

VISVESVARAYA TECHNOLOGICAL UNIVERSITY

“JnanaSangama”, Belgaum -590014, Karnataka.



CRYPTOGRAPHY AND NETWORK SECURITY AAT REPORT

On

Implementation of encryption and decryption using AES algorithm,
Cryptanalysis of AES using Crypt tool 1 and demonstration of ECC digital
signature on crypt tool 1

Submitted by

Kanjika Singh (1BM21CS086)

Neha Bhaskar Kamath (1BM21CS113)

Under the Guidance of

Prof. Syed Akram

Assistant Professor, BMSCE

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING



B. M. S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

November-2023 to February-2024

B. M. S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering



CERTIFICATE

This is to certify that the AAT work entitled “**Implementation of encryption and decryption using AES algorithm, Cryptanalysis of AES using Crypt tool 1 and demonstration of ECC digital signature on crypt tool 1**” is carried out by **Kanjika Singh(1BM21CS086)** and **Neha Bhaskar Kamath(1BM21CS113)** who are bonafide students of **B. M. S. College of Engineering**. It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visveswaraya Technological University, Belgaum during the year 2023-24. The AAT report has been approved as it satisfies the academic requirements in respect of **Cryptography(22CS5PCCRP)** work prescribed for the said degree.

Signature of the Guide

Prof. Syed Akram

Assistant Professor
BMSCE, Bengaluru

Signature of the HOD

Dr. Jyothi Nayak

Prof. & Head, Dept. of CSE
BMSCE, Bengaluru

B. M. S. COLLEGE OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND
ENGINEERING



DECLARATION

We, Kanjika Singh (1BM21CS086) and Neha Bhaskar Kamath (1BM21CS113), students of 5th Semester, B.E, Department of Computer Science and Engineering, B. M. S. College of Engineering, Bangalore, here by declare that, this AAT entitled " **Implementation of encryption and decryption using AES algorithm, Cryptanalysis of AES using Crypt tool 1 and demonstration of ECC digital signature on crypt tool 1**" has been carried out by us under the guidance of Prof. Syed Akram, Assistant Professor, Department of CSE, B. M. S. College of Engineering, Bangalore during the academic semester November-2023-February-2024.

We also declare that to the best of our knowledge and belief, the development reported here is not from part of any other report by any other students.

Signature:

Kanjika Singh (1BM21CS086)

Neha Bhaskar Kamath (1BM21CS113)

TABLE OF CONTENTS

| Chapter No | Title | Page No |
|-------------------|--|----------------|
| 1 | Introduction and Motivation | 5 |
| 1.1 | Introduction | 5 |
| 1.2 | Motivation | 5 |
| 2 | Literature Survey | 6-7 |
| 3 | Methodology | 8-17 |
| 3.1 | AES transformations and Key Expansion | 8-12 |
| 3.2 | Algorithm for AES | 13 |
| 3.3 | Modes of operation | 14 |
| 3.4 | ECC Digital signature | 15-16 |
| 3.5 | Work flow-diagram | 17 |
| 4 | Implementation and Results | 18-31 |
| 4.1 | Code for AES-128 encyption and decryption using CBC mode | 18-26 |
| 4.2 | Output | 27 |
| 4.3 | Cryptanalysis on AES-128,CBC mode using Crypt-tool 1 | 28-29 |
| 4.4 | ECC digital signature using Crypt-tool 1 | 30-31 |
| 5 | Conclusion and Future work | 32 |
| 5.1 | Conclusion | 32 |
| 5.2 | Future work | 32 |

CHAPTER 1

1.1 INTRODUCTION

In the realm of information security, the robustness of encryption algorithms plays a pivotal role in safeguarding sensitive data from unauthorized access. **The Advanced Encryption Standard (AES)** algorithm has emerged as a cornerstone in securing digital communication and data storage. As part of our project, we perform the encryption and decryption processes using the AES algorithm. Our approach involves a hands-on coding implementation to gain practical insights into the intricacies of AES encryption and decryption.

Subsequently, we use Crypt Tool 1 to perform cryptanalysis on the AES algorithm, quantifying the time required to breach its encryption key.

In addition to AES, this project ventures into the domain of Elliptic Curve Cryptography (ECC) for digital signatures using Crypt Tool 1.

AES is a non-Feistel cipher that encrypts and decrypts a data block of 128 bits. It uses 10, 12, or 14 rounds. The key size, which can be 128, 192, or 256 bits, depends on the number of rounds. This means that there are three different AES versions which are referred to as AES-128, AES-192, and AES-256. However, the round keys, which are created by the key-expansion algorithm are always 128 bits.

ECC, known for its efficiency and strength, further enhances the security landscape by providing secure and efficient means of authentication.

1.2 MOTIVATION

In today's interconnected digital landscape, the exchange of sensitive information over networks is commonplace. Encryption serves as the first line of defense, ensuring that data remains confidential and secure during transmission and storage. The escalating frequency and sophistication of cyber attacks underscore the need for robust cryptographic solutions.

AES, being a widely adopted encryption standard, plays a pivotal role in information security. Its proven strength and efficiency make it a cornerstone for securing sensitive data in various applications, from online transactions to communication platforms.

In addition to this, the exploration of Elliptic Curve Cryptography (ECC) for digital signatures is motivated by the increasing relevance of secure authentication. ECC's efficiency and strength make it a compelling choice for digital signatures, addressing the contemporary need for reliable methods of online verification.

The project offers educational value by bridging the gap between theoretical concepts and practical application. Through hands-on coding and experimentation, we aim to enhance our own understanding of encryption, decryption, and digital signatures.

Chapter 2

2 LITERATURE SURVEY

[1] In the study, the authors evaluate the performance of both AES-128 and AES-256 based on various parameters, including encryption and decryption times. The results indicate that the encryption and decryption times are influenced by the key size, with AES-256 exhibiting longer processing times due to its larger key size and more complex architecture. Specifically, the research findings show that when using strong and longer keys, the encryption time for both AES-128 and AES-256 increases. However, AES-256 demonstrates greater robustness and security due to its larger key size and architecture, resulting in longer decryption times compared to AES-128, which has an average decryption time of around 30 milliseconds.

[2] This paper analyzes the post-quantum security of AES. AES is the most popular and widely used block cipher, established as the encryption standard by the NIST in 2001. In order to determine the new security margin, i.e., the lowest number of non-attacked rounds in time less than 2^{128} encryptions, the paper provides generalized and quantized versions of the best known cryptanalysis on reduced-round AES, as well as a discussion on attacks that don't seem to benefit from a significant quantum speed-up. The paper proposes a new framework for structured search that encompasses both the classical and quantum attacks, and allows to efficiently compute their complexity. The best attack proposed in the paper is a quantum Demirci-Selçuk meet-in-the-middle attack. Unexpectedly, using the ideas underlying its design principle also enables to obtain new, counter-intuitive classical TMD trade-offs. In particular, memory can be reduced in some attacks against AES-256 and AES-128. One of the building blocks of the attacks is solving efficiently the AES S-Box differential equation, with respect to the quantum cost of a reversible S-Box. Judging by the results obtained so far, AES seems a resistant primitive in the postquantum world as well as in the classical one, with a bigger security margin with respect to quantum generic attacks.

Main results:

- Square attacks: The paper proposes quantum square attacks on 6-round AES-128, 7-round AES-192 and 7-round AES-256.
- Demirci-Selçuk Meet-in-the-Middle(DS-MITM): By rewriting and reordering the phases of the attack, the paper proposes a quantum attack on 8 rounds of AES-256, hence effectively speeding up the classical attack by nearly a quadratic factor. In the classical setting, DS-MITM provides the best single-key attacks, along with impossible differentials. It covers up to 9 rounds of AES-256.
- A detailed evaluation of the cost of Grover exhaustive search, that defines the security of the corresponding AES instances.
- Quantum tools to efficiently leverage the differential properties of the AES S-Box with a very small memory, a building block which could find applications outside the scope of this paper, and justification on our extensive usage of nested Grover procedures.
- New classical TMD trade-offs for DS-MITM attacks: the ideas that allow to accelerate quantumly this type of attacks can also be applied classically on AES-256 and AES-128, giving reductions in memory needs and new tradeoffs.

[3] Elliptic curve cryptography has the characteristics of high-security strength and low computational complexity. Elliptic curve cryptography relies on point multiplication, which is the most time-consuming part of the encryption and decryption process. The Elliptic Curve Cryptosystem is currently the most famous and potential public key cryptosystem. It is proposed based on the computational difficulty of discrete logarithms on the elliptic curve, and its security research is an important research area in academia. This paper analyzes the security of elliptic cryptographic curves from the performance comparison of ECC and RSA. Moreover, this paper implements RSA and ECC using random private keys, and the sample data input is 64-bit, 8-bit, and 256-bit. Experiments are done on MATLAB R2008a on an Intel Pentium dual-core processor. The findings reveal that RSA is efficient at encryption, but sluggish at decryption, whereas ECC is slow at encryption but efficient at decryption. Overall, ECC outperforms RSA in terms of efficiency and security. ECC surpasses RSA in terms of operational security and efficiency, according to this research.

Chapter 3

3.METHODOLOGY

3.1 AES Transformations and Key expansion

AES uses several rounds in which each round is made of several stages. Data block, a group of 128 bits, is transformed from one stage to another. However, before and after each stage, the data block is referred to as a state which is represented as a matrix of 4×4 bytes.

AES uses four types of transformations that are invertible.

At the encryption site, they are called **SubBytes(substitution)**, **ShiftRows(permutation)**, **MixColumns(mixing)**, and **AddRoundKey(key-adding)**. However, the last round has only three transformations.(Mixing transformation is missing).

At the decryption site, the inverse transformations are used: **InvSubByte**, **InvShiftRows**, **InvMixColumns**, and **AddRoundKey** (this one is self-invertible).

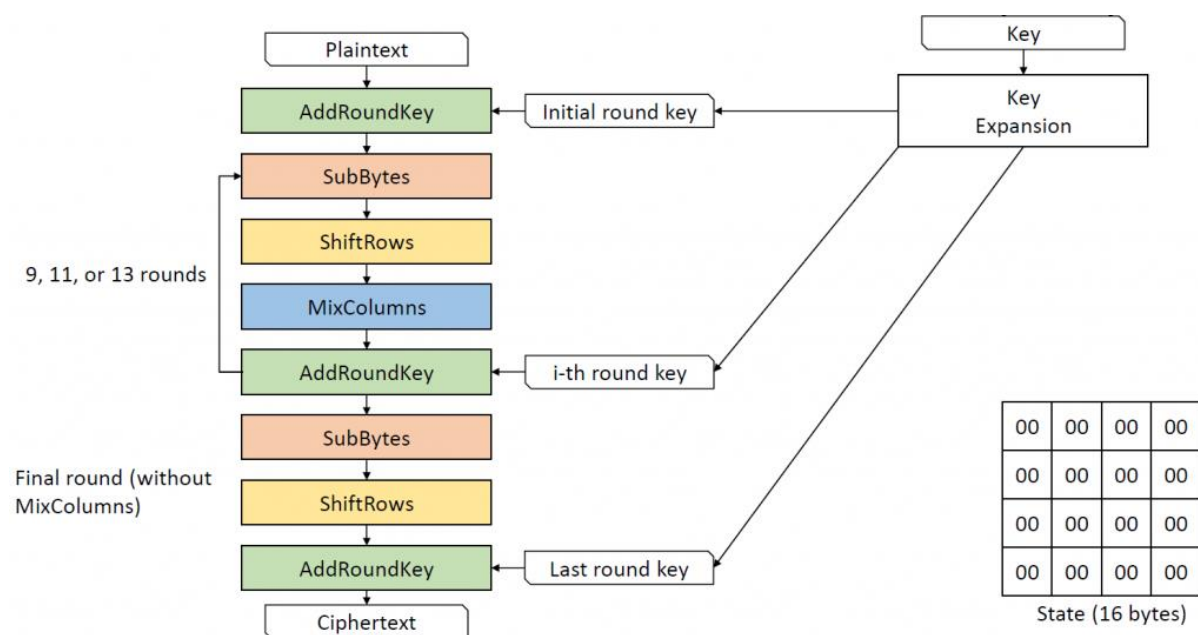


Fig 1.1 AES Structure at the encryption site

Transformations:

SubBytes

In SubBytes, each byte in the data block undergoes a non-linear substitution. This involves replacing each byte with another from a predefined substitution table (S-box). This operation involves 16 independent byte-to-byte transformations.

InvSubBytes is the inverse of SubBytes.

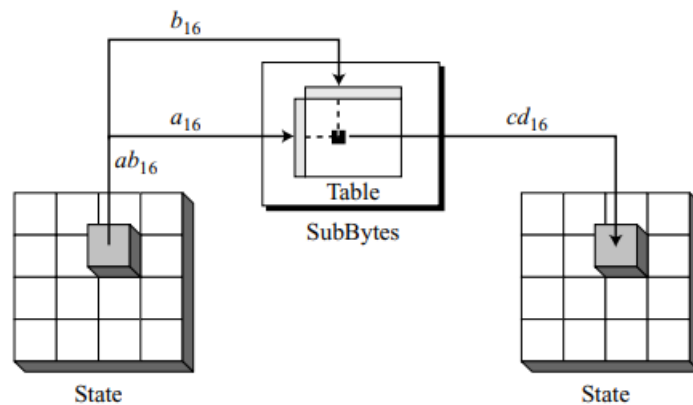


Fig 1.2 SubBytes transformation

ShiftRows

Each row in the state is shifted to the left. The number of shifts depends on the row number (0, 1, 2, or 3) of the state matrix. This means the row 0 is not shifted at all and the last row is shifted three bytes.

InvShiftRows shifts each row to the right. The number of shifts is the same as the row number (0, 1, 2, and 3) of the state matrix.

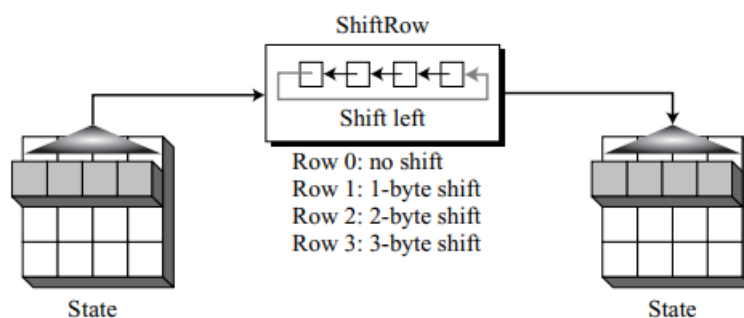


Fig 1.3 ShiftRows transformation

MixColumns

The MixColumns transformation operates at the column level; it transforms each column of the state to a new column. The transformation is actually the matrix multiplication of a state column by a constant square matrix.

The **InvMixColumns** transformation is basically the same as the MixColumns transformation. If the two constant matrices are inverses of each other, the two transformations are inverses of each other.

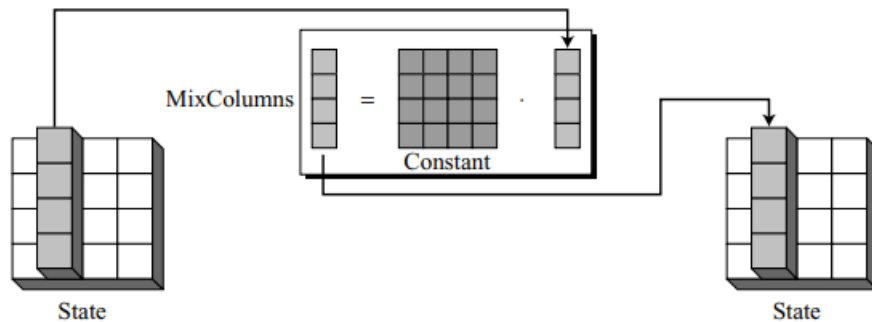


Fig 1.4 MixColumns transformation

AddRoundKey

AddRoundKey also proceeds one column at a time. It adds a round key word with each state column matrix. This operation is the inverse of itself.

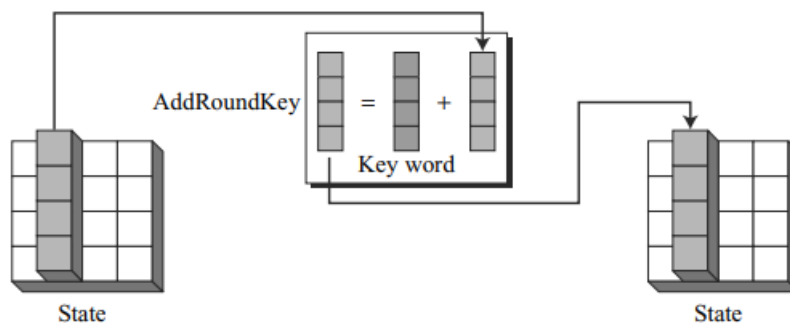


Fig 1.5 AddRoundKey transformation

Key Expansion for AES-128

AES uses a key-expansion process. If the number of rounds is N_r , the key-expansion routine creates $N_r + 1$ 128-bit round keys from one single 128-bit cipher key. This means that in AES-128, **44 words** are made from the original key.

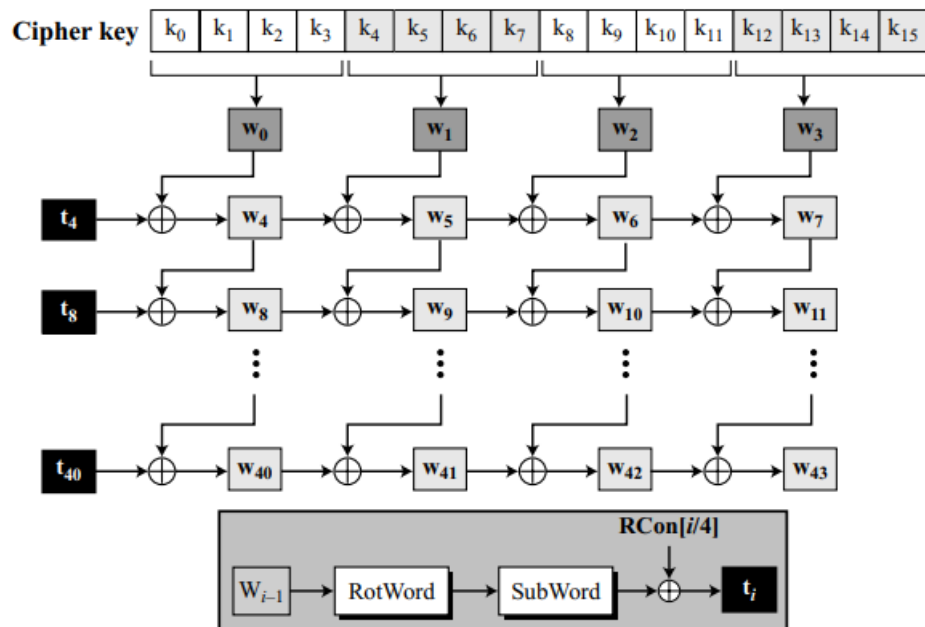


Fig 1.6 key Expansion in AES-128

Key Expansion Algorithm

KeyExpansion ([key0 to key15], [w0 to w43])

```
{
  for (i = 0 to 3)
    wi ← key4i
    + key4i+1 + key4i+2 + key4i+3
  for (i = 4 to 43)
    {
      if (i mod 4 ≠ 0) wi ← wi-1 + wi-4
    else
      {
        t ← SubWord (RotWord (wi-1)) ⊕ RConi/4 // t is a temporary word
        wi ← t + wi-4
      }
    }
}
```

```

}
}
}

```

RotWord: The RotWord (rotate word) routine is similar to the ShiftRows transformation, but it is applied to only one row. The routine takes a word as an array of four bytes and shifts each byte to the left with wrapping.

SubWord: The SubWord (substitute word) routine is similar to the SubBytes transformation, but it is applied only to four bytes. The routine takes each byte in the word and substitutes another byte for it.

| <i>Round</i> | <i>Constant (RCon)</i> | <i>Round</i> | <i>Constant (RCon)</i> |
|--------------|-------------------------------------|--------------|-------------------------------------|
| 1 | (<u>01</u> 00 00 00) ₁₆ | 6 | (<u>20</u> 00 00 00) ₁₆ |
| 2 | (<u>02</u> 00 00 00) ₁₆ | 7 | (<u>40</u> 00 00 00) ₁₆ |
| 3 | (<u>04</u> 00 00 00) ₁₆ | 8 | (<u>80</u> 00 00 00) ₁₆ |
| 4 | (<u>08</u> 00 00 00) ₁₆ | 9 | (<u>1B</u> 00 00 00) ₁₆ |
| 5 | (<u>10</u> 00 00 00) ₁₆ | 10 | (<u>36</u> 00 00 00) ₁₆ |

Fig 1.7 RCon constants

3.2 Algorithm for AES-128

3.2.1 Algorithm for encryption

```
Cipher (InBlock [16], OutBlock[16], w[0 ... 43])
{
  BlockToState (InBlock, S)
  S  $\leftarrow$  AddRoundKey (S, w[0...3])
  for (round = 1 to 10)
  {
    S  $\leftarrow$  SubBytes (S)
    S  $\leftarrow$  ShiftRows (S)
    if (round  $\neq$  10) S  $\leftarrow$  MixColumns (S)
    S  $\leftarrow$  AddRoundKey (S, w[4  $\times$  round, 4  $\times$  round + 3])
  }
  StateToBlock (S, OutBlock);
}
```

2.2.2 Algorithm for decryption

```
Inverse_Cipher (InBlock [16], OutBlock[16], w[0 ... 43])
{
  BlockToState (OutBlock, S)
  S  $\leftarrow$  AddRoundKey (S, w[40...43])
  for (round = 9 to 0)
  {
    S  $\leftarrow$  Inv_ShiftRows (S)
    S  $\leftarrow$  Inv_SubBytes (S)
    S  $\leftarrow$  AddRoundKey (S, w[4  $\times$  round, 4  $\times$  round + 3])
    if (round  $\neq$  0) S  $\leftarrow$  Inv_MixColumns (S)
  }
  StateToBlock (S, InBlock);
}
```

3.3 Modes of Operation

AES-128 encrypts and decrypts a block of 128 bits. In real life applications, the text to be enciphered is of variable size and normally much larger than 128 bits. Modes of operation have been devised to encipher text of any size using AES.

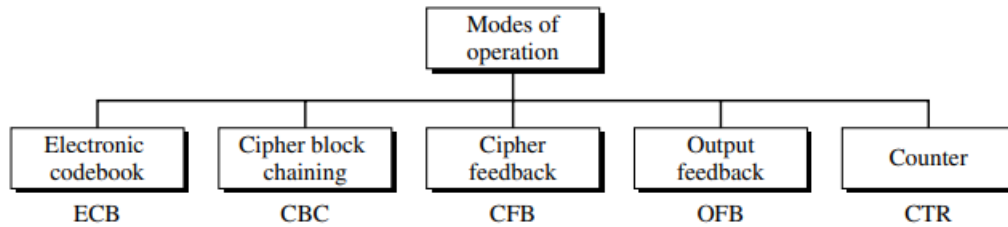


Fig 1.8 Five modes of operation

3.3.1 Cipher block chaining (CBC)

In CBC mode, each plaintext block is **exclusive-ored** with the previous ciphertext block before being encrypted. When a block is enciphered, the block is sent, but a copy of it is kept in memory to be used in the encryption of the next block. The reader may wonder about the initial block. There is no ciphertext block before the first block. In this case, a phony block called the **initialization vector (IV)** is used. The sender and receiver agree upon a specific predetermined IV.

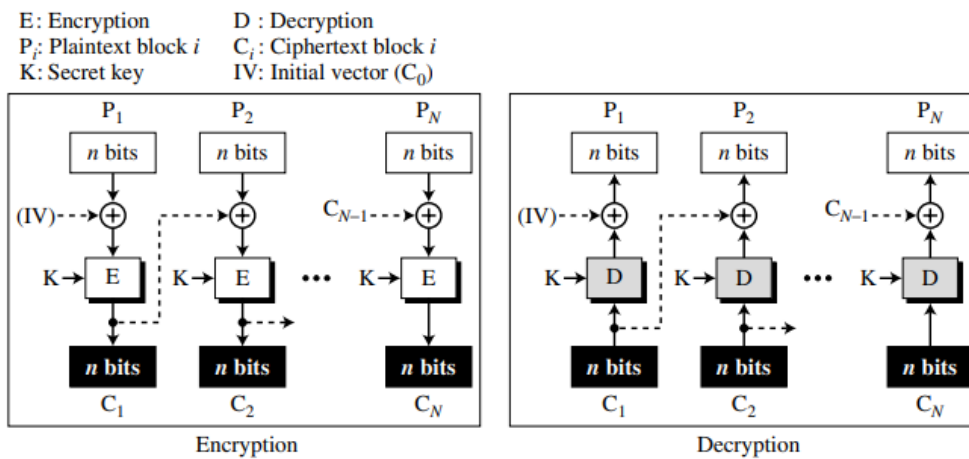


Fig 1.9 CBC mode of operation

3.4 ECC Digital Signature

3.4.1 Digital Signature

A digital signature is a cryptographic technique that provides a way to verify the authenticity and integrity of a digital message, document, or transaction. It involves the use of a private key to create the signature, and a corresponding public key to verify the signature.

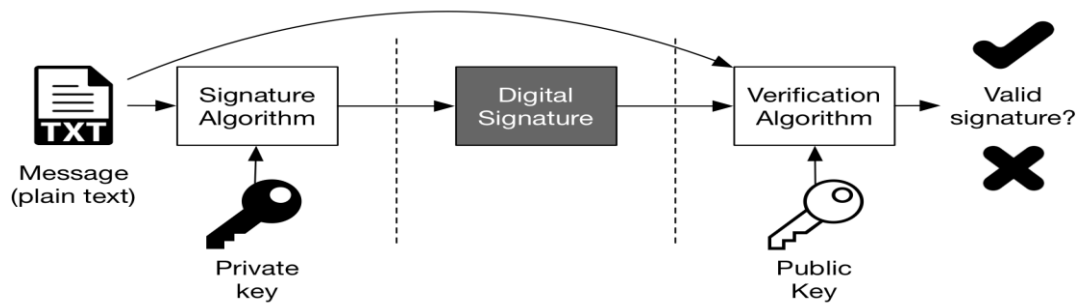


Fig 1.10 Basic Structure of Digital Signature

3.4.2 ECDSA (Elliptic Curve Digital Signature Algorithm)

ECDSA is a cryptographically secure digital signature scheme, based on the elliptic-curve cryptography. ECDSA relies on the math of the cyclic groups of elliptic curves over finite fields and on the difficulty of the ECDLP problem (Elliptic curve discrete logarithmic problem).

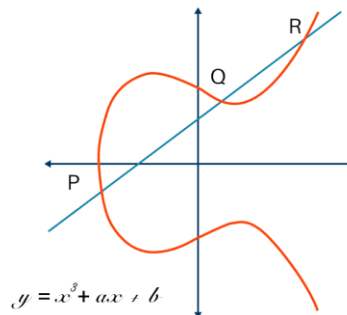


Fig 1.11 ECC curve

ECDSA has two phases: **Signature generation** at the sender's site and **Signature verification** at the receiver's site.

Signature generation

This is the process of creating a unique, signature for a digital message or document using a private key.

Parameters used:

- Curve equation- $y^2=x^3+ax+b$
- Base point (G)- point on elliptic curve
- Order (n)- defines the length of the private keys
- Hash function- applied to the message being signed.
- K- random integer
- Private key(d)- random integer within the range $[1, n-1]$
- Public key(Q) - $Q = d * G$

For each signature, a point (x, y) on the elliptic curve is calculated by performing scalar multiplication of the base point 'G' by the random number 'k.' $(x_1, y_1)=k*G$.

ECDSA produces a pair of values (r, s) as the resulting digital signature where:

- $r = x\text{-coordinate of } (k * G)$
- $s = (k^{-1} * (H(\text{message}) + d * r)) \bmod n$

Signature verification

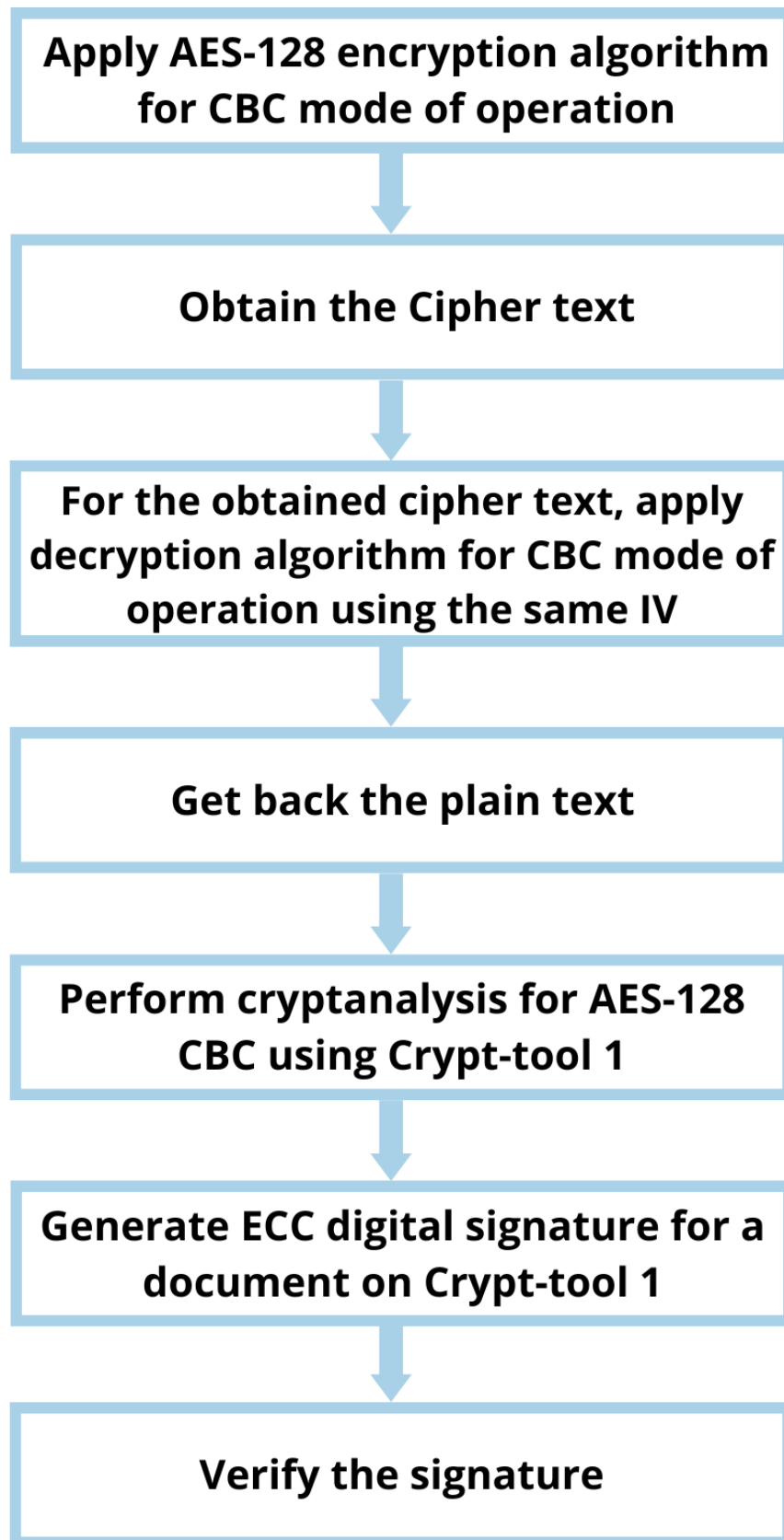
Input parameters:

- Public key (Q)
- Signature components (r, s)
- Hash function of the original message

Steps:

- Calculate $w \equiv s^{-1} \bmod n$
- Find $u_1 \equiv (H \cdot w) \bmod n$ and $u_2 \equiv (r \cdot w) \bmod n$
- Find point $U = u_1 \cdot G + u_2 \cdot Q$
- Verify that $r \equiv (x_u \bmod n)$
- If r matches the x-coordinate of the calculated point U, then the signature is considered valid. Otherwise, it is invalid.

3.5 Work flow



Chapter 4

4 IMPLEMENTATION AND RESULTS

4.1 Code for AES-128 encryption and decryption using CBC mode

```
import os

def sub_bytes(s):
    for i in range(4):
        for j in range(4):
            s[i][j] = s_box[s[i][j]]

def inv_sub_bytes(s):
    for i in range(4):
        for j in range(4):
            s[i][j] = inv_s_box[s[i][j]]

def shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[1][1], s[2][1], s[3][1], s[0][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[3][3], s[0][3], s[1][3], s[2][3]

def inv_shift_rows(s):
    s[0][1], s[1][1], s[2][1], s[3][1] = s[3][1], s[0][1], s[1][1], s[2][1]
    s[0][2], s[1][2], s[2][2], s[3][2] = s[2][2], s[3][2], s[0][2], s[1][2]
    s[0][3], s[1][3], s[2][3], s[3][3] = s[1][3], s[2][3], s[3][3], s[0][3]

def add_round_key(s, k):
    for i in range(4):
        for j in range(4):
            s[i][j] ^= k[i][j]
```

```
xtime = lambda a: (((a << 1) ^ 0x1B) & 0xFF) if (a & 0x80) else (a << 1)
```

```
def mix_single_column(a):
```

```
    # see Sec 4.1.2 in The Design of Rijndael
```

```
    t = a[0] ^ a[1] ^ a[2] ^ a[3]
```

```
    u = a[0]
```

```
    a[0] ^= t ^ xtime(a[0] ^ a[1])
```

```
    a[1] ^= t ^ xtime(a[1] ^ a[2])
```

```
    a[2] ^= t ^ xtime(a[2] ^ a[3])
```

```
    a[3] ^= t ^ xtime(a[3] ^ u)
```

```
def mix_columns(s):
```

```
    for i in range(4):
```

```
        mix_single_column(s[i])
```

```
def inv_mix_columns(s):
```

```
    # see Sec 4.1.3 in The Design of Rijndael
```

```
    for i in range(4):
```

```
        u = xtime(xtime(s[i][0] ^ s[i][2]))
```

```
        v = xtime(xtime(s[i][1] ^ s[i][3]))
```

```
        s[i][0] ^= u
```

```
        s[i][1] ^= v
```

```
        s[i][2] ^= u
```

```
        s[i][3] ^= v
```

```
    mix_columns(s)
```

```
r_con = (
```

```
    0x00, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40,
```

```
    0x80, 0x1B, 0x36, 0x6C, 0xD8, 0xAB, 0x4D, 0x9A,
```

```

    0x2F, 0x5E, 0xBC, 0x63, 0xC6, 0x97, 0x35, 0x6A,
    0xD4, 0xB3, 0x7D, 0xFA, 0xEF, 0xC5, 0x91, 0x39,
)

def bytes2matrix(text):
    """ Converts a 16-byte array into a 4x4 matrix. """
    return [list(text[i:i+4]) for i in range(0, len(text), 4)]

def matrix2bytes(matrix):
    """ Converts a 4x4 matrix into a 16-byte array. """
    return bytes(sum(matrix, []))

def xor_bytes(a, b):
    """ Returns a new byte array with the elements xor'ed. """
    return bytes(i^j for i, j in zip(a, b))

def inc_bytes(a):
    """ Returns a new byte array with the value increment by 1 """
    out = list(a)

    for i in reversed(range(len(out))):
        if out[i] == 0xFF:
            out[i] = 0
        else:
            out[i] += 1
            break

    return bytes(out)

def pad(plaintext):
    #Pads the given plaintext with PKCS#7 padding to a multiple of 16 bytes.
    padding_len = 16 - (len(plaintext) % 16)

```

```

padding = bytes([padding_len] * padding_len)

return plaintext + padding

def unpad(plaintext):
    #Removes a PKCS#7 padding, returning the unpadded text and ensuring the
    #padding was correct.

    padding_len = plaintext[-1]

    assert padding_len > 0

    message, padding = plaintext[:-padding_len], plaintext[-padding_len:]

    assert all(p == padding_len for p in padding)

    return message

def split_blocks(message, block_size=16, require_padding=True):
    assert len(message) % block_size == 0 or not require_padding

    return [message[i:i+16] for i in range(0, len(message), block_size)]

class AES:
    #Class for AES-128 encryption with CBC mode and PKCS#7

    rounds_by_key_size = { 16: 10, 24: 12, 32: 14}

    def __init__(self, master_key):
        # Initializes the object with a given key.

        assert len(master_key) in AES.rounds_by_key_size

        self.n_rounds = AES.rounds_by_key_size[len(master_key)]

        self._key_matrices = self._expand_key(master_key)

    def _expand_key(self, master_key):
        #Expands and returns a list of key matrices for the given master_key.

        # Initialize round keys with raw key material.

        key_columns = bytes2matrix(master_key)

```

```

iteration_size = len(master_key) // 4

i = 1

while len(key_columns) < (self.n_rounds + 1) * 4:

    # Copy previous word.

    word = list(key_columns[-1])

    # Perform schedule_core once every "row".

    if len(key_columns) % iteration_size == 0:

        # Circular shift.

        word.append(word.pop(0))

        # Map to S-BOX.

        word = [s_box[b] for b in word]

        # XOR with first byte of R-CON, since the others bytes of R-CON are 0.

        word[0] ^= r_con[i]

        i += 1

    elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:

        # Run word through S-box in the fourth iteration when using a

        # 256-bit key.

        word = [s_box[b] for b in word]

        # XOR with equivalent word from previous iteration.

        word = xor_bytes(word, key_columns[-iteration_size])

        key_columns.append(word)

    # Group key words in 4x4 byte matrices.

    return [key_columns[4*i : 4*(i+1)] for i in range(len(key_columns) // 4)]

```

```

def encrypt_block(self, plaintext):

    #Encrypts a single block of 16 byte long plaintext.

    assert len(plaintext) == 16

    plain_state = bytes2matrix(plaintext)

    add_round_key(plain_state, self._key_matrices[0])

    for i in range(1, self.n_rounds):

        sub_bytes(plain_state)

        shift_rows(plain_state)

        mix_columns(plain_state)

        add_round_key(plain_state, self._key_matrices[i])

    sub_bytes(plain_state)

    shift_rows(plain_state)

    add_round_key(plain_state, self._key_matrices[-1])

    return matrix2bytes(plain_state)

def decrypt_block(self, ciphertext):

    #Decrypts a single block of 16 byte long ciphertext.

    assert len(ciphertext) == 16

    cipher_state = bytes2matrix(ciphertext)

    add_round_key(cipher_state, self._key_matrices[-1])

    inv_shift_rows(cipher_state)

    inv_sub_bytes(cipher_state)

```

```

for i in range(self.n_rounds - 1, 0, -1):

    add_round_key(cipher_state, self._key_matrices[i])

    inv_mix_columns(cipher_state)

    inv_shift_rows(cipher_state)

    inv_sub_bytes(cipher_state)

add_round_key(cipher_state, self._key_matrices[0])

return matrix2bytes(cipher_state)

def encrypt_cbc(self, plaintext, iv=os.urandom(16)):

    #Encrypts `plaintext` using CBC mode and PKCS#7 padding, with the given
    #initialization vector (iv).

    plaintext = plaintext.encode('utf-8')

    assert len(iv) == 16

    plaintext = pad(plaintext)

    blocks = []

    previous = iv

    for plaintext_block in split_blocks(plaintext):

        # CBC mode encrypt: encrypt(plaintext_block XOR previous)

        block = self.encrypt_block(xor_bytes(plaintext_block, previous))

        blocks.append(block)

        previous = block

    return b".join(blocks).hex(),iv.hex()

```



```

def decrypt_cbc(self, ciphertext, iv):

    #Decrypts `ciphertext` using CBC mode and PKCS#7 padding, with the given
    #initialization vector (iv).

    ciphertext = bytes.fromhex(ciphertext)

    iv = bytes.fromhex(iv)

    assert len(iv) == 16

    blocks = []

    previous = iv

    for ciphertext_block in split_blocks(ciphertext):

        # CBC mode decrypt: previous XOR decrypt(ciphertext)

        blocks.append(xor_bytes(previous, self.decrypt_block(ciphertext_block)))

        previous = ciphertext_block

    return unpad(b''.join(blocks)).decode('utf-8')


import os

import gradio as gr

key = b'SecretKey1234567'

# Generate a random IV

iv = os.urandom(16)

# print("iv: ",iv.hex())

aes = AES(key)

chatbot_input = [gr.components.Textbox(label="Enter Plaintext here")]

chatbot_output=[gr.components.Textbox(label="Encrypted Text in AES
CBC"),gr.components.Textbox(label = 'Initialization vector')]

```

```

iface=gr.Interface(fn=aes.encrypt_cbc,inputs=chatbot_input,
outputs=chatbot_output,title="Encryption Demonstration").queue()

iface.launch(debug=True, share = True)


import gradio as gr


key = b'SecretKey1234567'

# Generate a random IV

# print("iv: ",iv.hex())


# Create an AES object

aes = AES(key)

chatbot_input      =      [gr.components.Textbox(label="Enter      Cipher      text
here"),gr.components.Textbox(label='Enter Initialization Vector')]

chatbot_output = [gr.components.Textbox(label="Decrypted Text in AES CBC")]

iface      =      gr.Interface(fn=aes.decrypt_cbc,      inputs=chatbot_input,
outputs=chatbot_output,title="Decrypt Demonstration").queue()

iface.launch(debug=True, share = True)


# Get ciphertext and IV from the user

input_ciphertext = bytes.fromhex(input("Enter the ciphertext: "))

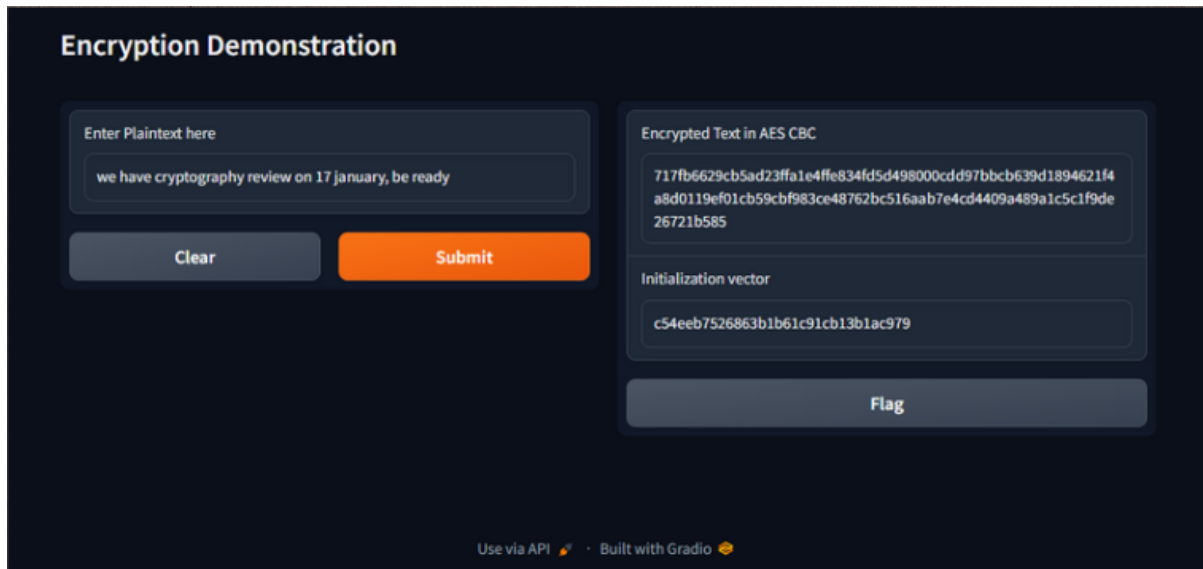
input_iv = bytes.fromhex(input("Enter the IV: "))

decrypted_text = aes.decrypt_cbc(input_ciphertext, input_iv)

print("Decrypted Text:", decrypted_text.decode('utf-8'))

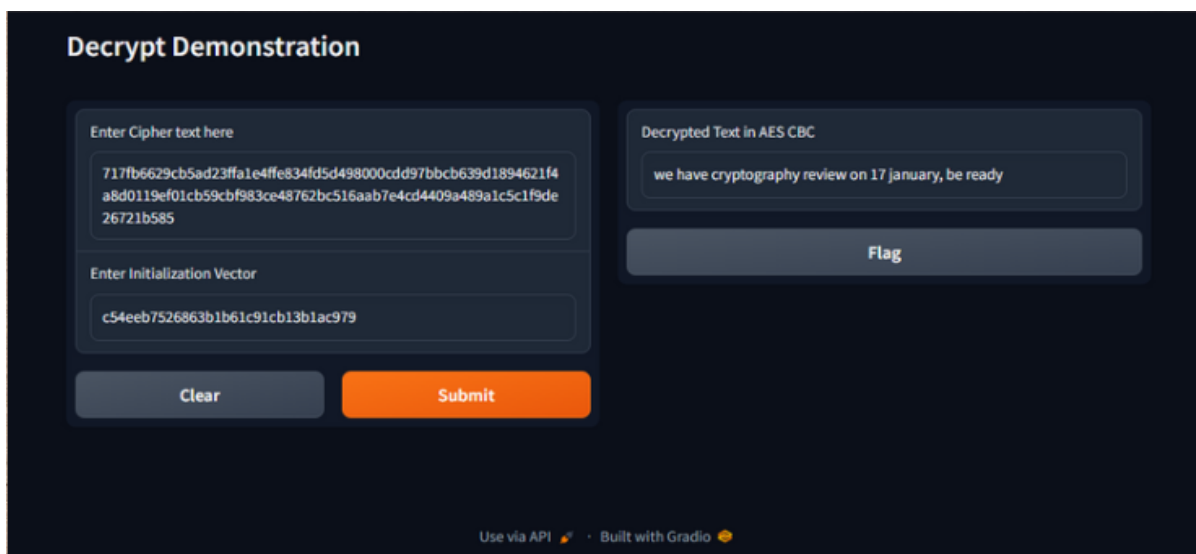
```

4.2 Output



The interface is titled "Encryption Demonstration". It features a dark theme with light gray text and buttons. On the left, there is a text input field labeled "Enter Plaintext here" containing the text "we have cryptography review on 17 january, be ready". Below this input are two buttons: a gray "Clear" button and an orange "Submit" button. On the right, there is a text area labeled "Encrypted Text in AES CBC" containing the hex string "717fb6629cb5ad23ffa1e4ffe834fd5d498000cdd97bbcb639d1894621f4a8d0119ef01cb59cbf983ce48762bc516aab7e4cd4409a489a1c5c1f9de26721b585". Below this is another text input field labeled "Initialization vector" containing the hex string "c54eeb7526863b1b61c91cb13b1ac979". At the bottom right of the right column is a gray "Flag" button. At the very bottom of the interface, there is a footer that says "Use via API" with a small icon and "Built with Gradio" with another small icon.

Fig 1.12 Encryption demonstration



The interface is titled "Decrypt Demonstration". It features a dark theme with light gray text and buttons. On the left, there is a text input field labeled "Enter Cipher text here" containing the hex string "717fb6629cb5ad23ffa1e4ffe834fd5d498000cdd97bbcb639d1894621f4a8d0119ef01cb59cbf983ce48762bc516aab7e4cd4409a489a1c5c1f9de26721b585". Below this input is another text input field labeled "Enter Initialization Vector" containing the hex string "c54eeb7526863b1b61c91cb13b1ac979". At the bottom left of the left column are two buttons: a gray "Clear" button and an orange "Submit" button. On the right, there is a text area labeled "Decrypted Text in AES CBC" containing the text "we have cryptography review on 17 january, be ready". Below this is a gray "Flag" button. At the very bottom of the interface, there is a footer that says "Use via API" with a small icon and "Built with Gradio" with another small icon.

Fig 1.13 Decryption demonstration

4.3 Cryptanalysis on AES-128,CBC mode using Crypt tool 1

Case 1: When the key is completely unknown

Plaintext: This is very confidential.

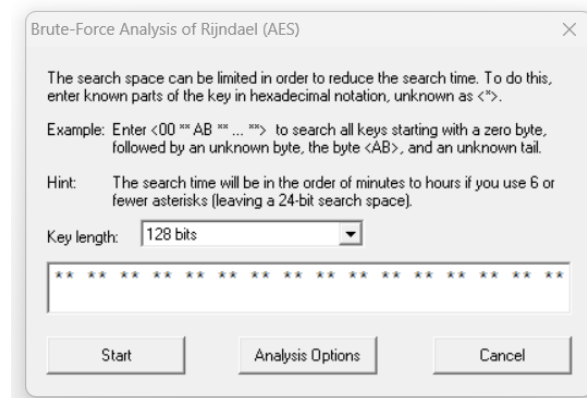


Fig 1.14 Key unknown

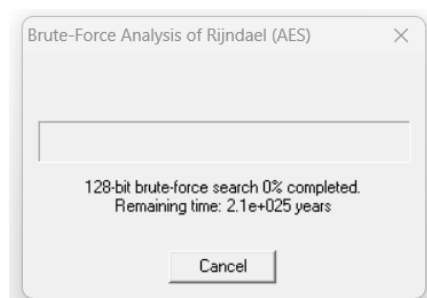


Fig 1.15 Time required to break the key

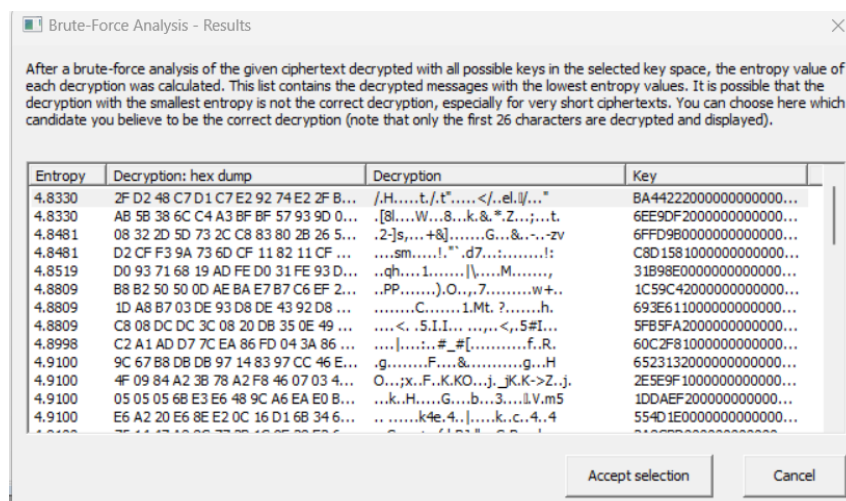


Fig 1.16 Some of the keys generated

Case 2: When first few bits of the key is known

Plaintext: This is very confidential.

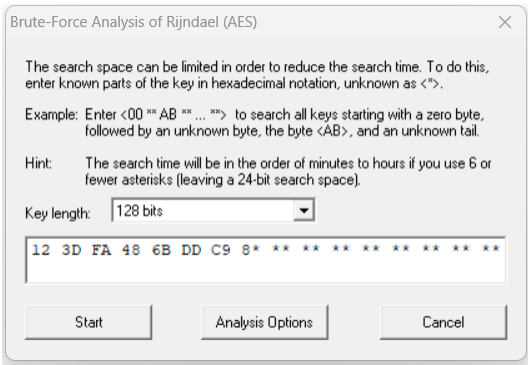


Fig 1.17 Initial few bits of the key known

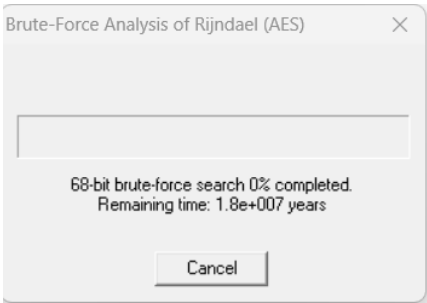


Fig 1.18 Time required to break the key

Brute-Force Analysis - Results

After a brute-force analysis of the given ciphertext decrypted with all possible keys in the selected key space, the entropy value of each decryption was calculated. This list contains the decrypted messages with the lowest entropy values. It is possible that the decryption with the smallest entropy is not the correct decryption, especially for very short ciphertexts. You can choose here which candidate you believe to be the correct decryption (note that only the first 26 characters are decrypted and displayed).

| Entropy | Decryption: hex dump | Decryption | Key |
|---------|---------------------------------------|-----------------------------------|-----------------------|
| 4.8002 | 7D 7A D2 81 50 63 2C 7D FB 43 7A ... | }z.,Pc,}.Cz...,M...,C^zz.z.,^... | 123DFA4868DDC9819F... |
| 4.8330 | 0E D6 AC 5C FC 17 CD 18 C5 05 53 ... | ...\\.....S\$A.>.s.....\$...s. | 123DFA4868DDC988C3... |
| 4.8330 | 63 95 8F AF 35 BE 63 E4 40 95 0C 3... | C...5.c.@..3.....R@~.....'... | 123DFA4868DDC982E1... |
| 4.8330 | 9A 21 21 1A 2C 2B 07 2E 07 A3 2... | .!l.,+....!.&;;.dl.&....!. | 123DFA4868DDC989DA... |
| 4.8519 | FB 8E 94 09 23 21 DA A5 FB 21 46 8... |#!...!F.#w...#.u.R...uT... .. | 123DFA4868DDC98698... |
| 4.8621 | 2B 96 84 2B 2B DC 29 A4 DF EB 88 D... | +..++..)......h.S..s..'+.S.Z... | 123DFA4868DDC985E4... |
| 4.8755 | 55 7F 6C 86 41 9F 55 94 CB 55 55 C... | U!l.A.U..UU.U!*)C/-..U.C...U... | 123DFA4868DDC98637... |
| 4.8809 | E8 27 32 34 34 1A 4B 4D F5 13 47 E... | .'244.KM.,G..\\...8.+4...4.[.&KI | 123DFA4868DDC9800D... |
| 4.8809 | 1A 81 14 66 CC C3 F4 F4 1A 59 B3 ... | ...f.....Y...CJE@E)..E.....E+ | 123DFA4868DDC986F2... |
| 4.8809 | C6 4D BC 17 9B 62 7A 4D 25 67 13 2... | .M...bzM%g.'%....%.%Mt...../... | 123DFA4868DDC98A00... |
| 4.8998 | C9 93 45 2B 00 44 45 C9 36 4C 45 0... | ..E+.DE.6LE..6..E.K.KQ.TQks.W..F | 123DFA4868DDC98C4F... |
| 4.8998 | D8 BC 2F DB E2 2F DB 09 D8 D9 F7 ... | ..f./.....V /.cV.3zbI). | 123DFA4868DDC98453... |
| 4.8998 | F4 B6 DF 91 4D F6 11 BD D5 4D B6 4... |M...M.MS..)*.....UU.U... | 123DFA4868DDC987BF... |

Accept selection Cancel

Fig 1.19 Some of the keys generated with the same initial bits of the key entered.

4.4 ECC Digital signature using crypt tool 1

4.4.1 Signature generation

Document:

This is very important and has to be kept hidden.

Signature generated:

```
Signature:      [c=]              731
329381525746332886589160536807875927
016109580471671178156422187031980
      [d=]              419290493303578
044305348728245137268017797500490831
319980287025507260201
      Signature length:  477
                        Algorithm:
      ECSP-DSA
Hash function:  SHA-1
      Key:      [S][Kanjika][
EC-prime239v1][1705422392]
      Message:  This is
      very important and has to be kept h
idden.
```

Analysis:

- **c** and **d** correspond to **r** and **s** of the resulting digital signature.
- Length of the signature: 477
- Algorithm used: ECDSA
- Hash function used: SHA-1
- Identifiers of key: [S][Kanjika][EC-prime239v1][1705422392]

4.4.2 Signature verification

Case 1: if corresponding(correct) key is used

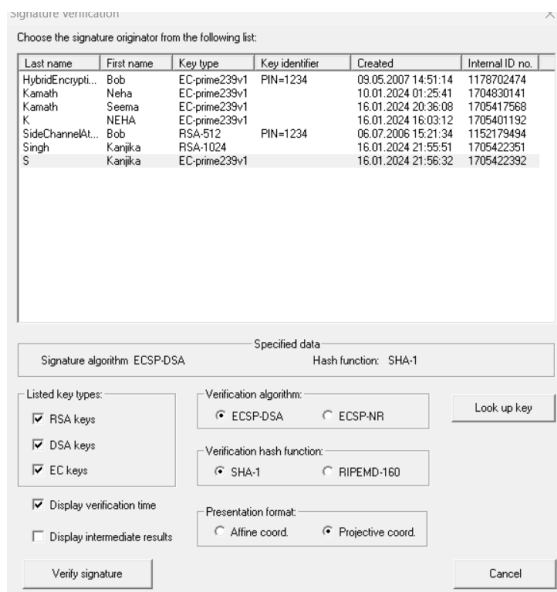


Fig 1.20 Signature valid

Case 2: if wrong key is used



Fig 1.21 Signature invalid

Chapter 5

5.1 Conclusion

In conclusion, this project has walked us through the inner working of AES encryption and decryption, offering a hands-on understanding of their application. The cryptanalysis using Crypt Tool 1 unveiled insights into the algorithm's strengths. Lastly, the ECC digital signature demonstration not only showcased the practical adaptability of cryptographic tools but also highlighted their pivotal role in ensuring the integrity and authenticity of digital signatures.

5.2 Future work

In terms of future work, we like to extend our implementation of AES encryption and decryption to the other modes of operation like ECB, CFB, OFB and CTR.

We like to optimize the performance of the AES encryption and decryption implementations.

We like to integrate AES within a larger cryptographic protocol, such as a secure communication or storage system which allows for a more holistic approach to end-to-end security.

For ECC-based digital signatures, future work could involve exploring variations or enhancements to the existing algorithms. This might include experimenting with different elliptic curves, key sizes, or hash functions to evaluate their impact on both security and efficiency.

We like to explore crypt tool 1 more and assess its effectiveness with other cryptographic algorithms and in various scenarios.

References

1. [A Comparative Study on AES 128 BIT AND AES 256 BIT](#)

Toa Bi Irie Guy-Cedric^{1*}, Suchithra. R., Research scholar, Jain University, Bangalore-560043, India, Head of Department of MSc IT, Jain University, Bangalore-560043, India

2. [Quantum Security Analysis of AES](#)

Xavier Bonnetain, María Naya-Plasencia, André Schrottenloher. Quantum Security Analysis of AES. IACR Transactions on Symmetric Cryptology, 2019, 2019 (2), pp.55-93. ff10.13154/tosc.v2019.i2.55- 93ff. fffal-02397049

3. [Research on the Security of Elliptic Curve Cryptography](#)

Jiaxu Bao, Queen Mary University of London, London, UK, E1 4NS

4. Introduction to Cryptography and Network Security- McGraw-Hill Forouzan Networking Series

5. <https://github.com/boppreh/aes>

6. <https://www.youtube.com/watch?v=0NGPhAPKYv4>