

# Cryptography Report

Class ID: 157182

**Encryption Algorithms Reimplementation and Verification**

**Instructor: Assoc.Prof Ha Duyen Trung**

## **Group Members:**

Nguyen Dang Duong – 20224342

Do Viet Dung – 20224340

Dao Xuan Bach – 20224356

Hoang Gia Duc – 20224339

*Hanoi, 2025*

# Foreword

We would like to express our sincere gratitude to Assoc.Prof Ha Duyen Trung for his exceptional guidance and support throughout this cryptography project. His deep expertise in cryptographic algorithms and secure system design has been invaluable to the success of this work.

In the course of this project, we have studied various cryptographic principles and, building on our self-directed exploration of modern cryptographic methods, this project brings together these concepts in the implementation of several core algorithms.

Through this project, we aim to provide clear, educational implementations of modern cryptographic algorithms, including DES, 3DES, AES, ECC, RSA, SHA-512, and Whirlpool. This work not only demonstrates the practical applications of these algorithms but also offers insights that will benefit future students in mastering cryptographic and secure system design.

---

## Task Assignment Table

Members	Assignments
Nguyen Dang Duong - 20224342	<b>DES, 3DES, AES, Whirlpool</b> implementation
Do Viet Dung - 20224340	<b>SHA-512, RSA, Whirlpool</b> implementation
Hoang Gia Duc - 20224339	<b>SHA-512, RSA, Timing Attack</b> implementation
Dao Xuan Bach - 20224356	<b>ECC, Brute-Force Attack</b> implementation

Table 1: Task Assignment Table

Above is the Task Assignment Table of our group. **Algorithms listed more than once** means they are **worked on together by multiple members**. An encryption algorithm requires research, understanding the structure, code implementation, analysis, and validation, so it is reasonable for more than one member to work on it.

As for **the report**, we are **all working on it together**.

## Source code

We have coded the **implementation from scratch** and uploaded it to GitHub. You can access the full code and simulations at the following link: [https://github.com/duong1124/Crypto\\_Algo](https://github.com/duong1124/Crypto_Algo)

---

# Contents

<b>Foreword</b>	<b>1</b>
<b>1 Algorithms Structure</b>	<b>4</b>
1.1 Data Encryption Standard (DES)	4
1.1.1 DES Structure	4
1.1.2 Multiple DES	8
1.1.3 Analysis	9
1.2 Advanced Encryption Standard (AES)	9
1.3 Elliptic Curve Cryptography (ECC)	13
1.4 RSA Algorithm	16
1.5 Secure Hash Algorithm (SHA-512)	18
1.6 Whirlpool Hash function	23
1.6.1 Whirlpool Cipher	23
1.6.2 Whirlpool Hash	24
<b>2 Attack Algorithms</b>	<b>26</b>
2.1 Brute-Force Attack	26
2.1.1 Brute Force Attack Approach	26
2.1.2 Results	27
2.1.3 Conclusion	27
2.2 Timing Attack	27
2.2.1 Conditions of the Attack	28
2.2.2 Implementation Description	28
<b>3 Simulation</b>	<b>30</b>
3.1 Algorithms Validation	30
3.1.1 DES	30
3.1.2 TripleDES	31
3.1.3 AES	31
3.1.4 RSA	31
3.1.5 ECC	32
3.1.6 SHA-512	33
3.1.7 Whirlpool Hash	34
3.2 Brute-Force Attack	35
3.3 Timing Attack	36
<b>Conclusion</b>	<b>37</b>

# Cryptography Algorithms Structure

The algorithms listed below are those we have learned in school, so we will focus primarily on the architectures that serve the purpose of reimplementation. This approach will involve analyzing the underlying structure, components, and functionality of each algorithm.

Reimplementation, in this context, refers to the process of understanding and then building or recreating these algorithms from both a theoretical and practical perspective, with a particular emphasis on their computational aspects.

The algorithms presented below were originally discussed in the book by

Behrouz A. Forouzan, *Introduction to Cryptography and Network Security*, McGraw-Hill, 2011. We are simply reiterating and re-implementing these algorithms based on the concepts introduced in that work.

## 1.1 Data Encryption Standard (DES)

DES is a symmetric-key block cipher. At the encryption site, DES takes a 64-bit plaintext and creates a 64-bit ciphertext; at the decryption site, DES takes a 64-bit ciphertext and creates a 64-bit block of plaintext. The same 56-bit cipher key is used for both encryption and decryption.

### 1.1.1 DES Structure

The encryption process is made of two permutations (P-boxes), which we call initial and final permutations, and sixteen Feistel rounds. Each round uses a different 48-bit round key generated.

The initial and final permutations are straight P-boxes that are inverses of each other. They have no cryptographic significance in DES. Each of these permutations takes a 64-bit input and permutes them according to a predefined rule.

The round takes  $L_{i-1}$  and  $R_{i-1}$  from the previous round (or the initial permutation box) and creates  $L_i$  and  $R_i$ , which go to the next round (or final permutation box). Each round has two cipher elements (mixer and swapper). Each of these elements is invertible. The swapper swaps the left half of the text with the right half. The mixer is invertible because of the XOR operation. All non-invertible elements are collected inside the function  $f(R_{i-1}, K_i)$  – which is the DES function.

The DES function applies a 48-bit key to the rightmost 32 bits  $R_{i-1}$  to produce a 32-bit output. This function is made up of four sections: an expansion P-box, a whitener (that adds key), a group of S-boxes, and a straight P-box.

To expand  $R_{i-1}$  from 32 bits to 48 bits, the 32-bit input is divided into 8 sections, each containing 4 bits. Each 4-bit section is expanded to 6 bits following a specific rule: the first four input bits are copied to the second to fifth output bits, while the first output bit comes from the last bit of the previous section, and the sixth output bit comes from the first bit of the next section. This process ensures that some input bits map to multiple output bits, such as input bit 5 mapping to output bits 6 and 8.

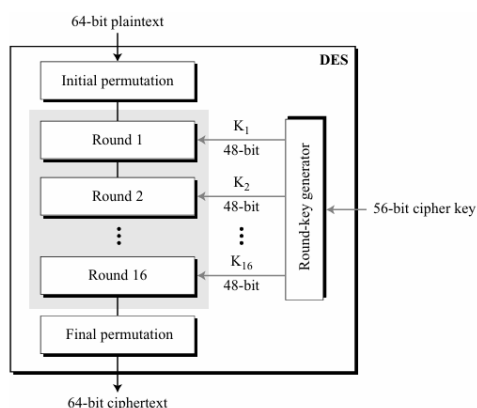


Figure 1.1: DES general structure

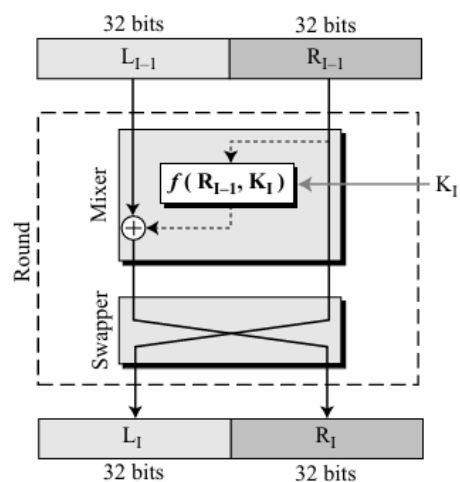


Figure 1.2: A round in DES

Initial Permutation	Final Permutation
58 50 42 34 26 18 10 02	40 08 48 16 56 24 64 32
60 52 44 36 28 20 12 04	39 07 47 15 55 23 63 31
62 54 46 38 30 22 14 06	38 06 46 14 54 22 62 30
64 56 48 40 32 24 16 08	37 05 45 13 53 21 61 29
57 49 41 33 25 17 09 01	36 04 44 12 52 20 60 28
59 51 43 35 27 19 11 03	35 03 43 11 51 19 59 27
61 53 45 37 29 21 13 05	34 02 42 10 50 18 58 26
63 55 47 39 31 23 15 07	33 01 41 09 49 17 57 25

Figure 1.3: DES Initial and final permutation tables

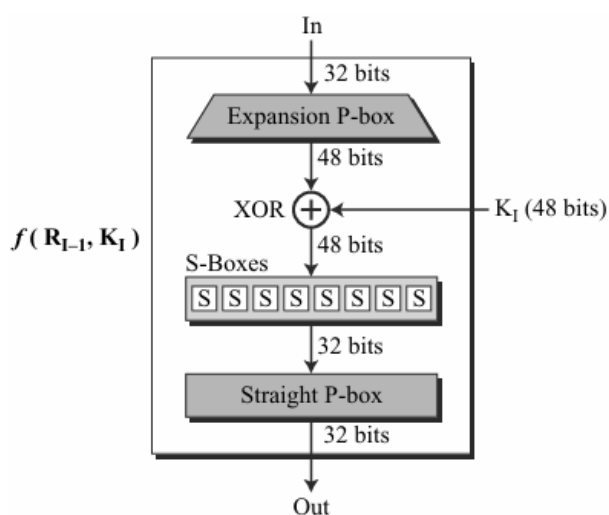


Figure 1.4: DES function

After the expansion permutation, DES uses the XOR operation on the expanded right section and the round key.

32	01	02	03	04	05
04	05	06	07	08	09
08	09	10	11	12	13
12	13	14	15	16	17
16	17	18	19	20	21
20	21	22	23	24	25
24	25	26	27	28	29
28	29	30	31	32	01

Figure 1.5: Expansion P-box table

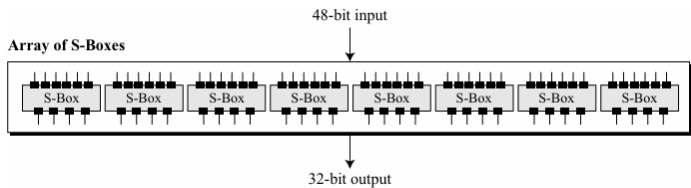


Figure 1.6: S-boxes

The 48-bit data from the second operation is divided into eight 6-bit chunks, and each chunk is fed into a substitution box (S-box). The result of each box is a 4-bit chunk; when these are combined, the result is a 32-bit text. The substitution in each S-box follows a pre-determined rule based on a 4-row by 16-column table. The combination of bits 1 and 6 of the input defines one of four rows; the combination of bits 2 through 5 defines one of the sixteen columns. This process is fundamental to the DES function, ensuring non-linear substitution and increasing encryption complexity.

Reverse Process of DES Decryption

The DES (Data Encryption Standard) algorithm is a symmetric key block cipher that operates on 64-bit blocks of data using a 56-bit key. The encryption process consists of 16 rounds of transformation, with a key used in each round. The decryption process follows the same steps as encryption, with a key distinction: the **order of the subkeys is reversed**.

- 1. **Initial Permutation:** The decryption process begins by applying the *initial permutation* to the ciphertext, just as in encryption.
- 2. **Key Generation and Reversal:** In the encryption process, 16 subkeys are generated from the 56-bit key and are used for each round. However, for decryption, the **order of the subkeys is reversed**. This ensures that applying the encryption process with the reversed key order correctly recovers the original plaintext. In the provided implementation, this reversal is achieved by calling `keys.reverse()` after the key generation.
- 3. **Rounds of Transformation:** In both encryption and decryption, the data undergoes 16 rounds of transformation using the Feistel network structure. Each round involves mixing the left and right halves of the data with the subkey, followed by swapping the halves, except for the final round.

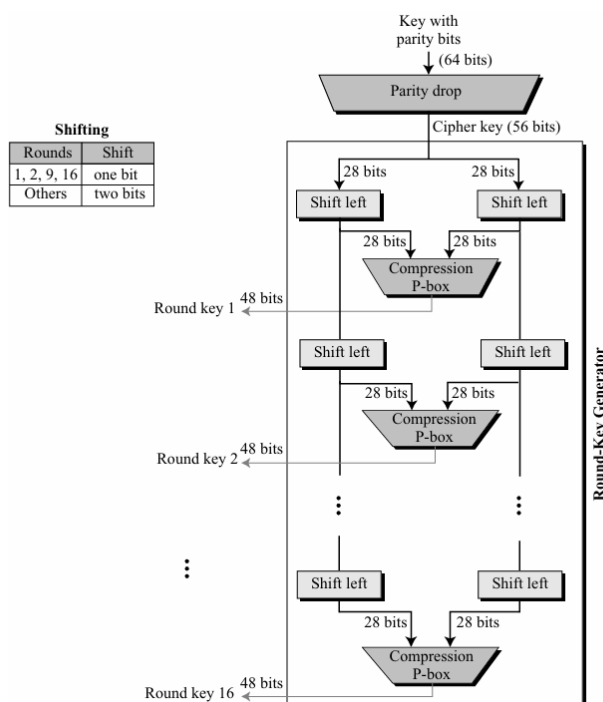


Figure 1.7: DES Key generation

4. **Final Permutation:** After all 16 rounds, the left and right halves are combined and the *final permutation* is applied to produce the decrypted plaintext.

By reversing the subkeys during decryption, DES ensures that the process of encryption and decryption are symmetric. The same algorithm and key are used for both encryption and decryption, with the only difference being the order of the subkeys.

## Key Generation

The round-key generator creates sixteen 48-bit keys from a 56-bit cipher key. However, the cipher key is typically provided as a 64-bit key, with 8 extra parity bits that are dropped before the key generation process. This preprocessing step is known as the **parity bit drop**, where bits 8, 16, 24, 32, ..., 64 are removed, leaving a 56-bit value. This 56-bit key is then used to generate the round keys.

The **parity bit drop** is a compression permutation that drops the parity bits from the 64-bit key and permutes the remaining bits according to a predefined rule. The resulting 56-bit key is used in subsequent key expansion.

After the parity drop, the key is divided into two 28-bit parts. Each part is subjected to a **circular shift left**: one bit in rounds 1, 2, 9, and 16; two bits in all other rounds. The two shifted parts are then recombined to form a 56-bit key for each round.

The **compression permutation** (a P-box) reduces the 56-bit key to 48 bits. This 48-bit key is then used in each round of the DES algorithm.



57	49	41	33	25	17	09	01
58	50	42	34	26	18	10	02
59	51	43	35	27	19	11	03
60	52	44	36	63	55	47	39
31	23	15	07	62	54	46	38
30	22	14	06	61	53	45	37
29	21	13	05	28	20	12	04

Figure 1.8: DES Parity drop table

14	17	11	24	01	05	03	28
15	06	21	10	23	19	12	04
26	08	16	07	27	20	13	02
41	52	31	37	47	55	30	40
51	45	33	48	44	49	39	56
34	53	46	42	50	36	29	32

Figure 1.9: Enter Caption

### 1.1.2 Multiple DES

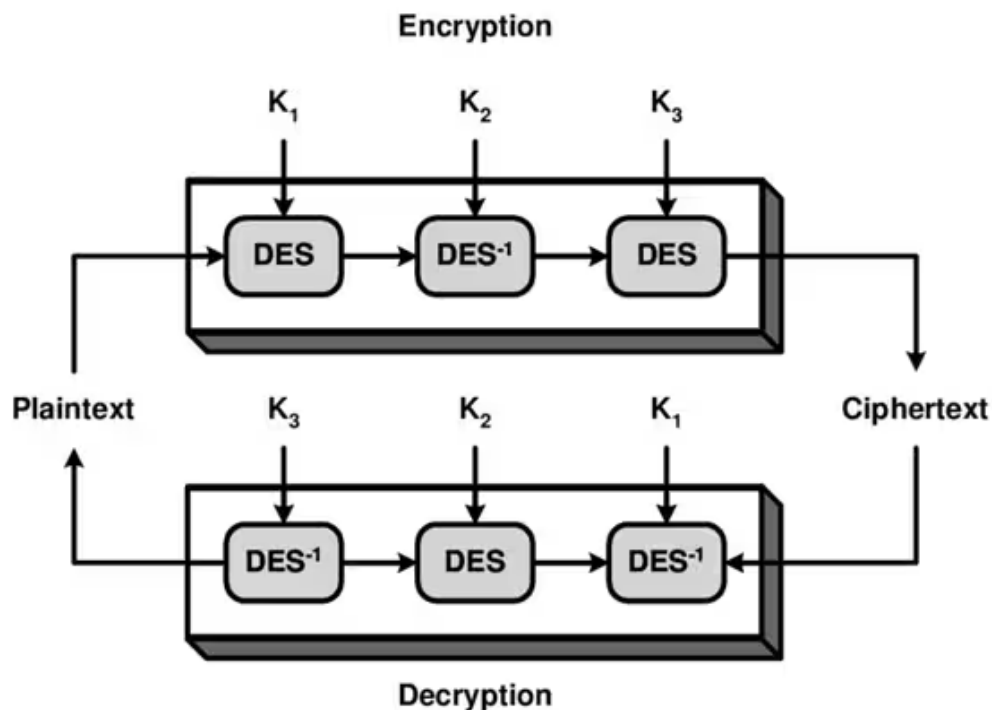


Figure 1.10: Triple DES

As we have seen, one of the major criticisms of DES regards its key length. With the available technology and the possibility of parallel processing, a brute-force attack on DES is feasible. One solution to improve the security of DES is to abandon DES and design a new cipher. We will explore this solution

in Chapter 7 with the advent of AES. However, another solution is to use multiple (cascaded) instances of DES with multiple keys. This solution, which does not require an investment in new software or hardware, is often referred to as **Triple DES (3DES)**.

Triple DES (3DES) improves the security of DES by performing the encryption and decryption process three times, using either two or three keys. The process is based on the *Encrypt-Decrypt-Encrypt (EDE)* methodology, which uses different keys in each stage.

In **Triple DES with two keys**, there are only two keys:  $k_1$  and  $k_2$ . The encryption process uses  $k_1$  in the first and third stages, while  $k_2$  is used in the second stage for decryption (reverse cipher). This method is compatible with single DES because the middle stage uses decryption in the encryption process and encryption in the decryption process.

Although Triple DES with two keys is stronger than double DES, it remains vulnerable to known-plaintext attacks. Despite this, it has been widely adopted in the banking industry due to its increased security over single DES.

To further improve security, **Triple DES with three keys** was introduced. This version uses three separate keys  $k_1$ ,  $k_2$ , and  $k_3$ , and the process is as follows: - The encryption process uses  $k_1$  and  $k_2$  in the first and second stages (EDE). - The decryption process uses  $k_2$  and  $k_1$  in the first and second stages, and  $k_3$  in the third stage (DED).

The use of three distinct keys makes 3DES with three keys significantly stronger and provides a higher level of security than using only two keys. However, the algorithm remains compatible with single DES, which is important for backward compatibility in legacy systems.

Triple DES with three keys has been used in several security applications such as PGP (Pretty Good Privacy), where it provides a robust solution for encrypting sensitive data.

### 1.1.3 Analysis

The primary weakness of DES is its relatively short 56-bit key, which makes it vulnerable to brute-force attacks. To address this, Triple DES (3DES) was introduced, using multiple applications of the DES algorithm with one or more keys. 3DES with two keys effectively has a 112-bit key length, and with three keys, it provides a 168-bit key length, significantly increasing security and resisting brute-force attacks. However, despite these improvements, 3DES is still vulnerable to known-plaintext attacks, and its computational complexity is higher than that of DES, which may impact performance in some applications.

While 3DES remains widely used, particularly in legacy systems, modern cryptographic systems have transitioned to more efficient and secure algorithms such as AES. Nevertheless, 3DES with two or three keys still provides robust security for many applications.

## 1.2 Advanced Encryption Standard (AES)

In 1997, NIST started looking for a replacement for DES, which would be called the Advanced Encryption Standard or AES. The NIST specifications required a block size of 128 bits and three different key sizes of 128, 192, and 256 bits.

### Rounds

AES is a non-Feistel cipher that encrypts and decrypts 128-bit data blocks using 10, 12, or 14 rounds, depending on the key size (128, 192, or 256 bits). The encryption process (cipher) and decryption process (inverse cipher) are similar, but round keys are applied in reverse order during decryption. The number

---

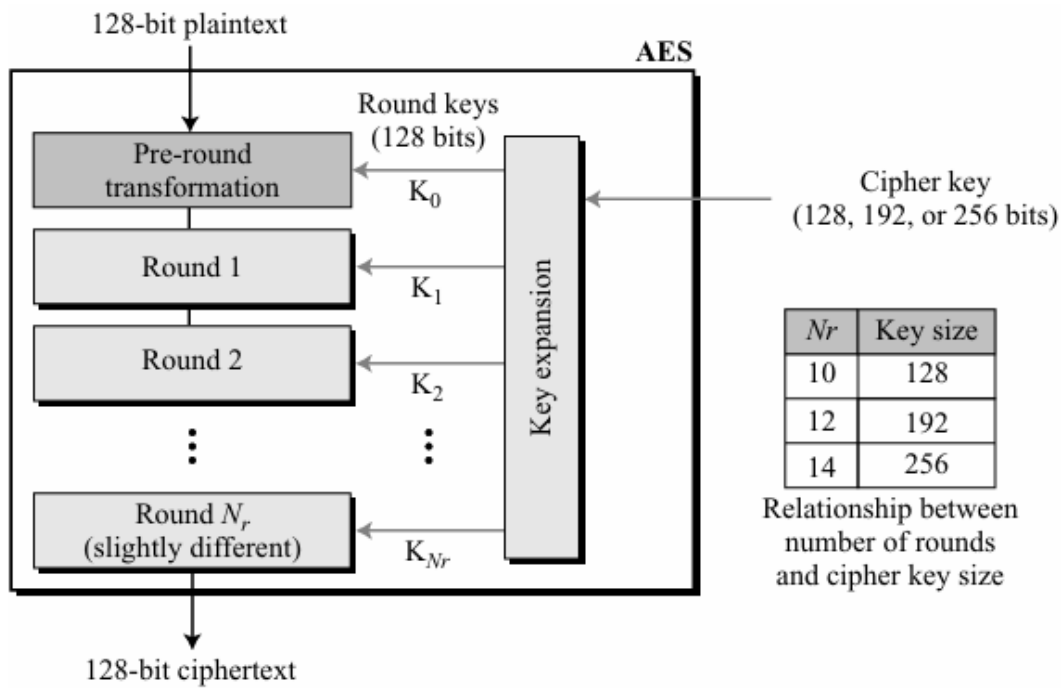


Figure 1.11: AES

of rounds ( $N_r$ ) varies with the key size, defining three versions: AES-128, AES-192, and AES-256. Round keys, generated by the key-expansion algorithm, are always 128 bits, matching the plaintext or ciphertext block size. The total number of round keys is  $N_r + 1$ , labeled as  $K_0, K_1, \dots, K_{N_r}$ .

### Data Units

AES uses five units of measurement to refer to data:

- **Bit:** A bit is a binary digit with a value of 0 or 1.
- **Byte:** A byte is a group of eight bits, which can be treated as a single entity, a row matrix ( $1 \times 8$ ) of eight bits, or a column matrix ( $8 \times 1$ ) of eight bits.
- **Word:** A word is a group of 32 bits, which can be treated as a single entity, a row matrix of four bytes, or a column matrix of four bytes.
- **Block:** A block in AES is a group of 128 bits, represented as a row matrix of 16 bytes.
- **State:** The state is a  $4 \times 4$  byte matrix (16 bytes), which evolves during encryption rounds.

### Structure of Each Round

At the decryption site, the inverse transformations are used: InvSubBytes, InvShiftRows, InvMixColumns, and AddRoundKey.

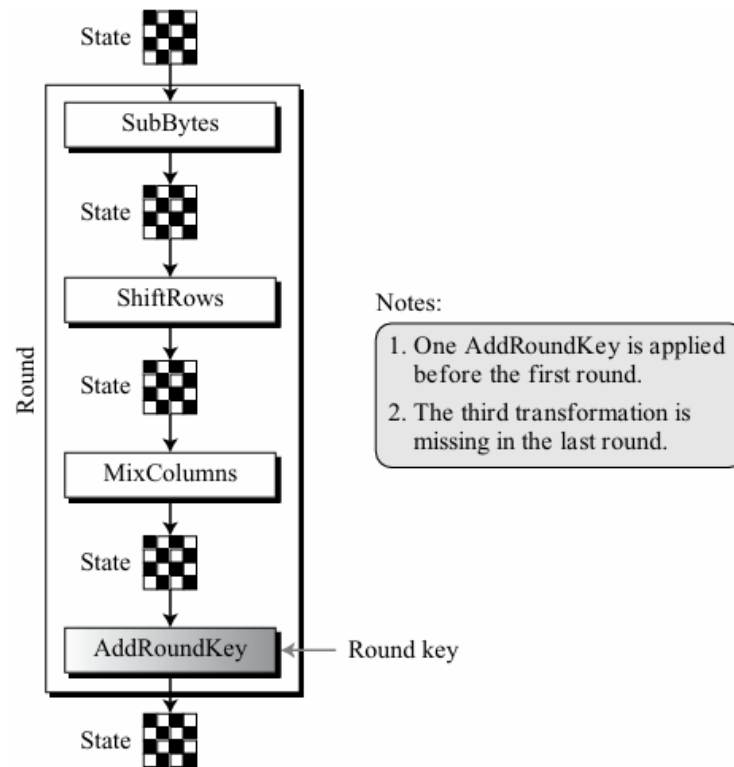


Figure 1.12: Structure of each round at the encryption site AES

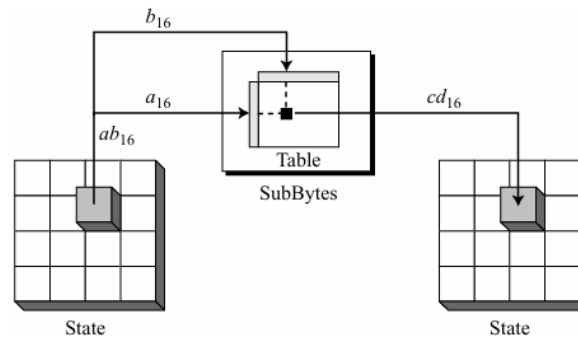


Figure 1.13: SubBytes AES

## SubBytes

The *SubBytes* transformation is the first step in AES encryption, applied to each byte of the state, which is treated as a  $4 \times 4$  byte matrix. It involves substituting each byte using a single substitution table (S-box), defined by a table lookup or mathematical calculation in the  $GF(2^8)$  field. The process interprets each byte as two hexadecimal digits, where the left digit indicates the row and the right digit indicates the column in the S-box to determine the new byte value. This transformation changes the content of each byte independently while preserving the matrix arrangement, providing a confusion effect with sixteen distinct byte-to-byte transformations.

## ShiftRows

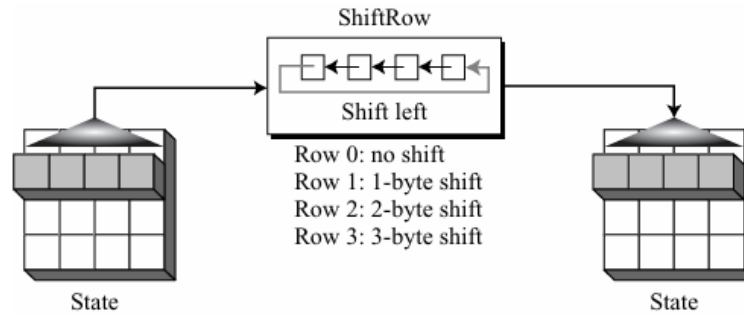


Figure 1.14: ShiftRows

The *ShiftRows* transformation in AES encryption involves left-shifting the rows of the  $4 \times 4$  state matrix, with the number of shifts depending on the row index (0 to 3). Row 0 remains unshifted, while row 1 is shifted by one byte, row 2 by two bytes, and row 3 by three bytes.

## MixColumns

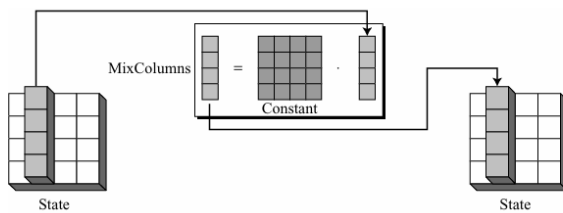


Figure 1.15: MixColumns AES

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \xleftrightarrow{\text{Inverse}} \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix}$$

$C$   $C^{-1}$

Figure 1.16: Constant matrices used by MixColumns and InvMixColumns

The *MixColumns* transformation in AES operates on each column of the  $4 \times 4$  state matrix, transforming it into a new column through matrix multiplication with a constant square matrix. Both the state column bytes and constant matrix elements are treated as 8-bit words (polynomials) with coefficients in  $\text{GF}(2)$ . Byte multiplication occurs in  $\text{GF}(2^8)$  with modulus  $0x11B$  or  $(x^8 + x^4 + x^3 + x + 1)$ , while addition is equivalent to XORing 8-bit words.

## AddRoundKey

The *AddRoundKey* transformation in AES incorporates the cipher key into the state during each round to prevent easy plaintext recovery from ciphertext. The cipher key, a secret shared between Alice and Bob, is expanded into  $N_r + 1$  round keys (each 128 bits, treated as four 32-bit words) using the key expansion process. The *AddRoundKey* transformation adds a round key word to each state column matrix, performed column by column, similar to *MixColumns*. Unlike *MixColumns*, which performs matrix multiplication, *AddRoundKey* uses matrix addition (XOR), making it its own inverse due to the reversible property of XOR.

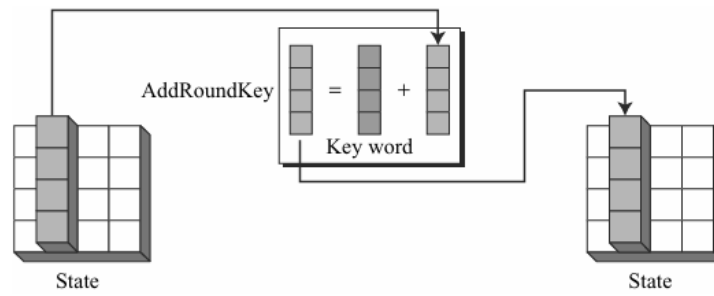


Figure 1.17: AddRoundKey AES

## Key Expansion in AES-128

The key expansion process in AES generates round keys as follows:

1. The first four words ( $w_0, w_1, w_2, w_3$ ) are derived from the cipher key, treated as a 16-byte array ( $k_0$  to  $k_{15}$ ), where each word ( $w_0$  to  $w_3$ ) is formed by concatenating four consecutive bytes ( $k_0 - k_3, k_4 - k_7$ , etc.), replicating the cipher key.
2. Subsequent words ( $w_i$  for  $i = 4$  to  $43$ ) are generated based on the following conditions:
  - If  $i \bmod 4 \neq 0$ , then  $w_i = w_{i-1} \oplus w_{i-4}$ , where the word is a combination of the previous word and the word four positions before it.
  - If  $i \bmod 4 = 0$ , then  $w_i = t \oplus w_{i-4}$ , where  $t$  is a temporary word created by applying **SubWord** and **RotWord** to  $w_{i-1}$ , and then XORing with a round constant ( $RCon$ ).

## 1.3 Elliptic Curve Cryptography (ECC)

Elliptic Curve Cryptography (ECC) is a type of asymmetric encryption that uses the mathematics of elliptic curves over finite fields to create secure public and private keys. Compared to traditional systems like RSA, which rely on very large prime numbers, ECC can offer the same level of security with much shorter keys. This makes ECC faster and more efficient, especially useful for devices with limited power and storage like smartphones and IoT devices. The idea was introduced in the 1980s by Victor Miller and Neal Koblitz, who showed that elliptic curves could be used to build secure encryption systems by replacing standard number operations with elliptic curve point operations.

### ECC over real number

General equation for an elliptic curve: Some cases to calculate in an elliptic curve, with lamda is the slope of the tangent line.

### ECC over GF(p)

Inverse point of  $P(x,y)$  is  $-P(x,-y)$  when  $-y$  is the additive inverse of  $y \bmod p$  Pseudocode for finding point on elliptic curve:

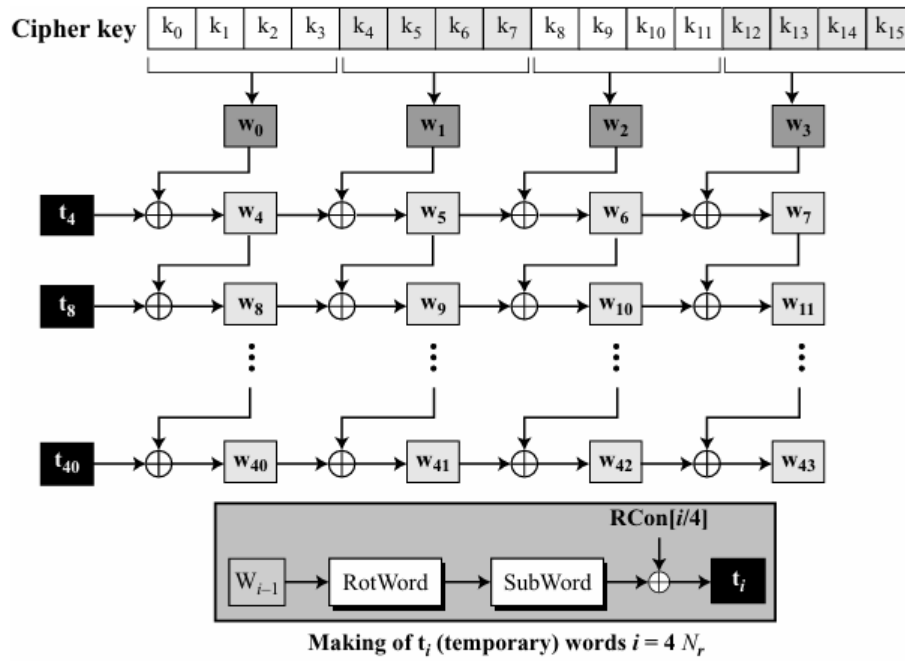


Figure 1.18: Key expansion in AES

$$y^2 + b_1xy + b_2y = x^3 + a_1x^2 + a_2x + a_3$$

Figure 1.19: General equation for real number, and for GF(p)

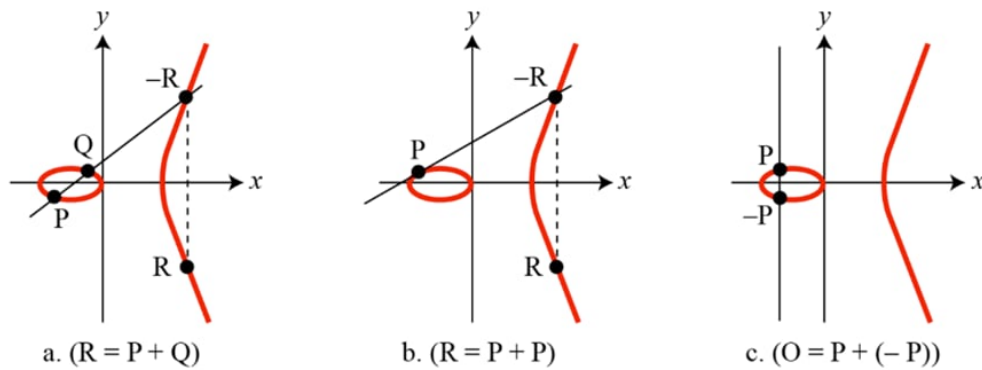


Figure 1.20: Cases calculation for point co-ordinates

### ECC over $GF(2^n)$

General equation for an elliptic curve: Inverse point of  $P(x,y)$  is  $-P(x,-y)$

Finding point on elliptic curve:

We can write an algorithm to find the points on the curve using generators for polynomials.

$$\lambda = (y_2 - y_1) / (x_2 - x_1)$$

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda (x_1 - x_3) - y_1$$

Figure 1.21: Case 1: Point addition

$$\lambda = (3x_1^2 + a) / (2y_1)$$

$$x_3 = \lambda^2 - x_1 - x_2 \qquad y_3 = \lambda (x_1 - x_3) - y_1$$

Figure 1.22: Case 2: Point doubling

**3. The intercepting point is at infinity; a point  $O$  as the point at infinity or zero point, which is the additive identity of the group.**

Figure 1.23: Case 3: Point in infinity

For  $\text{GF}(2^n)$ , we calculate (with lamda is the slope of the tangent line).

### Key generation

The security of ECC depends on the difficulty of solving the elliptic curve logarithm problem, meaning that ECC is secure because no one knows a fast way to solve the Elliptic Curve Discrete Logarithm Problem — figuring out how many times a known point was added to itself to get another point.



**Algorithm 10.12** Pseudocode for finding points on an elliptic curve

```

ellipticCurve_points ( $p, a, b$ ) //  $p$  is the modulus
{
   $x \leftarrow 0$ 
  while ( $x < p$ )
  {
     $w \leftarrow (x^3 + ax + b) \bmod p$  //  $w$  is  $y^2$ 
    if ( $w$  is a perfect square in  $\mathbb{Z}_p$ ) output  $(x, \sqrt{w}) (x, -\sqrt{w})$ 
     $x \leftarrow x + 1$ 
  }
}

```

Figure 1.24: Pseudocode for finding point on an elliptic curve

$$y^2 + xy = x^3 + ax^2 + b$$

Figure 1.25: General equation for real number, and for  $\text{GF}(2^n)$ 

$$\lambda = (y_2 + y_1) / (x_2 + x_1)$$

$$x_3 = \lambda^2 + \lambda + x_1 + x_2 + a \quad y_3 = \lambda (x_1 + x_3) + x_3 + y_1$$

Figure 1.26: Case 1: Point addition

$$\lambda = x_1 + y_1 / x_1$$

$$x_3 = \lambda^2 + \lambda + a \quad y_3 = x_1^2 + (\lambda + 1) x_3$$

Figure 1.27: Case 2: Point doubling

## 1.4 RSA Algorithm

The digital signature scheme changes the roles of the private and public keys. First, the private and public keys of the sender, not the receiver, are used. Second, the sender uses her own private key to sign the document; the receiver uses the sender's public key to verify it.

### Key generation

Key generation in the RSA digital signature scheme is identical to key generation in the RSA cryptosystem. Alice chooses two primes  $p$  and  $q$ , and calculates  $n = p \times q$ . She then calculates  $\varphi(n) = (p - 1)(q - 1)$ , where  $\varphi$  is Euler's totient function. Next, Alice selects the public exponent  $e$ , and calculates the private exponent  $d$  such that:

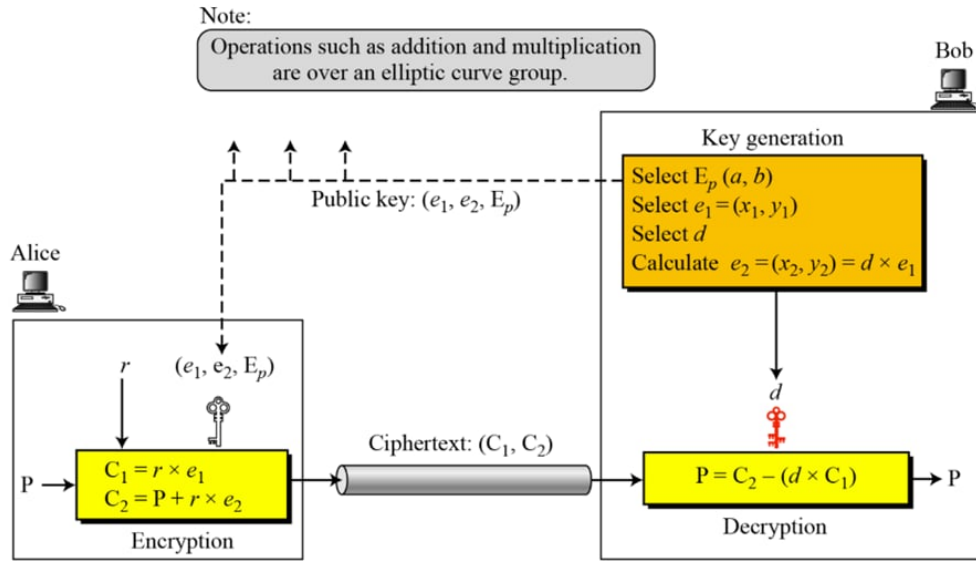


Figure 1.28: ElGamal cryptosystem using the elliptic curve

### Generating Public and Private Keys

$$E(a, b) \quad e_1(x_1, y_1) \quad d \quad e_2(x_2, y_2) = d \times e_1(x_1, y_1)$$

**Encryption**  $C_1 = r \times e_1 \quad C_2 = P + r \times e_2$

### Decryption

$P = C_2 - (d \times C_1)$  The minus sign here means adding with the inverse.

Figure 1.29: Key generations calculation

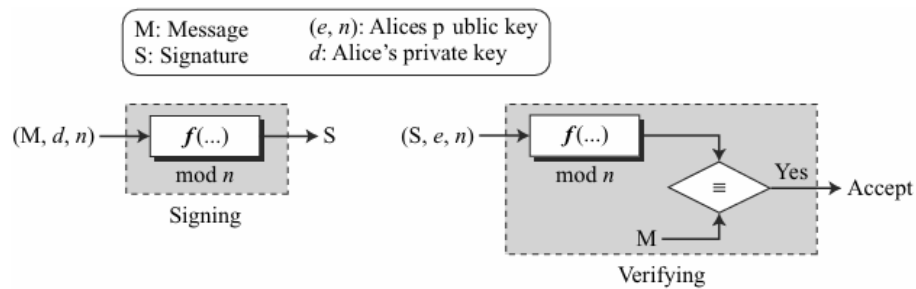


Figure 1.30: General idea behind the RSA digital signature scheme

$$e \times d \equiv 1 \pmod{\varphi(n)}$$

Alice keeps  $d$  as her private key and publicly announces  $n$  and  $e$  as her public key.

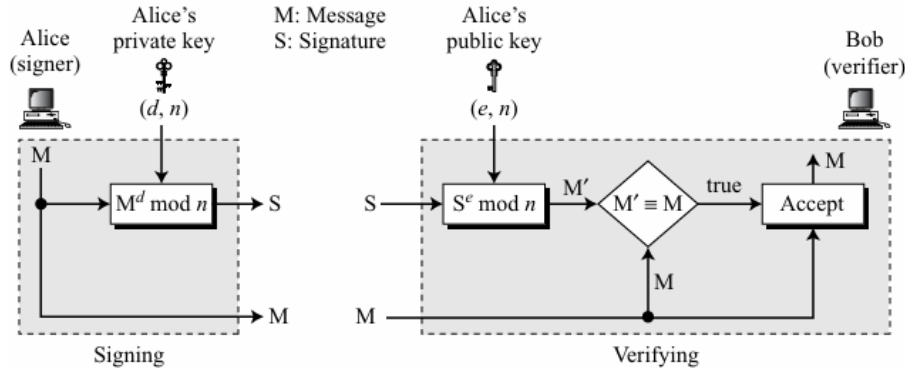


Figure 1.31: RSA digital signature scheme

**Signing:** Alice creates a signature out of the message using her private exponent,

$$S = M^d \bmod n$$

and sends the message and the signature to Bob.

**Verifying:** Bob receives  $M$  and  $S$ . Bob applies Alice's public exponent to the signature to create a copy of the message:

$$M' = S^e \bmod n$$

Bob compares the value of  $M'$  with the value of  $M$ . If the two values are congruent, Bob accepts the message. To prove this, we start with the verification criteria:

$$M' \equiv M \pmod{n} \rightarrow S^e \equiv M \pmod{n} \rightarrow M^{d \times e} \equiv M \pmod{n}$$

## 1.5 Secure Hash Algorithm (SHA-512)

SHA-512 (Secure Hash Algorithm 512-bit) is part of the SHA-2 family, designed by the National Security Agency (NSA). It takes an input message and produces a fixed 512-bit hash value, regardless of the input length. SHA-512 is widely used for data integrity and digital signatures. Internally, it follows the Merkle-Damgård construction, where the message is padded and divided into fixed-size blocks. Each block is processed in sequence by a compression function, chaining the output from one block as input to the next, until a final digest is produced.

### Length Field and Padding

Before the message digest can be created, SHA-512 requires the addition of a 128-bit unsigned-integer length field to the message that defines the length of the message in bits. The length field defines the length of the original message before adding the length field or the padding. Before the addition of the length field, we need to pad the original message to make the length a multiple of 1024. We reserve 128 bits for the length field. The length of the padding field can be calculated as follows. Let  $|M|$  be the length of the original message and  $|P|$  be the length of the padding field:

$$(|M| + |P| + 128) \equiv 0 \pmod{1024} \Rightarrow |P| = (-|M| - 128) \pmod{1024}$$

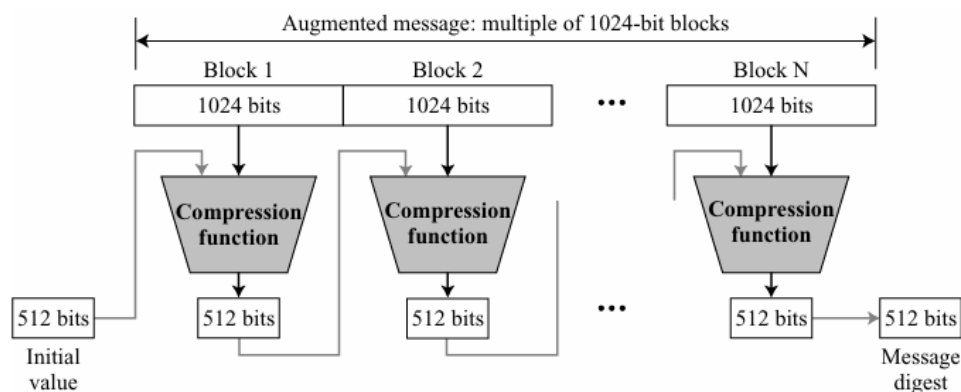


Figure 1.32: Message digest creation SHA-512

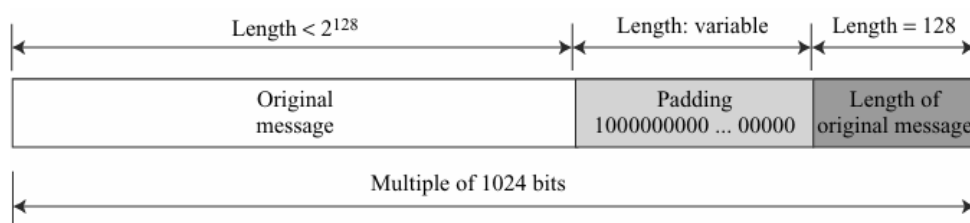


Figure 1.33: Padding and length field in SHA - 512

The format of the padding is one 1 followed by the necessary number of 0s to meet the length requirement.

## Words

SHA-512 operates on words; it is word-oriented. A word is defined as 64 bits. After the padding and the length field are added to the message, each block of the message consists of sixteen 64-bit words. The message digest is also made of 64-bit words, but the message digest is only eight words, which are named *A*, *B*, *C*, *D*, *E*, *F*, *G*, and *H*.

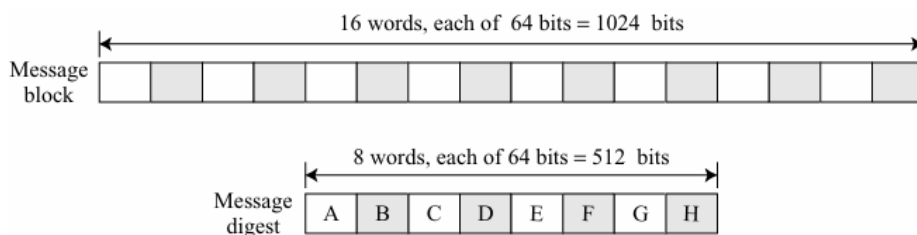


Figure 1.34: A message block and the digest as words

## Word Expansion

Before processing, each message block must be expanded. A block is made of 1024 bits, or sixteen 64-bit words. As we will see later, we need 80 words in the processing phase. Therefore, the 16-word block

needs to be expanded to 80 words, from  $W_0$  to  $W_{79}$ . The 1024-bit block becomes the first 16 words; the rest of the words come from already-made words according to the operation.

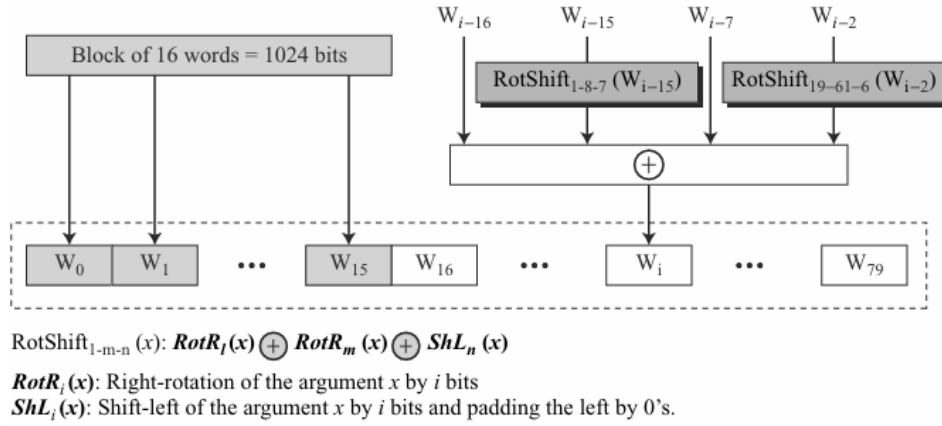


Figure 1.35: Word expansion in SHA-512

## Compression Function

SHA-512 creates a 512-bit (eight 64-bit words) message digest from a multiple-block message, where each block is 1024 bits. The processing of each block of data in SHA-512 involves 80 rounds. In each round, the contents of eight previous buffers, one word from the expanded block ( $W_i$ ), and one 64-bit constant ( $K_i$ ) are mixed together and operated on to create a new set of eight buffers. At the beginning of processing, the values of the eight buffers are saved into eight temporary variables. At the end of the process (after step 79), these values are added to the values created from step 79. We call this last operation the *final adding*.

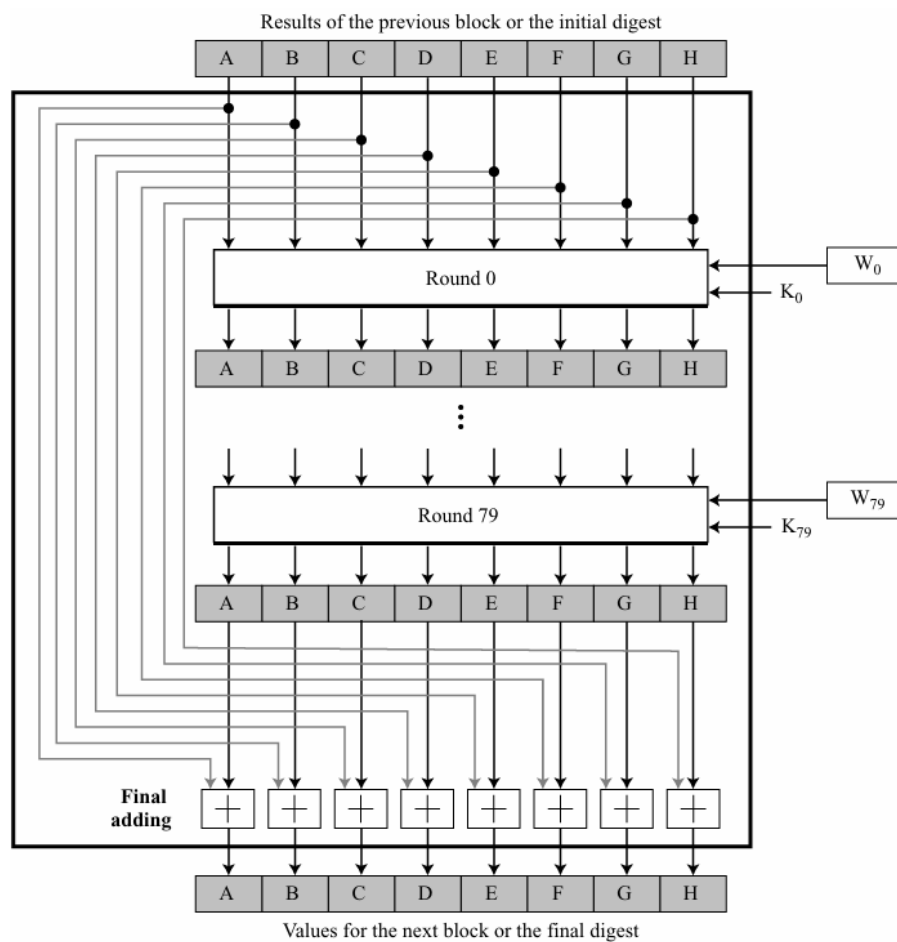


Figure 1.36: Compression function in SHA-512

### Structure of Each Round

In each round, eight new values for the 64-bit buffers are created from the values of the buffers in the previous round.

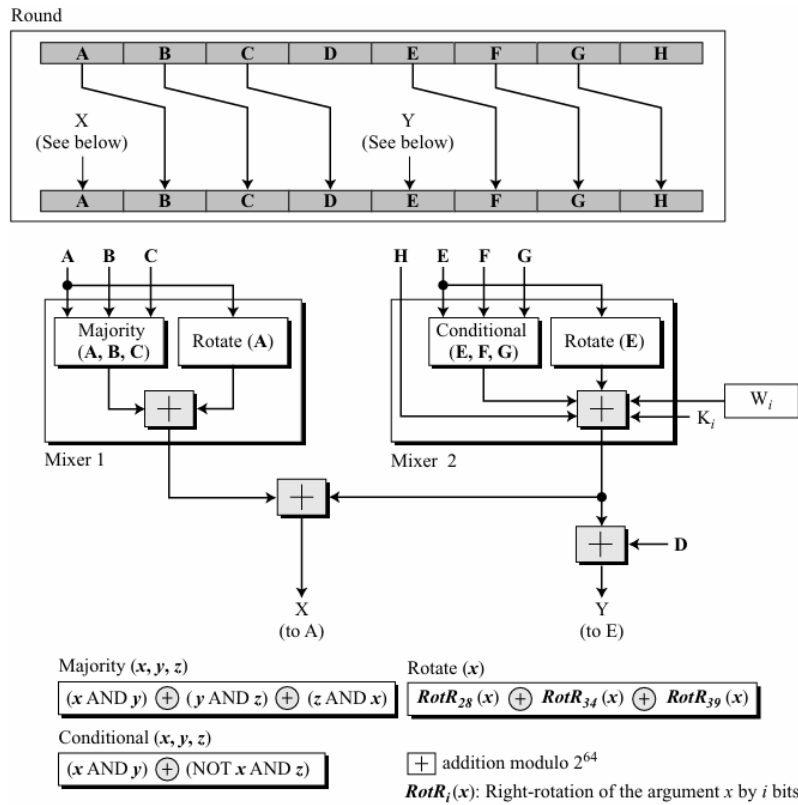


Figure 1.37: Structure of each round in SHA-512

### Constant Table and Initial Digest Initialization

SHA-512 uses a constant table for each round, and the initial digest values are stored in eight specific variables that serve as buffers during the processing of each block. These constants and initial digest values help in the iterative transformation of the message blocks into the final message digest.

428A2F98D728AE22	7137449123EF65CD	B5C0FBCFEC4D3B2F	E9B5DBA58189DBBC
3956C25BF348B538	59F111F1B605D019	923F82A4AF194F9B	AB1C5ED5DA6D8118
D807AA98A3030242	12835B0145706FBE	243185BE4EE4B28C	550C7DC3D5FFB4E2
72BE5D74F27B896F	80DEB1FE3B1696B1	9BDC06A725C71235	C19BF174CF692694
E49B69C19EF14AD2	EFBE4786384F25E3	0FC19DC68B8CD5B5	240CA1CC77AC9C65
2DE92C6F592B0275	4A7484AA6EA6E483	5CB0A9DCBD41FBD4	76F988DA831153B5
983E5152EE66DFAB	A831C66D2DB43210	B00327C898FB213F	BF597FC7BEEF0EE4
C6E00BF33DA88FC2	D5A79147930AA725	06CA6351E003826F	142929670A0E6E70
27B70A8546D22FFC	2E1B21385C26C926	4D2C6DFC5AC42AED	53380D139D95B3DF
650A73548BAF63DE	766A0ABB3C77B2A8	81C2C92E47EDAEE6	92722C851482353B
A2BFE8A14CF10364	A81A664BBC423001	C24B8B70D0F89791	C76C51A30654BE30
D192E819D6EF5218	D69906245565A910	F40E35855771202A	106AA07032BBD1B8
19A4C116B8D2D0C8	1E376C085141AB53	2748774CDF8EEB99	34B0BCB5E19B48A8
391C0CB3C5C95A63	4ED8AA4AE3418ACB	5B9CCA4F7763E373	682E6FF3D6B2B8A3
748F82EE5DEFB2FC	78A5636F43172F60	84C87814A1F0AB72	8CC702081A6439EC
90BEFFFA23631E28	A4506CEBDE82BDE9	BEF9A3F7B2C67915	C67178F2E372532B
CA273ECEEA26619C	D186B8C721C0C207	EADA7DD6CDE0EB1E	F57D4F7FEE6ED178
06F067AA72176FBA	0A637DC5A2C898A6	113F9804BEF90DAE	1B710B35131C471B
28DB77F523047D84	32CAAB7B40C72493	3C9EBE0A15C9BEBE	431D67C49C100D4C
4CC5D4BECB3E42B6	4597F299CFC657E2	5FCB6FAB3AD6FAEC	6C44198C4A475817

Figure 1.38: Eighty constants used for eighty rounds in SHA-512

## 1.6 Whirlpool Hash function

### 1.6.1 Whirlpool Cipher

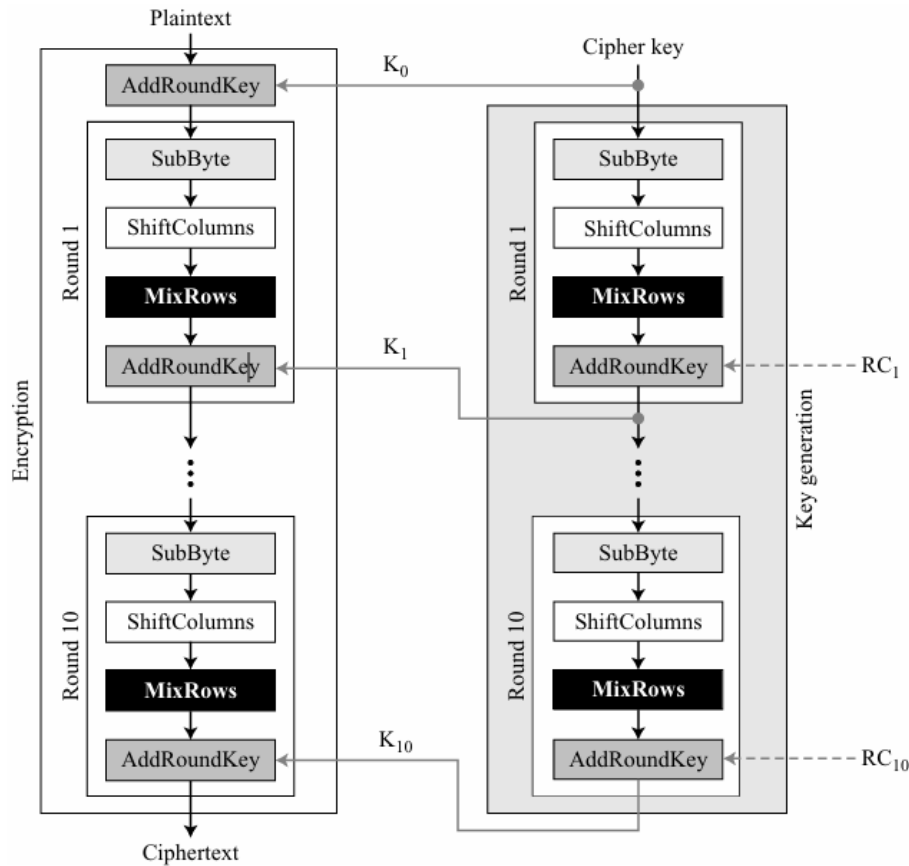


Figure 1.39: Whirlpool Cipher

Whirlpool is a non-Feistel block cipher designed primarily for use in hash algorithms. Unlike AES, which uses a Feistel structure, Whirlpool uses a straightforward round cipher with 10 rounds and operates on a 512-bit block size and 512-bit key size. It employs 11 round keys, each 512 bits long, and features a key expansion process that is distinct from AES.

#### Key Differences Between Whirlpool and AES

- **Block and State Size:** While AES uses 128-bit blocks and 128-bit state sizes, Whirlpool works with 512-bit blocks and 512-bit states, treating the block as a row matrix of 64 bytes and the state as a square 8x8 matrix of bytes. This results in a larger data size processed per round.
- **Transformation and SubBytes:** Both algorithms use a substitution step (SubBytes), but in Whirlpool, each byte is transformed based on an 8x8 matrix with 64 distinct byte-to-byte transformations. AES, in contrast, uses an S-box to perform the substitution. Whirlpool's transformation is non-linear and more complex compared to AES's SubBytes.



- **Shift Operation:** Whirlpool employs the *ShiftColumns* operation, similar to AES's *ShiftRows*, but shifts columns instead of rows. The degree of shifting depends on the column's position, adding a unique characteristic to Whirlpool's permutation process.
- **Mixing and AddRoundKey:** While AES uses MixColumns to diffuse the bits and AddRoundKey to XOR the round key, Whirlpool uses a similar process called MixRows and performs AddRoundKey byte by byte in the  $GF(2^8)$  field. Whirlpool also uses a different modulus - **0x11D** in its MixRows transformation compared to AES.
- **Key Expansion:** The key expansion in Whirlpool is unique. Instead of using a separate algorithm for round key generation, Whirlpool uses a copy of the encryption algorithm to generate round keys. The output of each round in the encryption algorithm becomes the round key for that round, and round constants (RCs) are used as virtual round keys during key expansion.

Whirlpool's design focuses on high security with a large state size and distinct round transformations. Its key expansion and non-Feistel structure set it apart from AES, making it a robust option for use in hash functions and cryptographic systems.

### 1.6.2 Whirlpool Hash

Whirlpool is an iterated cryptographic hash function based on the Miyaguchi-Preneel scheme, using a symmetric-key block cipher instead of a traditional compression function. The block cipher used in Whirlpool is a modified version of AES, tailored for hashing.

Before processing, the message must be padded. Whirlpool requires the original message length to be less than  $2^{256}$  bits. The message is padded with a single 1-bit, followed by the necessary number of 0-bits, making the total length an odd multiple of 256 bits. After padding, a 256-bit block is added to represent the length of the original message. The augmented message size is then an even multiple of 256 bits or 512 bits.

Whirlpool processes the message in 512-bit blocks. It initializes the 512-bit digest  $H_0$  to all zeros. This value is used as the cipher key for encrypting the first block. Each subsequent block is encrypted with the modified AES cipher, and the resulting ciphertext is used as the cipher key for the next block. The ciphertext from each block is XORed with the previous cipher key and the plaintext block. The final message digest is the 512-bit ciphertext after the last XOR operation.

---

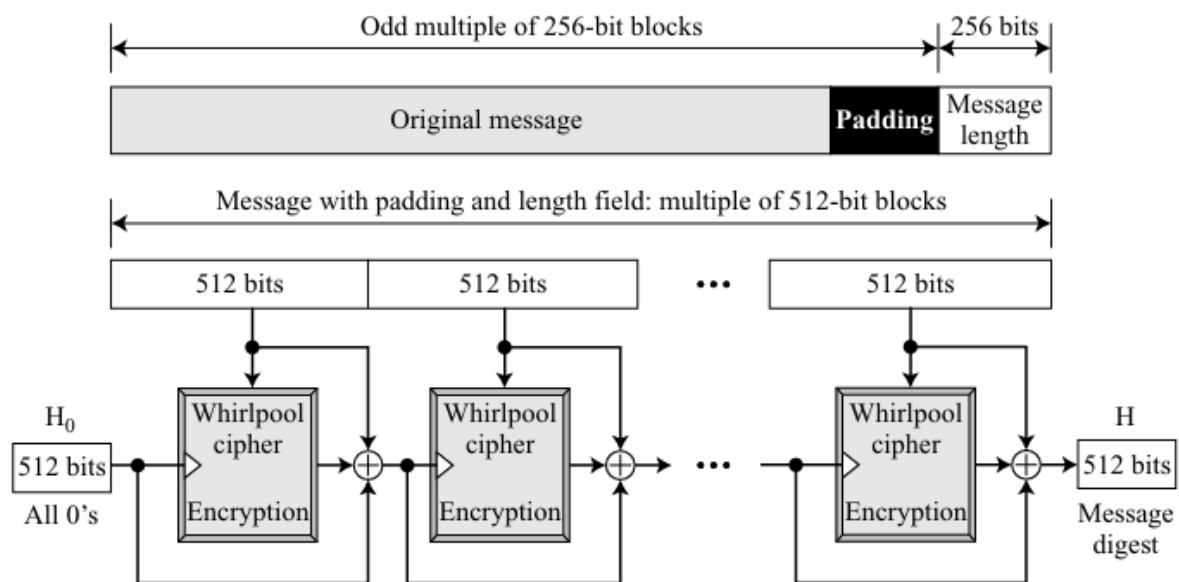


Figure 1.40: Whirlpool Hash

---

# Attack Algorithms

## 2.1 Brute-Force Attack

A brute-force attack is one of the most basic cryptographic attack methods, which involves systematically trying all possible keys or inputs until the correct one is found. The complexity of this attack depends on various factors, including the length of the key and the computational power available.

In the context of symmetric-key encryption algorithms like DES, 3DES, and AES, this approach can be used to break the encryption by testing each potential key until the decrypted ciphertext matches the known plaintext.

In this section, we will describe a brute force attack implementation that can be applied to a generic cryptographic algorithm. The attack is conducted using a class called `BruteForceAttack`, which takes an encryption algorithm as input and attempts to recover the key by testing different possible keys.

### 2.1.1 Brute Force Attack Approach

The `BruteForceAttack` class is designed to perform a brute force attack on various cryptographic algorithms (e.g., DES, 3DES, AES). The core functionality is as follows:

- **Key Generation:** The attack starts by generating a test case consisting of a known plaintext, an original key, and the corresponding ciphertext. The plaintext is encrypted using the provided cryptographic algorithm, and the ciphertext is used in the attack.
- **Brute Force Process:** The algorithm iterates over a range of possible keys and decrypts the ciphertext using each key. For each key, the decrypted output is compared with the known plaintext. If the output matches, the correct key is found.
- **Limiting the Search Space:** The number of attempts is limited to a predefined maximum value (e.g., 1000 attempts). This helps ensure that the brute force attack does not take excessive time.

The brute force attack is implemented in the following steps:

#### Generating Test Case

The `generate_test_case` method generates a simple test case consisting of:

- A 16-character plaintext: "123456ABCD132536"
- A 16-character key: "AABB09182736CCDD"
- A corresponding ciphertext generated by encrypting the plaintext with the given key.

This test case allows us to demonstrate how the brute force attack works by trying different keys and checking if the decrypted ciphertext matches the original plaintext.

### Brute Force Attack Function

The `brute_force_attack` method is the core of the brute force attack. This method performs the following steps:

- It attempts to decrypt the ciphertext using various test keys, starting from "TEST0000" up to "TEST9999".
- For each key, it decrypts the ciphertext and compares the result with the known plaintext.
- If a match is found, the correct key is returned, and the attack stops.
- If no match is found within the maximum number of attempts, the attack reports failure.

### Demonstrating the Attack

The `demonstrate_brute_force` method demonstrates how the brute force attack works in practice. It generates a test case, prints the plaintext, key, and ciphertext, and then attempts to find the key using the brute force method.

The method outputs progress updates during the attack, such as the number of attempted keys, and reports the time taken to complete the attack.

#### 2.1.2 Results

In a brute force attack, the success of the attack depends on the length of the key and the computational power available. For example, a 56-bit key, as used in DES, has 72 quadrillion possible combinations, which makes brute-forcing feasible with enough time and computing power. However, for algorithms like AES with much larger key sizes (e.g., 128 bits), the search space becomes prohibitively large, and brute forcing is computationally infeasible.

For a practical demonstration, the brute force attack was successfully able to find the correct key for the given test case within the specified number of attempts. The attack time and the number of attempts were displayed during the process. In this case, a maximum of 1000 attempts was used, which is a small number relative to the actual possible key space in real-world cryptographic systems.

#### 2.1.3 Conclusion

Brute force attacks, while simple, are computationally expensive and impractical for strong encryption algorithms with sufficiently long keys. For example, while brute force is effective against weak ciphers like DES, it is not feasible for modern algorithms like AES with long key sizes. The effectiveness of brute force attacks highlights the importance of using long keys in cryptographic systems to ensure the security of encrypted data.

`BruteForceAttack` provides a simple demonstration of how brute force works, but in real-world scenarios, more sophisticated attacks such as cryptanalysis are used to break cryptographic systems.

## 2.2 Timing Attack

A Timing Attack is a type of side-channel attack aimed at deducing an encryption key by measuring differences in the execution time of cryptographic operations. This technique is particularly dangerous

---

because it does not require direct access to the encrypted data, relying instead on processing time, often exploiting insecure key verification implementations (e.g., byte-by-byte comparison).

By measuring and analyzing the execution time of multiple checks with different trial keys, an attacker can determine each byte of the correct key based on the longest time, which corresponds to the highest number of matching bytes.

### 2.2.1 Conditions of the Attack

For a Timing Attack to succeed, several prerequisite conditions must be met:

- **Byte-by-byte key checking:** The system must compare the key byte by byte and stop immediately upon detecting a mismatch. This creates a measurable time difference between partially matching and non-matching cases.
- **Measurable processing time:** The attacker must be able to measure the execution time of the key checking process with high precision, sufficient to distinguish small differences (typically in milliseconds or microseconds).
- **Repeated access:** The attacker needs the ability to perform multiple key checks with different values to collect enough timing data, minimizing the impact of noise in the measurements.
- **Knowledge of plaintext and ciphertext:** In some cases, a Timing Attack requires the attacker to know both the plaintext and ciphertext to verify the guessed key by decryption.
- **Low noise levels:** The execution environment must have low timing noise (e.g., unaffected by background processes or system load) to ensure reliable time measurements.

### 2.2.2 Implementation Description

#### Source Code Overview

The `TimingAttack` class is developed to perform a Timing Attack, based on the assumption that the key verification process of an encryption algorithm can be exploited through differences in execution time. This class is part of a research project on decryption techniques, aimed at evaluating the security of implemented encryption algorithms. The main methods include:

- `generate_test_case`: Generates a test dataset consisting of plaintext, key, and ciphertext.
  - `_vulnerable_key_check`: Simulates an insecure key verification mechanism, performing byte-by-byte comparison and stopping upon detecting a mismatch.
  - `_timing_attack`: Executes the attack by measuring the execution time to deduce each character of the key.
  - `demonstrate_timing_attack`: Coordinates the entire process, from generating the test case to displaying the results.
-

## Implementation Process

### Generating Test Case:

The `generate_test_case` method creates a test dataset with:

- Plaintext: "123456ABCD132536" (16 hex characters, equivalent to 8 bytes)
- Key: "AABB09182736CCDD" (8 bytes, 64 bits, compatible with DES)
- Ciphertext: Generated using `algo.encrypt` from the plaintext and key, converted to binary format via `hex2bin`.

This dataset is used to simulate an attack scenario with known plaintext and ciphertext.

### Simulating Vulnerable Key Checking:

The `_vulnerable_key_check` function compares the trial key (`input_key`) with the correct key (`correct_key`) byte by byte. If a mismatch is detected, the function stops immediately and returns `False`. If all bytes match, it returns `True` after adding a small delay (`time.sleep(0.0001)`) for each matching byte, creating a measurable time difference.

### Measuring Execution Time to Guess Each Byte of the Key:

The `_timing_attack` method deduces each hex character of the key by:

- Creating trial keys by appending each character from the hex character set to the guessed key, filling the remaining positions with 0.
- Measuring the execution time of `_vulnerable_key_check` for each trial key, repeating 50 times to calculate an average, thereby reducing noise.
- Selecting the character with the longest average time as the correct one, based on the assumption that matching bytes increase the time due to the delay.

### Verifying the Guessed Key by Decryption:

After each guess, a temporary key (`test_key`) is created by concatenating the guessed key with 0 characters for the remaining positions. The `algo.decrypt` method is used to decrypt the ciphertext with `test_key`. If the result matches the plaintext (compared in binary format), the key is confirmed as correct, and the process terminates.

---

# Simulation

Below are the simulation results visualizations. You can find more details in our source code uploaded on GitHub, with the link provided at the beginning of the report.

## 3.1 Algorithms Validation

### 3.1.1 DES

The algorithm ran well with correct logic and data handling. The plaintext and decrypted ciphertext matched.

```
DES

from algorithms import DES

plaintext = "123456ABCD1234"
plaintext = hex2bin(plaintext)
print("Plain text size: [len(plaintext)] bits")

key = "A8809592736CC0"

key = hex2bin(key)
print("Key size: [len(key)] bits")

des = DES()
ciphertext = des.encrypt(plaintext, key, print_round_test=True)
print("Ciphertext: [bin2hex(ciphertext)]")
decrypted_text = des.decrypt(ciphertext, key, print_round_test=True)
print("Decrypted text: [bin2hex(decrypted_text)]")

✓ bin

Plain text size: 64 bits
Key size: 64 bits
Round 1: L: 18CA18AD, R: 5A78E394, Key: 194CD072DE8C
Round 2: L: 5A78E394, R: 4A1210F6, Key: 4568581ABCCE
Round 3: L: 4A1210F6, R: B8089591, Key: 06EDA4ACF5B5
Round 4: L: B8089591, R: 236779C2, Key: DA2D032B6EE3
Round 5: L: 236779C2, R: A15A4887, Key: 69A629FEC913
Round 6: L: A15A4887, R: 2E8F9C65, Key: C1948E87475E
Round 7: L: 2E8F9C65, R: A9FC20A3, Key: 708AD2DDB3C0
Round 8: L: A9FC20A3, R: 308BEE97, Key: 34F822F0C66D
Round 9: L: 308BEE97, R: 10AF9D37, Key: 848B4473DCCC
Round 10: L: 10AF9D37, R: 6CA6CB20, Key: 02765708B5BF
Round 11: L: 6CA6CB20, R: FF3C485F, Key: 6D5560AF7CA5
Round 12: L: FF3C485F, R: 22A5963B, Key: C2C1E96A4BF3
Round 13: L: 22A5963B, R: 387CCDAA, Key: 99C31397C91F
Round 14: L: 387CCDAA, R: BD2DD2AB, Key: 251B8BC717D0
Round 15: L: BD2DD2AB, R: CF26B472, Key: 3330C5D9A36D
Round 16: L: 19BA9212, R: CF26B472, Key: 181C5D75C66D
Ciphertext: C0B7A8D05F3A829C
Round 1: L: CF26B472, R: BD2DD2AB, Key: 181C5D75C66D
Round 2: L: BD2DD2AB, R: 387CCDAA, Key: 3330C5D9A36D
Round 3: L: 387CCDAA, R: 22A5963B, Key: 251B8BC717D0
Round 4: L: 22A5963B, R: FF3C485F, Key: 99C31397C91F
Round 5: L: FF3C485F, R: 6CA6CB20, Key: C2C1E96A4BF3
Round 6: L: 6CA6CB20, R: 10AF9D37, Key: 6D5560AF7CA5
Round 7: L: 10AF9D37, R: 308BEE97, Key: 02765708B5BF
Round 8: L: 308BEE97, R: A9FC20A3, Key: 848B4473DCCC
Round 9: L: A9FC20A3, R: 2E8F9C65, Key: 34F822F0C66D
Round 10: L: 2E8F9C65, R: A15A4887, Key: 708AD2DDB3C0
Round 11: L: A15A4887, R: 236779C2, Key: C1948E87475E
Round 12: L: 236779C2, R: B8089591, Key: 69A629FEC913
Round 13: L: B8089591, R: 4A1210F6, Key: DA2D032B6EE3
Round 14: L: 4A1210F6, R: 5A78E394, Key: 06EDA4ACF5B5
Round 15: L: 5A78E394, R: 18CA18AD, Key: 4568581ABCCE
Round 16: L: 14A7D678, R: 18CA18AD, Key: 194CD072DE8C
Decrypted text: 123456ABCD1234
```

Figure 3.1: DES Validation

```
1 Plain text size: 64 bits
2 Key size: 64 bits
3 Round 1: L: 18CA18AD, R: 5A78E394, Key: 194CD072DE8C
4 Round 2: L: 5A78E394, R: 4A1210F6, Key: 4568581ABCCE
5 Round 3: L: 4A1210F6, R: B8089591, Key: 06EDA4ACF5B5
6 Round 4: L: B8089591, R: 236779C2, Key: DA2D032B6EE3
7 Round 5: L: 236779C2, R: A15A4887, Key: 69A629FEC913
8 Round 6: L: A15A4887, R: 2E8F9C65, Key: C1948E87475E
9 Round 7: L: 2E8F9C65, R: A9FC20A3, Key: 708AD2DDB3C0
10 Round 8: L: A9FC20A3, R: 308BEE97, Key: 34F822F0C66D
11 Round 9: L: 308BEE97, R: 10AF9D37, Key: 848B4473DCCC
12 Round 10: L: 10AF9D37, R: 6CA6CB20, Key: 02765708B5BF
13 Round 11: L: 6CA6CB20, R: FF3C485F, Key: 6D5560AF7CA5
14 Round 12: L: FF3C485F, R: 22A5963B, Key: C2C1E96A4BF3
15 Round 13: L: 22A5963B, R: 387CCDAA, Key: 99C31397C91F
16 Round 14: L: 387CCDAA, R: BD2DD2AB, Key: 251B8BC717D0
17 Round 15: L: BD2DD2AB, R: CF26B472, Key: 3330C5D9A36D
18 Round 16: L: 19BA9212, R: CF26B472, Key: 181C5D75C66D
19 Ciphertext: C0B7A8D05F3A829C
20 Round 1: L: CF26B472, R: BD2DD2AB, Key: 181C5D75C66D
21 Round 2: L: BD2DD2AB, R: 387CCDAA, Key: 3330C5D9A36D
22 Round 3: L: 387CCDAA, R: 22A5963B, Key: 251B8BC717D0
23 Round 4: L: 22A5963B, R: FF3C485F, Key: 99C31397C91F
24 Round 5: L: FF3C485F, R: 6CA6CB20, Key: C2C1E96A4BF3
25 Round 6: L: 6CA6CB20, R: 10AF9D37, Key: 6D5560AF7CA5
26 Round 7: L: 10AF9D37, R: 308BEE97, Key: 02765708B5BF
27 Round 8: L: 308BEE97, R: A9FC20A3, Key: 848B4473DCCC
28 Round 9: L: A9FC20A3, R: 2E8F9C65, Key: 34F822F0C66D
29 Round 10: L: 2E8F9C65, R: A15A4887, Key: 708AD2DDB3C0
30 Round 11: L: A15A4887, R: 236779C2, Key: C1948E87475E
31 Round 12: L: 236779C2, R: B8089591, Key: 69A629FEC913
32 Round 13: L: B8089591, R: 4A1210F6, Key: DA2D032B6EE3
33 Round 14: L: 4A1210F6, R: 5A78E394, Key: 06EDA4ACF5B5
34 Round 15: L: 5A78E394, R: 18CA18AD, Key: 4568581ABCCE
35 Round 16: L: 14A7D678, R: 18CA18AD, Key: 194CD072DE8C
36 Decrypted text: 123456ABCD1234
37
```

Figure 3.2: We can see the text of each round in DES through 16 rounds in Encryption and Decryption

### 3.1.2 TripleDES

The algorithm ran well with correct logic and data handling. The plaintext and decrypted ciphertext matched.

```

3DES

from algorithms import TripleDES

tDES = TripleDES()

plaintext = "123456ABCD132536"
plaintext = hex2bin(plaintext)
print(f"Plain text size: {len(plaintext)} bits")

k1 = 'AABB00182736CCDD'
# Randomly mutate the key from k1 using a seed value
k2 = tDES.mutate_key(k1, seed = 1)
k3 = tDES.mutate_key(k1, seed = 2)

k1b = hex2bin(k1)
k2b = hex2bin(k2)
k3b = hex2bin(k3)

cipher = tDES.encrypt(plaintext, k1b, k2b, k3b)
print("Encrypted:", bin2hex(cipher))

decrypted = tDES.decrypt(cipher, k1b, k2b, k3b)
print("Decrypted:", bin2hex(decrypted))

101 ✓ 0.0s
... Plain text size: 64 bits
Encrypted: 680E152E8B6BC5AF
Decrypted: 123456ABCD132536

```

Figure 3.3: Triple DES

### 3.1.3 AES

The algorithm ran well with correct logic and data handling. The plaintext and decrypted ciphertext matched.

```

AES

from algorithms import AES

input_bytes = bytearray(b"exampleplaintext") # 16 bytes
key = bytearray(b"thisisakey123456") # 16 bytes
key_size = len(key) # Key size in bytes

aes = AES(key_size)

encrypted_data = aes.encrypt(input_bytes, key)
print("Encrypted:", encrypted_data)

decrypted_data = aes.decrypt(encrypted_data, key)
print("Decrypted:", decrypted_data)

111 ✓ 0.0s
... Encrypted: bytearray(b'\xc6\xc8/z\xa7\x90\x8a\x84\t!\xe2\xb9/\xb1b\xce')
Decrypted: bytearray(b'exampleplaintext')

```

Figure 3.4: AES

### 3.1.4 RSA

To handle large data, which in this case involves large integers, we added an option `one_at_a_time` to the RSA algorithm. This option allows for encrypting one character at a time rather than concatenating the entire message into a single large integer. Below are the results for both methods:

#### 1. One at a Time:



- **Ciphertext:** The encrypted values for each character are represented as a list: [16807, 1024, 161051, 161051, 537824].
- **Decrypted:** The decrypted values for each character are: [7, 4, 11, 11, 14]. These values correspond to the original characters in the plaintext message "hello".

## 2. Concatenate:

- **Ciphertext:** When concatenating the message into a single large integer, the ciphertext generated is: 2631140162161542955.
- **Decrypted:** Decrypting the concatenated ciphertext results in the original message: "hello".

```

RSA

from algorithms import RSA

plaintext = "hello"
p, q = generate_prime_pair(bit_size=32)
#p, q = 7, 13 # for testing
print(f"Generated primes: p = {p}, q = {q}")

rsa = RSA()
public_key, private_key = rsa.generate_keys(p, q)

print("-----ONE AT A TIME-----")
test_ciphertext = rsa.encrypt(plaintext, public_key, text=False, one_at_a_time=True)
print(f"Ciphertext: {test_ciphertext}")

test_decrypted = rsa.decrypt(test_ciphertext, private_key, text=False)
print(f"Decrypted: {test_decrypted}")

print("-----CONCATENATE-----")
true_ciphertext = rsa.encrypt(plaintext, public_key, text=False, one_at_a_time=False)
print(f"Ciphertext: {true_ciphertext}")

true_decrypted = rsa.decrypt(true_ciphertext, private_key, text=False)
print(f"Decrypted: {true_decrypted}")
✓ 0.0s

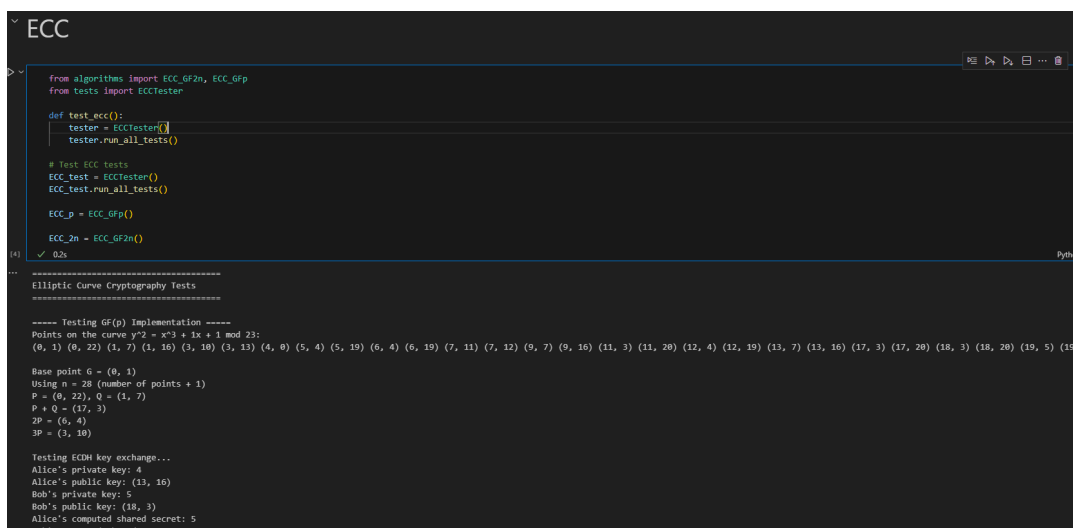
Generated primes: p = 4233579569, q = 2832818257
-----ONE AT A TIME-----
Ciphertext: [16807, 1024, 161051, 161051, 537824]
Decrypted: [7, 4, 11, 11, 14]
-----CONCATENATE-----
Ciphertext: 2631140162161542955
Decrypted: hello

```

Figure 3.5: RSA

### 3.1.5 ECC

The algorithm ran well with correct logic and data handling.



```

ECC

from algorithms import ECC_GF2n, ECC_Gfp
from tests import ECCTester

def test_ecc():
    tester = ECCTester()
    tester.run_all_tests()

# Test ECC tests
ECC_test = ECCTester()
ECC_test.run_all_tests()

ECC_p = ECC_Gfp()
ECC_2n = ECC_GF2n()

✓ 0.2s

=====
Elliptic Curve Cryptography Tests
=====

----- Testing GF(p) Implementation -----
Points on the curve y^2 = x^3 + 1x + 1 mod 23:
(0, 1) (0, 22) (1, 7) (1, 16) (3, 10) (3, 13) (4, 0) (5, 4) (5, 19) (6, 4) (6, 19) (7, 11) (7, 12) (9, 7) (9, 16) (11, 3) (11, 20) (12, 4) (12, 19) (13, 7) (13, 16) (17, 3) (17, 20) (18, 3) (18, 20) (19, 5) (19, 18)

Base point G = (0, 1)
Using n = 28 (number of points + 1)
P = (0, 22), Q = (1, 7)
P + Q = (17, 3)
2P = (6, 4)
3P = (3, 10)

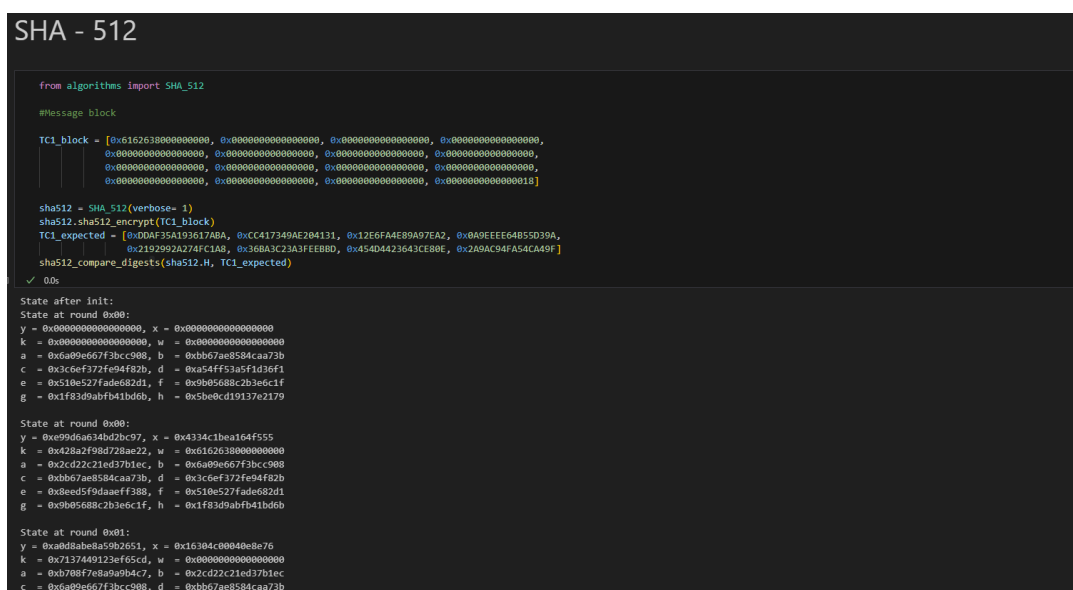
Testing ECDH key exchange...
Alice's private key: 4
Alice's public key: (13, 16)
Bob's private key: 5
Bob's public key: (18, 3)
Alice's computed shared secret: 5
Bob's computed shared secret: 5

```

Figure 3.6: ECC

### 3.1.6 SHA-512

The algorithm ran well with correct logic and data handling. We also print out 80 rounds of SHA-512.



```

SHA - 512

from algorithms import SHA_512

#Message block

TC1_block = [0x6162638000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
             0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
             0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000000,
             0x0000000000000000, 0x0000000000000000, 0x0000000000000000, 0x0000000000000018]

sha512 = SHA_512(verbose= 1)
sha512.sha512_encrypt(TC1_block)
TC1_expected = [0xDDAF35A193617ABA, 0xCC417349AE204131, 0x12E6FA4E89A97EA2, 0x0A9EEEE64855D3DA,
               0x2192992A274FC1A8, 0x36BA3C23A3FEEBBD, 0x454D4423643CE80E, 0x2A9AC94FA54CA49F]
sha512.compare_digests(sha512.H, TC1_expected)

✓ 0.0s

State after init:
State at round 0x00:
y = 0x0000000000000000, x = 0x0000000000000000
k = 0x0000000000000000, w = 0x0000000000000000
a = 0x6a09e667f3bcc908, b = 0xbb67ae8584caa73b
c = 0x3c6ef372fe94f82b, d = 0xa54ff53a5f1d36f1
e = 0x510e527fade682d1, f = 0x9b05688c2b3e6c1f
g = 0xf183d9abfb41bd6b, h = 0x5be0cd19137e2179

State at round 0x01:
y = 0xe99d8a634db2bc07, x = 0x4334c1bea164f555
k = 0x428a2f98d728ae22, w = 0x6162638000000000
a = 0x2cd22c21ed37b1ec, b = 0x6a09e667f3bcc908
c = 0xbb67ae8584caa73b, d = 0x3c6ef372fe94f82b
e = 0x8eed5f9daeff388f, f = 0x510e527fade682d1
g = 0x9b05688c2b3e6c1f, h = 0xf183d9abfb41bd6b

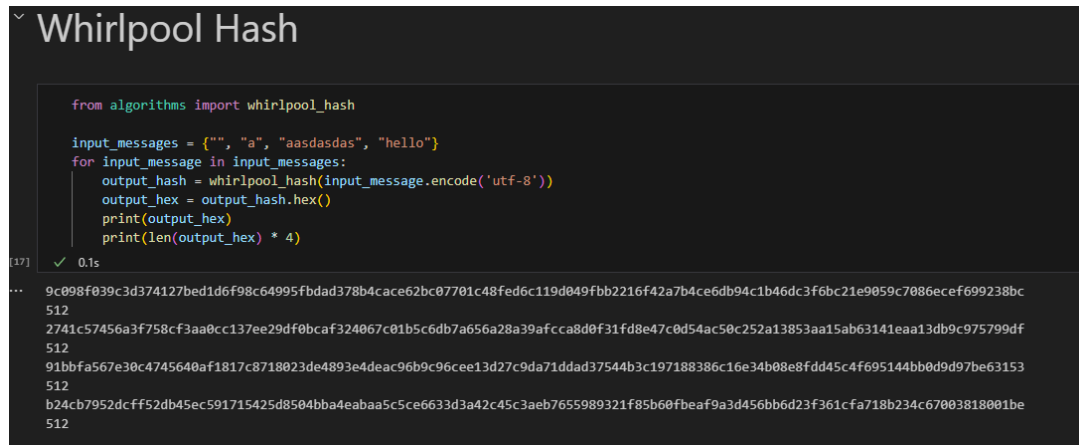
State at round 0x02:
y = 0xa0d8a8e8a59b2651, x = 0x16304c00040e8e76
k = 0x7137449123ef65cd, w = 0x0000000000000000
a = 0xb708f7e8a9a9b4c7, b = 0x2cd22c21ed37b1ec
c = 0x6a09e667f3bcc908, d = 0xbb67ae8584caa73b

```

Figure 3.7: SHA-512

### 3.1.7 Whirlpool Hash

The Whirlpool Hash algorithm ran successfully with correct logic and data handling. Regardless of the input message, the algorithm consistently produces a 512-bit output (message digest). This behavior is independent of the size or content of the input message.



```
from algorithms import whirlpool_hash

input_messages = ("", "a", "aasdadas", "hello")
for input_message in input_messages:
    output_hash = whirlpool_hash(input_message.encode('utf-8'))
    output_hex = output_hash.hex()
    print(output_hex)
    print(len(output_hex) * 4)
```

[17] ✓ 0.1s

9c098f039c3d374127bed1d6f98c64995fbdad378b4cace62bc07701c48fed6c119d049fbb2216f42a7b4ce6db94c1b46dc3f6bc21e9059c7086ecef699238bc  
512  
2741c57456a3f758cf3aa0cc137ee29df0bcdf324067c01b5c6db7a656a28a39afcca8d0f31fd8e47c0d54ac50c252a13853aa15ab63141eaa13db9c975799df  
512  
91bbfa567e30c4745640af1817c8718023de4893e4deac96b9c96cee13d27c9da71ddad37544b3c197188386c16e34b08e8fdd45c4f695144bb0d9d97be63153  
512  
b24cb7952dcff52db45ec591715425d8504bba4eabaa5c5ce6633d3a42c45c3aeb7655989321f85b60fbaef9a3d456bb6d23f361cfa718b234c67003818001be  
512

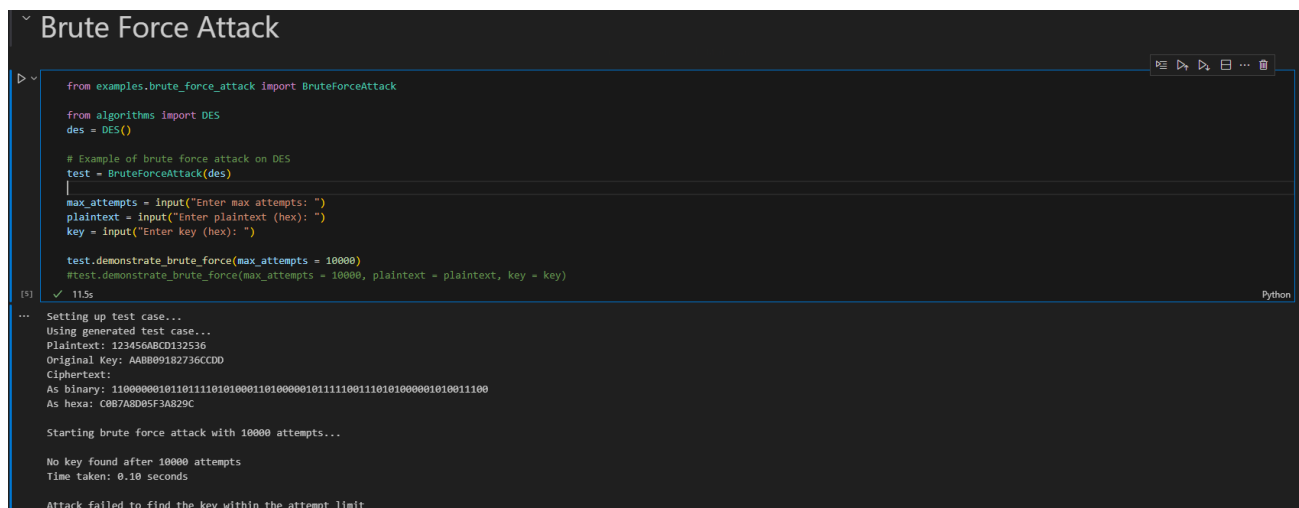
Figure 3.8: Whirlpool Hash

## 3.2 Brute-Force Attack

The brute force attack was set to a maximum of **10,000 attempts** (*You can change this argument to any value you wish, but 10,000 was used here for theoretical testing*) due to the limitations of the computational resources of a personal computer. The attack was unable to find the key within the allowed attempts.

- **Attempts Limit:** 10,000
- **Time Taken:** 0.10 seconds
- **Result:** No key found after 10,000 attempts.

This attempt limit was imposed to avoid excessive computational time and resource usage. To successfully complete the brute force attack, the number of attempts would need to cover the entire **key space** of the encryption algorithm. In the worst-case scenario, the maximum number of attempts required would be equal to the total size of the key space, which depends on the key length of the algorithm.



```
Brute Force Attack

from examples.brute_force_attack import BruteForceAttack

from algorithms import DES
des = DES()

# Example of brute force attack on DES
test = BruteForceAttack(des)

max_attempts = input("Enter max attempts: ")
plaintext = input("Enter plaintext (hex): ")
key = input("Enter key (hex): ")

test.demonstrate_brute_force(max_attempts = 10000)
#test.demonstrate_brute_force(max_attempts = 10000, plaintext = plaintext, key = key)

[5] ✓ 11.5s Python

...
Setting up test case...
Using generated test case...
Plaintext: 123456ABCD132536
Original Key: AABBB9182736CCDD
Ciphertext:
As binary: 110000001011011110101000110100000101111001110101000001010011100
As hexa: C0B7ABD05F3A829C

Starting brute force attack with 10000 attempts...

No key found after 10000 attempts
Time taken: 0.10 seconds

Attack failed to find the key within the attempt limit
```

Figure 3.9: Brute-Force Attack

## 3.3 Timing Attack

The attack proceeds by guessing each byte of the key one by one based on the time taken for encryption operations. The attack successfully guessed each byte of the key, building the key progressively.

However, it's important to note that the success of the timing attack depends on various factors, including the precision of the computational timing and the ability to distinguish slight differences in time intervals. Therefore, the ability to recover the key is not always guaranteed and depends on the system's ability to execute the attack with enough precision.

```

Timing Attack

from examples.timing_attack import TimingAttack

test = TimingAttack(des)

max_attempts = input("Enter max attempts: ")
plaintext = input("Enter plaintext (hex): ")
key = input("Enter key (hex): ")

test.demonstrate_timing_attack(max_attempts = 1000)
#test.demonstrate_timing_attack(max_attempts = 10000, plaintext = plaintext, key = key)

Generating test case...
Plaintext: 123456ABCDEF123456
Original key: AAB09182736CCDD
Ciphertext:
As binary 110000001011011110101000110100000101111001110101000001010011100
As hex C0B7A0D05F3A829C

Starting timing attack with 1000 chars...
Guessing byte 1...
Guessed char: A, Current key: A
Guessing byte 1...
Guessed char: A, Current key: AA
Guessing byte 2...
Guessed char: B, Current key: AAB
Guessing byte 2...
Guessed char: B, Current key: AAB
Guessing byte 3...
Guessed char: 0, Current key: AAB0
Guessing byte 3...
Guessed char: 0, Current key: AAB0
Guessing byte 4...
Guessed char: 1, Current key: AAB091
Guessing byte 4...
Guessed char: 0, Current key: AAB0918

```

Figure 3.10: Timing Attack code

```

Starting timing attack with 1000 chars...
Guessing byte 1...
Guessed char: A, Current key: A
Guessing byte 1...
Guessed char: A, Current key: AA
Guessing byte 2...
Guessed char: B, Current key: AAB
Guessing byte 2...
Guessed char: B, Current key: AAB
Guessing byte 3...
Guessed char: 0, Current key: AAB0
Guessing byte 3...
Guessed char: 0, Current key: AAB0
Guessing byte 4...
Guessed char: 1, Current key: AAB091
Guessing byte 4...
Guessed char: 8, Current key: AAB0918
Guessing byte 5...
Guessed char: 2, Current key: AAB09182
Guessing byte 5...
Guessed char: 7, Current key: AAB091827
Guessing byte 6...
Guessed char: 3, Current key: AAB0918273
Guessing byte 6...
Guessed char: 6, Current key: AAB09182736
Guessing byte 7...
Guessed char: C, Current key: AAB09182736C
Guessing byte 7...
Guessed char: C, Current key: AAB09182736CC
Guessing byte 8...
Guessed char: D, Current key: AAB09182736CCD
Guessing byte 8...
Guessed char: D, Current key: AAB09182736CCDD

Key found after guessing 16 chars!
Time taken: 215.02 seconds

Success! Found key: AAB09182736CCDD

```

Figure 3.11: Timing Attack result, in this time we have successfully found the key

# Conclusion

In this project, we explored and re-implemented several widely-used cryptographic algorithms. By re-creating these algorithms, we gained a deeper understanding of their inner workings and the computational aspects that contribute to their security and efficiency. This hands-on approach provided valuable insights into the theory behind these algorithms and the practical challenges involved in implementing them.

The project demonstrated how cryptographic algorithms function at a low level, emphasizing the importance of encryption/decryption processes and optimization techniques. We also explored potential vulnerabilities, such as the brute force and timing attacks, and observed the computational complexity of breaking various encryption schemes.

The focus of this project was on the re-implementation and simulation of classical cryptographic algorithms for educational purposes, which not only enhances practical knowledge but also provides critical theoretical insights, benefitting anyone interested in learning about cryptography, secure systems, and the fundamental principles behind modern encryption techniques.

## Bibliography

- [1] Ha Duyen Trung, *Lecture Notes on Cryptography*, HUST, 2025.
- [2] Behrouz A. Forouzan, *Introduction to Cryptography and Network Security*, McGraw-Hill, 2011.
- [3] GeeksforGeeks