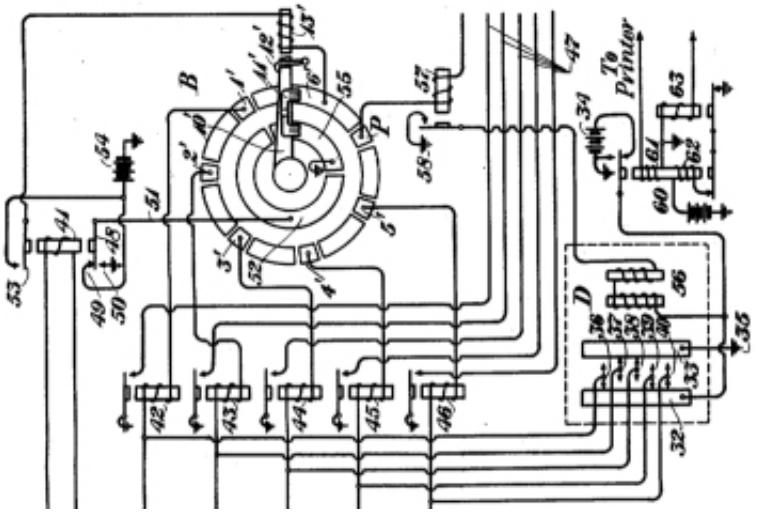


Figure 1.2: A technical diagram from Vernam's famous "Secret signaling system" patent of 1919.

1.2 Method used

In the following section, the utilized method will be clarified through usage of an example. In this example, the message *"cryptography"* will first be encrypted by its sender, sent to its intended recipient, and finally decoded by the recipient.

1.2.1 Generation of the random key

In order to encrypt the plaintext, a key must first be generated. This key will be utilized to encrypt the plaintext through the usage of modular addition, turning it into the ciphertext.

This key must fulfil some crucial criteria [3]. Foremost, the length of the key (the amount of characters contained within it) must be equivalent to or greater than the length of the plaintext; otherwise, it is not possible to per-

form any encryption (using the OTP). Secondly, the key must be generated randomly. This is mainly due to the fact that a randomly generated key makes frequency analysis[4], the form of cryptanalysis most commonly used to break classical ciphers, impossible.

The key consists of numbers. Usually, when the plaintext is made up of Latin letters, the numbers range between 0 and 25. The key can be converted into Latin letters through the same method applied to the plaintext outlined in the following chapter, however, this is not necessary, although the key is often transported in the form of text.

As the message being encrypted in this example has 12 characters, the key must also posses at least 12 characters. For the sake of this example, the key *"sytruifgnihm"* will be utilized.

1.2.2 Modular addition of the key and plaintext

Next, the ciphertext is created through modular addition of the key and the plaintext. This can be applied not only to a message consisting of alphabetical characters, but also to any sequence of bits. If the plaintext consists of a message made up of alphabetical characters, the plaintext and the key are added using arithmetic referred to as *"addition modulo 26"*. The correct mathematical notation for modular arithmetic is $(a + b) \, mod \, c$, where c is referred to as the *modulus*, which is the value that cannot be passed nor reached in modular addition. In order to perform modular addition, the variables a and b are first added, after which they are divided by the modulus c, up to an integer. The resulting remainder r is the final result of the operation.

Before the modular addition of the plaintext and the key can begin, each character (of the plaintext as well as of the secret key, in the case that the key was generated as a string of Latin letters instead of as a sequence of numbers) must be converted to a number, corresponding to it's position in the Latin alphabet:

c	r	y	p	t	o	g	r	a	p	h	y
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
2	17	24	15	19	14	6	17	0	15	7	24

As the key was also generated in the form of characters, it too must be converted to a sequence of numbers:

s	y	t	r	u	i	f	q	n	i	h	m
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
18	24	19	17	20	8	5	16	13	8	7	12

Afterwards, the key and the plaintext are added together utilizing modular addition. Of course, all operations below are performed in *mod 26*. Finally, the resulting numbers are converted to the character to which they correspond in the Latin alphabet. The resulting sequence of characters is referred to as the ciphertext.

2+18	17+24	24+19	15+17	19+20	14+8	6+5	17+16	0+13	15+8	7+7	24+12
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
20	15	17	6	13	22	11	7	13	23	14	10
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
u	p	r	g	n	w	l	h	n	x	o	k

The modular addition of the plaintext and the key can also, however, be performed bitwise. This simply means that, instead of adding a message consisting of (up to) 26 different letters with a key (also consisting of up to 26 different letters), we simply add the bits of the plaintext (a computer file consisting of bits), which can assume the value of either 1 or 2, with the bits of the key. The bits are added in modulo 2. This process is referred to as XOR, one of the basic bitwise operations which can be performed directly by a computers central processor.

0	1	0	1
+ (mod 2)	+ (mod 2)	+ (mod 2)	+ (mod 2)
0	0	1	1
↓	↓	↓	↓
0	1	1	0

Figure 1.3: The formula used by an XOR gate.

1.2.3 Decoding of the ciphertext using the key

Assuming the message has been delivered to the intended recipient, who must hold a copy of the secret key (which, as OTP is a symmetrical cryptographical method, is identical to the one utilized to create the ciphertext), the recipient can now decode the ciphertext in order to view the plaintext.

Since the ciphertext was created through the modular addition of the plaintext and the key, the recipient can utilize modular subtraction in order to view the plaintext. In order to do this, the recipient must subtract the key from the ciphertext in modulo 26, and convert the resulting numbers to Latin characters. However, it is now also necessary for the numbers not to become negative. Fortunately, this is also made possible by modular arithmetic, as the values simply loop back around from 0 as well. Once again, all below operations are in *mod 26*.

20-18	15-24	17-19	6-17	13-20	22-8	11-5	7-16	13-13	23-8	14-7	10-12
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
2	17	24	15	19	14	6	17	0	15	7	24
↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓	↓
c	r	y	p	t	o	g	r	a	p	h	y

Now, the message's journey is complete, having passed from plaintext into ciphertext, being transported to its intended recipient in the form of ciphertext, and finally being decoded and read by its recipient.

1.3 Perfect secrecy: Information-theoretical security

An important characteristic of perfect secrecy is that the ciphertext conveys no information at all in regards to the plaintext; this means that it is not possible to garner any information about the key nor the plaintext if one is in possession of the ciphertext [5] [6]. In probability theory, where C contains all possible ciphertexts and M contains all possible messages:

$forc \in C :$
$\forall m_1, m_2 \in M :$
$P(E_k(m_1) = c) = P(E_k(m_2) = c)$

E_k is a defined as a cipher using the key k if there exists a function $D_k(m) = c$, also using the key k, which decrypts each ciphertext to a unique message.

From this, two other important characteristics of a cryptographical system with perfect secrecy can be derived. Firstly, perfect message indistinguishability. This means that, even if you are provided with multiple different plaintexts and one ciphertext, it is impossible to assign any one plaintext to the provided ciphertext. Secondly, if the key is the same length as the plaintext (as is the case in OTP), then there exists a key for every possible encryption from plaintext to ciphertext. This means that any plaintext can be converted to any ciphertext, making it impossible to determine the key without access to both the plaintext and the ciphertext. This is referred to as perfect key ambiguity.

Although many believe that the ciphertext does provide information about the plaintext, namely its length, this can easily be solved through padding, where characters of no importance to the plaintext are added to it before encryption [2].

1.3.1 Mathematical proof

In 1949, Shannon proved that OTP did, in fact, posess perfect secrecy[7]. First, the properties and contents of the sets and variables used in this proof must be clarified:

Definition 1.3.1. Let M contain all possible messages; meaning all possible bit sequences.

Definition 1.3.2. Let m be the plaintext. $m \in M$.

Definition 1.3.3. Let c be the ciphertext.

Definition 1.3.4. Let K contain all possible keys.

Definition 1.3.5. Let k be the key. $k \in K$.

Definition 1.3.6. Let the function $E_k(m) = c$. The function E is an encryption process using the key k, which outputs the ciphertext c using the plaintext m.

Definition 1.3.7. Let m^* be a possible message, selected at random using a uniform distribution: $\forall m^* \in M : P(m^*) = \frac{1}{|M|}$

Definition 1.3.8. Let k^* be a possible key, selected at random using a uniform distribution. $\forall k^* \in M : P(k^*) = \frac{1}{|K|}$

Lemma 1.3.9. *In order for perfect secrecy to apply, then:*

$P(m = m^* \mid E(m) = c) = P(m = m^*)$ *must be true.*

Remark 1.3.10. This is because, if the probability that the actual plaintext m is equal to some other message m^*, given the ciphertext c, is equal to the probability that $m = m^*$, then the ciphertext c provides no additional information to the attacker. The ciphertext not providing any information about the plaintext nor the key is the definition of perfect secrecy; therefore, if the above equation can be proven to be true in the case of OTP, then the perfect secrecy of OTP will also have been proven.

Theorem 1.3.11. $P(m = m^* \mid E(m) = c) = P(m = m^*)$ *is true in the case of OTP.*

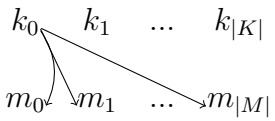
Proof. Per the definition of conditional probability,

$P(m = m^* \mid E_k(m) = c) = \frac{P(m=m^* \cap E_k(m)=c)}{P(E_k(m)=c)}$.

First, the computation of $P(E_k(m) = c)$:

Remark 1.3.12. The distribution of the messages is not uniform, as it must be assumed that a potential attacker has some form of knowledge about the nature of the message. Therefore, each message is not equally likely, meaning that $P(m = m^*) \neq \frac{1}{|M|}$. $P(E_k(m) = c)$ can also be taken as $P(k = k^*)$, because the probability of the function encrypting a certain message into a certain ciphertext hinges solely on the key. As the key is randomly generated, each possible key is equally likely. Therefore, $P(k = k^*) = P(E_k(m) = c) = \frac{1}{|K|}$. The mathematical computation of $P(E_k(m) = c)$ reaffirms this conclusion.

In order to find the probability that the given key k maps the given message m to the given ciphertext c, all probabilities that a certain key-message pair, k_i and m_i, encrypt to the given ciphertext c, are summed up. This summation needs to be divided by the probability space. Because each key is applied to each message, the probability space is $|M| * |K|$.



$$P(E_k(m) = c) = \sum_{\substack{m_i \in M \\ k_i \in K}} P(E_{k_i}(m_i) = c) * \frac{1}{|M|*|K|}$$

In OTP, there exists exactly one key which maps a specific message to a specific ciphertext. Therefore:

$$\sum_{k_i \in K} P(E_{k_i}(m_i) = c) = 1$$

$$\sum_{m_i \in M} 1 = |M|$$

$\Rightarrow P(E_k(m) = c) = \frac{|M|}{|M|*|K|} = \frac{1}{|K|}$

Remark 1.3.13. Note that, due to the above mentioned possibility that the attacker possesses some knowledge of the message, $P(m = m^*) \neq \frac{1}{|M|}$. Therefore, $P(m = m^*)$ cannot be further simplified. In order to find $P(m = m^* \cap E_k(m) = c)$, the definition of conditional probability as well as the property $P(E_k(m) = c) = \frac{1}{|K|}$ are used.

$P(m = m^* \cap E_k(m) = c) = \frac{P(m=m^*)}{|K|}$

Now, the values computed for $P(E_k(m) = c)$ and $P(m = m^* \cap E_k(m) = c)$ are inserted into the left hand side of the equation to see if it equals the right hand side of the equation.

$$\frac{\frac{P(m = m^*)}{|K|}}{\frac{1}{|K|}} = P(m = m^*)$$

$= RHS$ □

1.3.2 Consequences of the proof

The most important conclusion which Shannon drew from his proof was that, in order for a cipher to posess perfect secrecy, $|K| \geqslant |M|$[7]. This means that the key needs to be at least as long as the message. Note that using a key as long or longer than the message does not, however, constitute guaranteed perfect secrecy: It is merely an absolute necessity.

1.4 Issues with OTP

1.4.1 True randomness in generating the key

One of the most important aspects of the utilized key is that it must be completely random (see 1.2.1). However, generating a random sequence of numbers or letters is no trivial task. In fact, most random number generation functions of even modern programming languages are not adequately random for use in cryptography. This is especially difficult for computers, considering that a computer is, in essence, designed to follow a set of predictable instructions as quickly as possible.

Therefore, in order to generate a sufficiently random key, it is preferable to utilize hardware random number generators, for example by utilizing a true random noise source [3]. One such example is the key generator built by mils electronic, where a set of parallel ring oscillators are sampled. Because the oscillation speed of each ring is influenced by random factors such as local variations in temperature and voltage.

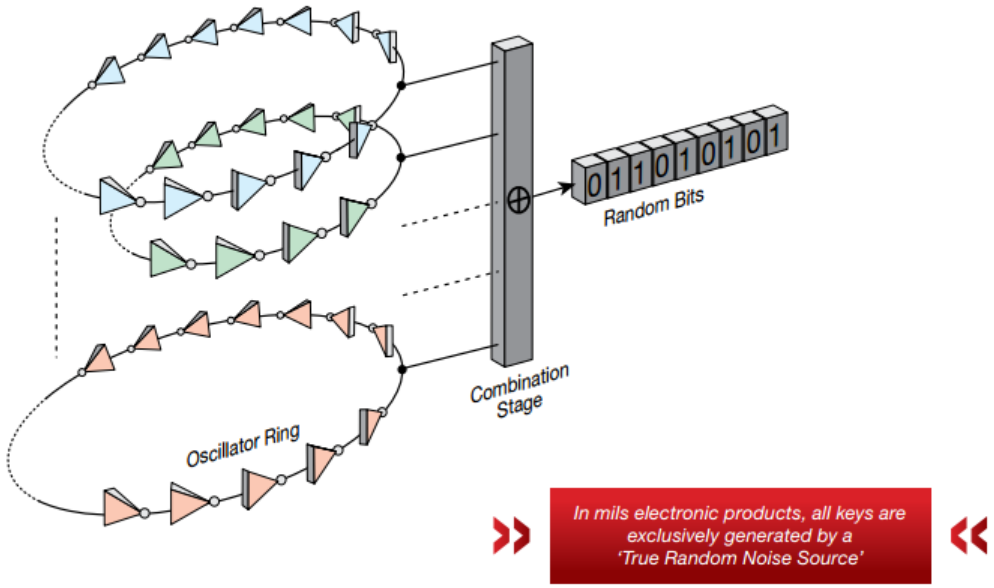


Figure 1.4: A hardware random number generator with sufficient randomness to be utilized in the generation of one-time pads.

1.4.2 Secure distribution of the key itself

The main issue in regards to OTP is that, while it posses perfect security, therefore solving the issue of safely transmitting the ciphertext, the problem instead becomes how to securely transmit the key to the intended recipient of the encrypted message.

Therefore, in order to ensure that the plaintext cannot be accessed by any party other than its intended recipient, the complete security of the transmittal of the key must also be ensured.

However, if the communicants already possess a method by which the key can be transmitted with infallible security, then it stands to reason that the plaintext itself could just as well be submitted through the same channel, considering that it has the same number of bytes (or characters) as the key. So, the problem of securely transmitting the plaintext has simply been trans-

ferred to the problem of securely transmitting the key.

One method of mitigating this issue is to first send one very long key, which can then be used for multiple messages in the future. Using this method, only one secure transmission has to be made in order to ensure the security of multiple transmissions.

1.4.3 Secure disposal of a utilized key

After a key has been fully utilized, it has to be disposed of securely, in order to prevent any information from being compromised in the future, which is possible should the key continue to exist. Therefore, it is necessary to completely eradicate the entirety of the key. If the key was stored on a digital media, such as a USB drive or HDD, the only completely secure method of disposal is considered to be complete incineration.

Chapter 2

AES: The Advanced Encryption Standard

2.1 AES viewed from a high level

Viewed from a high level, the AES algorithm consists of one XOR step, followed by 10 rounds of cryptographical measures, with each round as well as the initial XOR step using a specific subkey, generated using the main key. By passing through these steps, each block of plaintext is converted into a block of Ciphertext. In the following chapter, each individual aspect of the complete AES algorithm will be more closely examined.

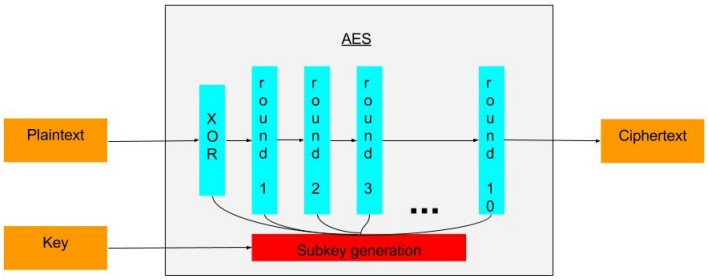


Figure 2.1: A high-level description of AES.

2.2 Encryption

Each round possesses 4 steps, explained in detail below. However, before those operations can be understood, some basic terminology must first be clarified.

The "state" is the matrix containing the information which is manipulated throughout the encryption process. As such, the state begins its journey through AES as the block of plaintext which is to be encrypted, and ends it as a block of ciphertext. The state is represented in a 4x4 column-major order matrix; this means that, in a block containing 16 bytes, the standard size of an AES plaintext block as well as the standard size of an AES key ($16bytes * 8 = 128bits$).

$$\begin{pmatrix} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{pmatrix}$$

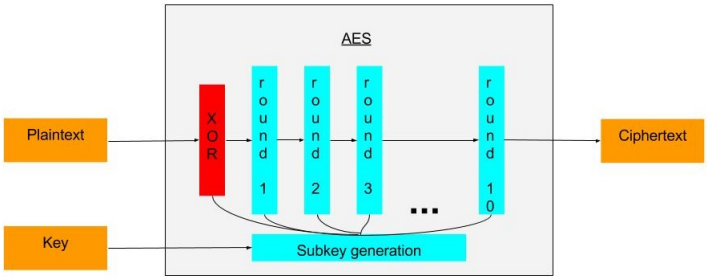
It is important to note that, because the state is a column-major order matrix, the bytes are counted from "top to bottom"; that is, in the direction of the columns, and not in the direction of the rows. This shows how the bytes from the plaintext are ordered into the state. However, in order to clarify the manipulations performed throughout the 10 rounds of AES, it is more beneficial to note the starting position each byte takes in the subscript instead of noting its chronological order in the plaintext block.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

The subkey possesses the same size as the state, and is made into a matrix in the exact same fashion; this is, of course, necessary for the successful modular

addition of the bytes contained within the state and those contained within the subkey using the XOR operation.

2.2.1 Initial XOR operation

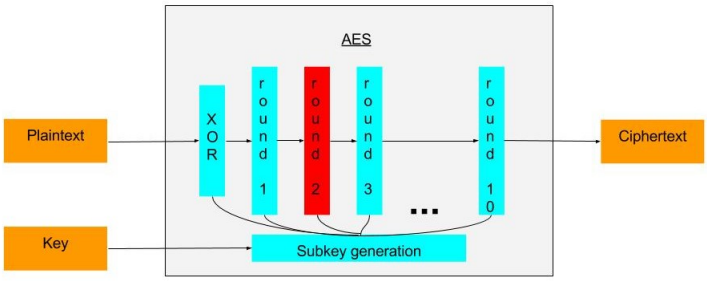


This step is identical to the XOR operation already described in the chapter regarding OTP; once again, a simple XOR gate is used to add the state with the initial XOR operations own subkey (which, crucially, is not the main key itself, but is, in fact, derived from the main key during the process of subkey generation).

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} + \begin{pmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{pmatrix}$$

Here, each byte of the state is combined with the byte of the subkey which possesses the corresponding position using the XOR operation. For example, $a_{2,1}$ would be combined with $k_{2,1}$. The resulting byte is then placed at the matching coordinate (in this example, once again the coordinate 2,1) of the resulting matrix. The completed matrix resulting from this operation is the new state. Next, this resulting state proceeds into the first of ten so-called "rounds".

2.2.2 Rounds



2.2.2.1 Galois Field $GF(2^8)$ arithmetic

Before the math behind AES can be fully understood, the Galois Field $GF(2^8)$ must first be more closely examined. This field contains only 256 values, ranging from 0-255; the range which can be stored in a byte. Secondly, all bytes are written using hexadecimal notation. Most importantly, all binary strings are represented as polynomials; this is best illustrated through an example [8].

$11001000 = x^7 + x^6 + x^3$

$10000111 = x^7 + x^2 + x + 1$

Addition is made rather simple through this; keeping in mind that, since addition is an XOR operation, two identical factors neutralize each other $(x^n + x^n = 0)$.

$(x^7 + x^6 + x^3) + (x^7 + x^2 + x + 1) = x^6 + x^3 + x^2 + x + 1 = 01001111 = 4f$

As can be seen, this method is in keeping with the XOR addition of bytes. However, multiplication within the field is not quite as simple.

The issue with multiplying two bytes with each other within the finite field is that it is possible to receive a polynomial of higher order than 7; polynomials like this cannot be stored as a byte. Therefore, a reducing polynomial of the order 8 must be utilized. AES defines this reducing polynomial as $x^8 +$

$x^4 + x^3 + x + 1$. The Galois Field $GF(2^8)$ utilizing the reducing polynomial $x^8 + x^4 + x^3 + x + 1$ for multiplication is referred to as Rijndael's finite field[9].

In order to prevent a result that cannot be stored as a byte, all results of multiplications between bytes are then taken modulo the reducing polynomial. This means that, if the degree of a resulting polynomial exceeds 7, an XOR division using the reducing polynomial must be performed.

2.2.2.2 Finding the multiplicative inverse using the extended euclidean algorithm

The multiplicative inverse of a number is defined as equalling one when multiplied with the number in question. Normally, this is not too difficult to find: The number itself can simply be taken to the power of -1. For example, the multiplicative inverse of 5 is $\frac{1}{5}$.

In AES however, it is often necessary to find the multiplicative inverse of a byte. Furthermore, this multiplicative inverse must be found within the Rijndael field, meaning the multiplicative inverse must be found modulo the reducing polynomial. To this end, the extended euclidean algorithm is used.

The regular Euclidean algorithm is simply a computationally efficient method to find the greatest common divisor of two positive integers. In order to find the gcd, the Euclidean algorithm checks how many times the smaller integer goes into the larger one, leaving r over. The algorithm then checks how many times r goes into the smaller integer, continuing in this manner until $r = 0$. Once this point has been reached, the last value before r=0 is the gcd.

$a = k * b + r$

Next, assign b to a and r to b: $a = b, b = r$

Continue to repeat these steps until $r = 0$.

The regular euclidean algorithm can be applied to polynomials over a finite field, such as Rijndael's finite field, with no issues as long as the rules of the

field are observed[10].

The extended Euclidean algorithm is used to compute the integers x and y ($x, y \in \mathbb{Z}$) of Bézouts Identity, in order to fulfil the following equation:

$ax + by = gcd(a, b)$

The following example uses $a = 240$ and $b = 46$.

$$\begin{pmatrix} r_k & x_k & y_k & q_k \\ 240 & 1 & 0 & 1 \\ 46 & 0 & 1 & 5 \\ 10 & 1 & -5 & 4 \\ 6 & -4 & 21 & 1 \\ 4 & 5 & -26 & 1 \\ 2 & -9 & 47 & 2 \\ 0 & & & \end{pmatrix}$$

The starting conditions of the algorithm are always identical: $x_0 = 1, x_1 = 0$ and $y_0 = 0, y_1 = 1$. Next, the largest value of a and b is placed in r_0 (in this case 240), with the smaller value being placed in r_1 (in this case 46). The quotient q is how many times 46 goes into 240, namely 5 times. The remainder of 10 is then placed into the next spot in the r_k column. Now, x and y can be computed: $x_2 = x_0 - x_1 * q_1 \Rightarrow 1 = 1 - 0 * 5$

$y_2 = y_0 - y_1 * q_1 \Rightarrow -5 = 0 - 1 * 5$

This can be used to calculate all x_k and y_k values[11]:

$x_k = x_{k-2} - x_{k-1} * q_{k-1}$

$y_k = y_{k-2} - y_{k-1} * q_{k-1}$

All columns are then filled using this method until a value in the r_k column reaches 0. At this point, the values in the row above the 0 are the answers to the equation $ax + by = gcd(a, b)$.

Within Rijndaels finite field, this becomes more complicated, as the algorithm must be applied within a finite field which employs a reducing polynomial, as well as being applied to polynomials instead of integers.

Since the reducing polynomial is $x^8 + x^4 + x^3 + x + 1$, the equation that must be solved to find the multiplicative inverse of the polynomial p goes as follows:

$$a * p + b * (x^8 + x^4 + x^3 + x + 1) = 1 = gcd(p, x^8 + x^4 + x^3 + x + 1)$$

All operations within the field are performed modulo the reducing polynomial, meaning that the gcd of the reducing polynomial and any other polynomial is always equal to one. Furthermore, $b * (x^8 + x^4 + x^3 + x + 1) = 0$.

In order to find the multiplicative inverse of p, namely a, the extended euclidean algorithm can be applied. As always, it is of great importance that the rules of the field are observed.

2.2.2.3 Subkey generation (KEYEXPANSIONS)

As can be seen in the high-level explanation, AES requires 11 128-bit subkeys (also referred to as round keys); one for the initial XOR operation, and then one for each of the 10 rounds. These subkeys are derived from the main key using the "Rijndael key schedule" [12].

The first round key, utilized for the initial XOR operation, is simply the master key itself.

$$\begin{pmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{pmatrix}$$

Each subsequent round key must be generated through key expansion.

The first four bytes of the 16-byte round key, which are contained within the first column of the round key matrix, are generated first. To this end, the last, as in the rightmost, column is stored in a temporary variable, for example x. The so-called RotWord operation is then employed, wherein each byte is transposed one step upwards.

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \xrightarrow{RotWord} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_0 \end{pmatrix}$$

Afterwards, the four bytes resulting from the RotWord operation are each replaced through utilization of a Rijndael S-Box (see chapter 2.2.2.2). The resulting vector now needs to be added to the first, as in the leftmost, column of the master key using an XOR gate.

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_0 \end{pmatrix} \xrightarrow{SubBytes} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} + \begin{pmatrix} k_0 \\ k_1 \\ k_2 \\ k_3 \end{pmatrix}$$

The final step to obtain the first four bytes of the round key is referred to as the RCon, meaning round constant [13, p. 15] step. In this step, a vector dependent on the iteration, the chronological number of the round key being generated starting with one, is added using an XOR gate. These vectors are constant for every application of AES; for example, the RCon vector obtained using an iteration of eight will always be used in the process of obtaining the first four bytes of the eighth round key.

The RCon vector is calculated through the following formula, with i standing for the iteration:

$$\begin{pmatrix} 02^{i-1} \\ 00 \\ 00 \\ 00 \end{pmatrix}$$

As this operation takes place in Rijndael's finite field, the byte $02 = x$, and if the RCon value possesses an order higher than 7, it must be taken modulo the reducing polynomial (see chapter 2.2.2.1). After the RCon operation is complete, the first column of the round key has been obtained.

The generation of the other three columns is now relatively simple; only the XOR operation is necessary from this point onwards. In order to obtain the second column of the round key, the first column of the round key is added to the second column of the master key. The second column of the round key is now added to the third column of the master key in order to obtain the third column of the round key. Finally, in order to obtain the fourth column of the round key, the fourth column of the master key is added to the third column of the round key.

$$\begin{pmatrix} r_0 & r_4 = r_0 + k_4 & r_8 = r_4 + k_8 & r_{12} = r_8 + k_{12} \\ r_1 & r_5 = r_1 + k_5 & r_9 = r_5 + k_9 & r_{13} = r_9 + k_{13} \\ r_2 & r_6 = r_2 + k_6 & r_{10} = r_6 + k_{10} & r_{14} = r_{10} + k_{14} \\ r_3 & r_7 = r_3 + k_7 & r_{11} = r_7 + k_{11} & r_{15} = r_{11} + k_{15} \end{pmatrix}$$

This round key is taken as the master key for the generation of the next round key. This process is then repeated until ten new keys have been generated.

2.2.2.4 Substitution using lookup table (SUBBYTES)

The first substitution is relatively simple to perform; this is because it is a bytewise operation. This means that the data being manipulated is one single byte. In this case, the manipulation is a substitution of each byte with another byte, the substitution being determined by a lookup table, in which each individual byte is assigned another byte with which it must be swapped.

The reason that a lookup table is practical is that there are only $2^8 = 256$ different possible bytes.

This swapping of bytes is performed using a Rijndael S-box as the lookup table. Firstly, it is very important to note that this substitution is performed in a Galois field (also referred to as a finite field)[14], specifically $GF(2^8)$.

The S-Box performs a revertible, complete transformation of the plaintext[15]. This means that the transformation is reversible (which is necessary for decryption), and that no two different bytes are transformed into the same byte; this is the criteria for a full permutation.

In practice, the table is pre-calculated, since it is the same for each byte. Therefore, the most simple method of performing this substitution is to first split the byte into two 4-bit nibbles. If the byte is represented in hexadecimal, then the first nibble is simply the first of the two values, the second nibble being the second of the two values. For example, the byte $(7c)_h$, $(01111100)_2$ in binary, would be converted to the byte $(c6)_h$.

S-Box Values																
S(rs)		s														
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
r	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 2.2: The lookup table used for AES, in hexadecimal notation[15].

First, the multiplicative inverse of the byte is calculated within Rijndael's finite field. This multiplicative inverse, also a byte, is then stored as a vector $[x_7; x_6...x_0]$ to be used in an affine transformation. As always, it is important to note that the plus sign denotes an XOR operation.

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} * \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix}$$

So, after being manipulated by the S-Box, the byte of the state is replaced by the byte $[b_7; b_6...b_0]$. This step is performed for each of the 16 bytes in the state. After each byte of the state has been replaced, the new state continues on to the ShiftRows step.

2.2.2.5 Transposition (SHIFTROWS)

The next operation, transposition, operates upon the rows of the state by shifting each byte by a certain offset. The offset starts with 0, and is then increased by 1 for each row downwards from the top row.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \xrightarrow{ShiftRows} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix}$$

2.2.2.6 Substitution using formula (MIXCOLUMNS)

The MixColumns step replaces each byte in the state with a different byte, dependent on all the bytes of that same column. Once again, it is important to note that all multiplications are performed within Rijndael's finite field.

In the MixColumns step, each column (and crucially not each individual byte) is treated as a polynomial in Rijndael's finite field. However, the reducing

polynomial x^4+1 is used instead of $x^8+x^4+x^3+x+1$, as the column (which contains four values) will be represented as a third-order polynomial[13].

They are then multiplied, modulo the reducing polynomial, with a fixed polynomial $c(x) = 03 * x^3 + 01 * x^2 + 01 * x + 02$. This process can also be shown by treating the column as a vector, and multiplying it with a special matrix:

The column (treated as a vector) is multiplied with a circulant (meaning that each row of the matrix is identical to the one above it, except that the values are offset by 1 to the right) MDS ("maximum distance separable", a representation of a function known to have useful properties in cryptography) Matrix [16].

It is very important to note that the values of the cyclical MDS matrix are not regular numbers, but actually represent hexadecimal bytes (for example, $03 = 11$, which is then padded with zeroes until it possesses 8 values, making it 00000011. This is permissible in Rijndael's finite field, as $11 = x + 1 = 00000011$)

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} * \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

Now, it becomes evident that each byte of the resulting column is dependent on every byte of the manipulated column. As always, all operations take place in Rijndael's finite field.

$s_0 = 02 * a_0 + 03 * a_1 + 01 * a_2 + 01 * a_3$
$s_1 = 01 * a_0 + 02 * a_1 + 03 * a_2 + 01 * a_3$
$s_2 = 01 * a_0 + 01 * a_1 + 02 * a_2 + 03 * a_3$
$s_3 = 03 * a_0 + 01 * a_1 + 01 * a_2 + 02 * a_3$

It is important to note that the last round of AES does not possess a MixColumns step; this is because, if there were a MixColumns step included in

the last round of encryption, it would not possess a corresponding invMix-Columns in the final round of decryption[17].

2.2.2.7 Round-closing XOR operation (ADDROUNDKEY)

Finally, the state is added to the round's unique round key using modular addition. The method is the same as described in chapter 2.2.2; an XOR gate is, once again, utilized.

2.3 Decryption

Just like encryption, decryption also begins with key generation. Because the algorithm used is exactly the same as in encryption, the same round keys will once again be generated using the same master key that was already used for encryption. While each decryption round contains the inverses of the steps used in encryption, it is important to note that the order in which these steps appear in the rounds is not identical to the order in which their non-inversed counterparts appear in encryption.

2.3.1 Initial inverse XOR operation

Since all steps in decryption are the exact inverse of the encryption steps, the last round key used in encryption is used first in decryption[18]. The inverse of the XOR operation is, fortunately, also simply the XOR operation. The following example demonstrates how adding the key twice yields the same state as before having added the key.

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ + & + & + & + \\ 0 & 1 & 1 & 0 \\ = & = & = & = \\ 1 & 1 & 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ + & + & + & + \\ 0 & 1 & 1 & 0 \\ = & = & = & = \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

2.3.2 Rounds

Just like in encryption, decryption consists of on initial inverse add round key step, followed by ten rounds, with the last round missing the inverse MixColumns step.

2.3.2.1 INV. SHIFTROWS

The Inv. ShiftRows step is rather simple, as it simply performs the exact opposite manipulation of the regular ShiftRows step.

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,2} & a_{2,3} & a_{2,0} & a_{2,1} \\ a_{3,3} & a_{3,0} & a_{3,1} & a_{3,2} \end{pmatrix} \xrightarrow{Inv.ShiftRows} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

2.3.2.2 INV. SUBBYTES

Just as in the regular SubBytes step, a lookup table is usually utilized in any practical implementations. The lookup table is generated using the following Process:

First, the inverse of the affine transformation performed in encryption is calculated and stored in a byte $[x_7; x_6...x_0]$. For the byte $[b_7; b_6...b_0]$, the inverse affine transformation is performed as follows:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} * \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

Finally, the multiplicative inverse of the byte $[x_7; x_6...x_0]$ is found in Rijndael's finite field. This process is applied until all 16 bytes of the state have been replaced.

2.3.2.3 INV. ADDROUNDKEY

Since the inverse of an XOR addition is also the XOR addition (see 2.3.1), this step simply consists of adding the inverse round key to the state. For example, round key 9 would be the first round key to be added to the state, since round key 10 will already have been used in the initial inverse XOR operation[18].

2.3.2.4 INV. MIXCOLUMNS

The final step of each decryption round, except the last decryption round, is the Inv. MixColumns operation. The Inv. MicColumns step is performed by multiplying every column of the state with a specific polynomial d(x)[13]. Once again, it is important to note that each column is seen as a third-order polynomial in Rijndaels finite field, with the reducing polynomial $x^4 + 1$ (see 2.2.2.6).

$d(x)$ has to be the inverse of the fixed polynomial used in the regular MixColumns step, $c(x)$, so it can be calculated in the following way:

$c(x) * d(x) = 01$
$(03 * x^3 + 01 * x^2 + 01 * x + 02) * d(x) = 01$
$\Rightarrow d(x) = 0B * x^3 + 0D * x^2 + 09 * x + 0E$

Just like the regular MixColumns operation, the Inv. MiColumns operation can also be performed by viewing the column to be replaced as a vector, and then multiplying it with a special matrix. This matrix is also the inverse of the matrix used in the regular MixColumns step.

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} * \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} = \begin{pmatrix} 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \\ 00 & 00 & 00 & 01 \end{pmatrix}$$

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} * \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

2.4 Fulfilment of Shannon's properties

In 1945, Claude Shannon wrote a report that would become central to cryptography. This report was titled "A Mathematical Theory of Cryptography", and it identified two properties necessary to ensure the security of any cipher[19]. These were termed diffusion and confusion. These two properties, if fulfilled, serve to make methods of cryptanalysis, in particular statistical analysis, less effective.

2.4.1 Diffusion

Diffusion requires that, as a rule of thumb, changing one bit of the plaintext should alter half of the ciphertext, with the reverse also being true; this is referred to as the avalanche effect[20]. This is best achieved by spreading the values of the plaintext out over the entire ciphertext; hence the term "diffusion".

In AES, diffusion is provided by the MixColumns and the Shiftrows steps. The diffusion provided by the ShiftRows step is self-evident; as the bytes are entered into the state in a column-major order, mixing the bytes along the rows is a good source of diffusion.

MixColumns is a good source of diffuion because it creates a dependecy of four bytes on just one byte; if a single byte of the four substituted bytes are changed, every byte resulting from the substitution is also changed. This effect is further compounded because it is performed nine times, meaning that if just one bit, contained within a byte, of the initial plaintext is altered, the ciphertext will be dramatically different.

2.4.2 Confusion

Confusion requires that each bit of the ciphertext relies on multiple different parts of the key. This means that the input data must be altered in a non-

linear, difficult to reverse manner, thereby obscuring the relation between the ciphertext and the key[21].

In AES, confusion is provided by the SubBytes step. This can be seen if the journey of a single byte through AES is examined; Since each byte goes through so many substitutions, it is difficult to find the key even if one were to possess a large number of plaintext-ciphertext pairs.

2.5 Problems with AES

In 1997, NIST deemed it necessary to replace DES, the data encryption standard, with AES, the advanced encryption standard, because of DES' vulnerability to brute force attacks due to its small key size. 15 submissions were received, but Rijndael was selected due to its good performance and security[22].

Though AES was thouroughly vetted during the AES selection process and is considered secure enough to be used in encryption of top secret documents by the National Security Agency[23] of the united states, it is unfortunately not perfect.

2.5.1 Overconfidence and over-reliance

AES is widely used by large businesses and the United States government. Therefore, it would be a large threat to many nations national security if the Cipher were to be compromised.

One of the earliest scares was the XSL attack, which relied on the relative algebraic simplicity of AES[24].However, this proposed attack was later shown to be technically infeasible[25]; nevertheless, this shows that while AES is very secure, the cryptographic community must remain vigilant in its continued research into possible attacks in order to ensure the security of highly sensitive data across the world.

2.5.2 Keeping of a shared secret

One of AES' major issues is shared with OTP and all other symmetric-key encryption methods; the confidentiality of the key. The same argument that was made against OTP can be made against AES as well, namely that because the encryption method as well as the decryption method are both publicly known, the problem of keeping the message or file secret is simply transferred to keeping the key secret, while still being able to exchange it between the sender and the recipient.

However, in the case of AES, this issue is strongly mitigated: While the key used in OTP has to be at least as long as the message, AES keys are only 128, 192 or 256 bits long, with top secret documentation requiring the usage of either a 192 or 256-bit key[23].

Bibliography

[1] G.S Vernam. "Secret Signaling system". U.S. Patent 1,310,719. 1919.

[2] Gordon Welchman. *The Hut Six Story: Breaking the Enigma Codes*. 1997.

[3] mils electronic. *One Time Pad Encryption: The unbreakable encryption method*. 2008.

[4] learncryptography.com. *Frequency Analysis*. 2014. URL: https://learncryptography.com/cryptanalysis/frequency-analysis.

[5] figlesquidge. *Simply put, what does perfect secrecy mean?* 2014. URL: http://crypto.stackexchange.com/questions/3896/simply-put-what-does-perfect-secrecy-mean.

[6] Alfred Menezes et al. *Handbook of Applied Cryptography*. 1996.

[7] Claude E. Shannon. "Communication Theory of Secrecy Systems". In: *Bell System Technical Journal* (1949).

[8] Natarajan Meghanathan. *Advanced Encryption Standard (AES): Sub Stages; Finite Field Arithmetic*. 2015. URL: https://www.youtube.com/watch?v=dN2pJRLRcb0.

[9] Sam Trenholme. *AES' Galois field*. 2010. URL: http://www.samiam.org/galois.html.

[10] Mitch Keller. *Finding the Greatest Common Divisor of Polynomials Over a Finite Field*. 2013. URL: https://www.youtube.com/watch?v=q9lKz-cicWI.

[11] Ken Ward. *Computing Both the gcd and Solving ax+by=d*. acessed 2011. URL: https://trans4mind.com/personal_development/mathematics/numberTheory/euclidsAlgorithmAndExtendedAlgorith.htm.

[12] Sam Trenholme. *Rijndael's key schedule*. 2010. URL: http://www.samiam.org/key-schedule.html.

[13] Joan Daemen and Vincent Rijmen. *The Rijndael Block Cipher*. 1991. URL: http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf.

[14] Christoforus Juan Benvenuto. *Galois Field in Cryptography*. 2012. URL: https://www.math.washington.edu/~morrow/336_12/papers/juan.pdf.

[15] Neal R Wagner. *The Laws of Cryptography: Advanced Encryption Standard: S-Boxes*. 2001. URL: http://www.cs.utsa.edu/~wagner/laws/SBoxes.html.

[16] Kit Choy Xintong. *Understanding AES Mix-Columns Transformation Calculation*. 2016. URL: http://www.angelfire.com/biz7/atleast/mix_columns.pdf.

[17] Paulo Ebermann. *Why is MixColumns omitted from the last round of AES?* 2011. URL: http://crypto.stackexchange.com/questions/1346/why-is-mixcolumns-omitted-from-the-last-round-of-aes.

[18] Paulo Ebermann. *In which order are the round keys used during AES decryption?* 2012. URL: http://crypto.stackexchange.com/questions/2712/in-which-order-are-the-round-keys-used-during-aes-decryption.

[19] Claude Shannon. "A Mathematical Theory of Communication". In: *The Bell System Technical Journal* (1948).

[20] Amish Kumar and Namita Tiwari. *effective implementation and avalanche effect of AES*. 2012. URL: http://airccse.org/journal/ijsptm/papers/1213ijsptm03.pdf.

[21] Oleski. *When confusion is applied during encryption*. 2016. URL: http://crypto.stackexchange.com/questions/6699/when-confusion-is-applied-during-encryption.

[22] NIST. *Commerce Department Announces Winner of Global Information Security Competition*. 2000. URL: https://www.nist.gov/news-events/news/2000/10/commerce-department-announces-winner-global-information-security.

[23] CNSS. *CNSS Policy No. 15, Fact Sheet No. 1*. 2003.

[24] Nicolas T. Courtois and Josef Pieprzyk. *Cryptanalysis of Block Ciphers with Overdefined Systems of Equations*. 2002.

[25] T. Moh. *On the Courtois-Pieprzyk's Attack on Rijndael*. 2002. URL: https://web.archive.org/web/20100305200432/http://www.usdsi.com/aes.html.