Technische Hochschule Ingolstadt

Fakultät Informatik

Cryptology SS25

Advanced Encryption Standard

Jiahui Dai
Student-ID: 00153014
B.Sc. Computer Science and Artificial Intelligence

Yana Halamakh
Student-ID: 00151596
B.Sc. Computer Science and Artificial Intelligence

Zeynep Melisa Akyol
Student-ID: 00138182
B.Sc. Computer Science and Artificial Intelligence

Supervisor: Prof. Katherine Roegner
Date: 15 May 2025

Declaration

I hereby declare that we have written the report independently, have not yet submitted it elsewhere, have not used any sources or aids other than those indicated, and have marked direct and indirect quotations as such.

Ingolstadt, 15 May 2025

[illegible]

Signature: Jiahui Dai

I hereby declare that we have written the report independently, have not yet submitted it elsewhere, have not used any sources or aids other than those indicated, and have marked direct and indirect quotations as such.

Ingolstadt, 15 May 2025

[illegible]

Signature: Yana Halamakh

I hereby declare that we have written the report independently, have not yet submitted it elsewhere, have not used any sources or aids other than those indicated, and have marked direct and indirect quotations as such.

Ingolstadt, 15 May 2025

[illegible]

Signature: Zeynep Melisa Akyol

Acronyms

AES Advanced Encryption Standard. 1–3, 6, 8–12, 15, 19, 22

ATM automated teller machine. 1

CBC Cipher Block Chaining. 18, 19

CCM Counter with CBC-MAC. 26

CFB Cipher Feedback. 19

CTR Counter. 19

DES Data Encryption Standard. 1, 3

ECB Electronic Codebook. 18, 19

GCM Galois/Counter Mode. 26

IoT Internet of Things. 25

IV Initialisation Vector. 19

MAC Message Authentication Code. 26

NIST National Institute of Standards and Technology. 1, 9, 22

OFB Output Feedback. 19

PQC Post-Quantum Cryptography. 1

RFID Radio-Frequency Identification. 25

VPN virtual private network. 24

XTS XEX-based Tweaked CodeBook. 19

Abstract

This report explores the Advanced Encryption Standard (AES) in depth, which is a foundational element in modern symmetric block cipher design and a crucial element in modern cryptographic framework. By recalling AES within its historical background, this paper sheds light on the shortcomings of its predecessor, the Data Encryption Standard (DES), which AES was explicitly created to address. The document proceeds to unpack the inner workings of AES—detailing the mechanisms of both encryption and decryption, the structure of key expansion, and the underlying mathematics based on operations in Galois Fields. Core algorithmic steps are carefully explained, including SubBytes, ShiftRows, MixColumns, and AddRoundKey. In order to support the understanding, through the report, an implementation of a working AES-128 in Python is shown, and thoroughly tested against the official NIST-provided test vectors to ensure its accuracy. Further analysis of the real-world implementation of AES across domains, such as wireless communications, secure cloud storage, and encrypted messaging, highlights its broad adoption. Therefore, a balanced review of AES's strengths, such as its strong security guarantees and computational efficiency, while recognising specific practical issues through key distribution challenges and the lack of native authentication elements, is discussed. In its final analysis, we explore how AES withstands emerging quantum computing threats, concluding that the algorithms remain relevant and reliable for modern and future cybersecurity demands.

Contents

List of Figures

List of Tables

List of Code Listings

1 Introduction

1.1 Background

In today's connected digital world, cryptographic algorithms are implemented in every device and applied to every link to protect information in transmission and in storage. Data security is a challenging issue that covers many areas including computer and communication. There has been a rise of cyber attacks in gaining access to computers or computer networks, to gain information of private and sensitive data to gain information via communication channels or unsecure files [1].

Cryptology is the science of devising methods that allow information to be sent in a secure form in such a way that the only person able to retrieve the information is the recipient, through the use of algorithms. Data is exchanged while communicating from one system to another through the use of networks. It is important to encrypt the message so that unintended recipients (intruders) are unable to read the message as network security is highly based on cryptology [2].

Over the past 50 years, the use of cryptographic tools has expanded dramatically, from limited environments like automated teller machines (ATMs) encryption to every digital application used today. National Institute of Standards and Technology (NIST) has played a unique leading role in developing critical standards [3]:

- **Data Encryption Standard (DES)**: Developed in 1973 to protect computer data to allow for large-scale interoperability. DES uses a 64-bit block cipher with 56-bit key.
- **Advanced Encryption Standard (AES)**: Developed in 1997 to superceeded DES, with the use of 128-bit block cipher with three key length options: 128, 192, and 256 bits.
- **Public-Key Cryptology**: Invented in 1976 to allow different parties to establish keys without a protected channel and enabling the function of digital signatures.
- **Post-Quantum Cryptography (PQC)**: Begun development in 2016 to develop quantum-resistant cryptography standards (i.e., PQC) to provide security protection against quantum computers. As of writing this paper, NIST has released the first three finalised Post-Quantum Encryption Standards [4].

1.2 Purpose of the paper

The purpose of this paper is to provide a comprehensive overview of AES, one of the most widely adopted encryption algorithms used to secure digital data. This paper aims to answer the following key questions:

- What is the purpose of AES? What security issue does it address? (Section 2)
- How does AES work? (Section 3)
- Why does it work? What are the underlying mathematics surrounding the procedure? (Section 4)
- An example to demonstrate our understanding of AES. (Section 5)

Furthermore, the paper discusses the real-world applications of AES and its limitations (Section 6).

2 Advanced Encryption Standard (AES)

2.1 What is AES?

Advanced Encryption Standard (AES) is the successor to DES, with the algorithm developed by Vincent Rijmen and Joan Daeman (herein defined as Rijndael algorithm). It is a symmetric block cipher that can process data blocks of 128 bits, using cipher keys with lengths of 128, 192, and 256 bits [5]. As the algorithm utilises different key lengths, they may be referred to as “AES-128”, “AES-192” and “AES-256” based on their key length, with “AES-128” being adopted as the standard. Collectively, they will be referred as “the AES algorithm”.

2.2 The role of AES in data security

AES has been widely used in practice since it was standardised by NIST more than 20 years ago. It is implemented in protocols and products such as TLS, IPsec, IEEE 802.11i, SSH, WhatsApp, Signal, hard disk encryption tools, and many others. AES is the most widely used cipher in the world today and has strongly influenced modern block cipher design. Its design includes the wide trail strategy, which shows how the linear layer helps resist statistical attacks. Many successful modern block ciphers use design elements originally introduced in AES. Currently, no analytical attack that is significantly better than brute force is known on AES.

In addition to its strong theoretical foundations, AES is highly efficient in both software and hardware. It supports high-throughput implementations and benefits from dedicated CPU instructions such as AES-NI (Advanced Encryption Standard New Instructions), accelerating performance in many modern systems. Even advanced cryptanalysis techniques, such as biclique attacks, provide only negligible improvements over brute-force approaches.

Through this unique combination of robust security, practical efficiency, and broad adoption, AES plays a central role in safeguarding digital data across various applications and platforms.

2.3 Security issues addressed

Over the years, several attacks have been proposed against the AES, including square attacks, impossible differential attacks, related-key attacks, and biclique attacks. However, none of these techniques broke the full-round AES under practical conditions.

The biclique attack currently represents the best-known classical cryptanalytic technique on the full AES. AES-128's complexity is approximately 2^{126}, which is only marginally better than exhaustive key search. Similar minor improvements have been reported for AES-192 and AES-256. These theoretical results do not translate into feasible attacks in practice.

Algebraic approaches have also been explored by representing AES operations as polynomial equations. While these representations are mathematically elegant, no practical attack has emerged due to the infeasibility of solving the resulting systems on full-round AES.

In the quantum setting, Grover's algorithm reduces the complexity of brute-force search from 2^n to approximately $2^{n/2}$. However, recent quantum circuit analysis shows that this theoretical speedup does not compromise AES. The most advanced implementations estimate quantum attack complexities as $2^{156.3}$ for AES-128, $2^{221.6}$ for AES-192, and $2^{286.1}$ for AES-256.

These figures, derived from the latest quantum resource evaluations by Jang et al. [6], indicate that AES remains secure despite quantum adversaries, given current and near-future technological capabilities.

All known classical and quantum attacks require unrealistic assumptions or apply only to reduced-round versions of AES. Consequently, AES continues to be considered secure for current and foreseeable post-quantum scenarios.

3 How does AES work?

3.1 Encryption and decryption

Encryption is the transformation of message bits (*plaintext*) into unintelligible form (*ciphertext*) using mathematical formulas (encryption algorithm). Only the intended recipient with the decryption algorithm can decrypt the ciphertext to see the original message [7]. Figure 1 shows the encryption and decryption process.

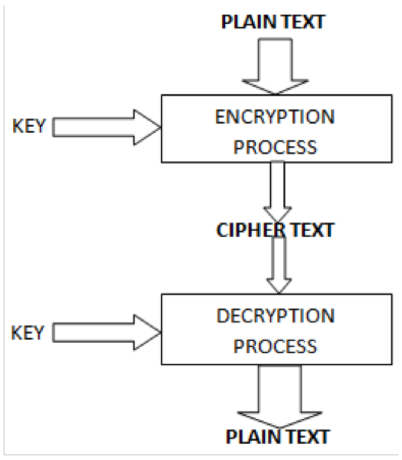


Figure 1: Encryption and decryption process [2].

3.1.1 Symmetric encryption

Symmetric encryption algorithms use a single key that both the sender and recipient have. This key is kept secret among sender and receiver so that no intruder can steal the data to be transferred by encrypting it [2].

3.1.2 Asymmetric encryption

Asymmetric encryption algorithm or public-key systems use two keys: a public key and a private key. The public key is known to everyone while the private key is only used by the recipient of messages [2].

Asymmetric encryption provides more security as compared to symmetric key encryption. However, in case of encryption speed, symmetric encryption is on the lead [2].

3.2 Stream and block ciphers

Stream ciphers encrypt data one bit or one byte at a time. They use a keystream, where each keystream bit is added to the plaintext individually [8]. An example would be the Caesar cipher, which substitutes one character with another individually.

Block ciphers process fixed-size blocks of data, with each block encrypted separately [8].

Stream ciphers are faster and more efficient than block ciphers as they encrypt one bit of data at a time rather than the entire blocks. Block ciphers can offer stronger security over stream ciphers, depending on its use [8].

3.3 AES algorithm workflow

AES is a symmetric block cipher that operates on 128-bit blocks of data. It processes these blocks in several rounds, with each round consisting of multiple layers that manipulate the data in specific ways. These layers introduce both *confusion* and *diffusion* to strengthen the encryption.

3.3.1 Encryption process

The AES structure consists of the following layers [8]:

1. **Key Addition Layer**: A 128-bit round key is `XOR` with the state, with round key generated in Key Expansion (Section 4.6).
2. **Byte Substitution Layer (S-Box)**: Each byte of the state is non-linearly transformed using lookup tables. This introduces confusion, ensuring that small changes in the input lead to significant, non-linear changes in the output.
3. **Diffusion Layer**: This layer spreads the influence of each byte over the entire block. It is divided into two sub-layers:
   (a) **ShiftRow Layer**: The rows of the state are shifted cyclically, which helps spread the data.
   (b) **MixColumn Layer**: This layer performs a matrix multiplication operation on the columns of the state, mixing the data across the block.

Each round, except the initial round, consists of all three layers. The initial round consists of only the Key Addition layer. The final round omits the MixColumn transformation, making both encryption and decryption operations symmetric. Figure 2 shows the block diagram of AES encryption.

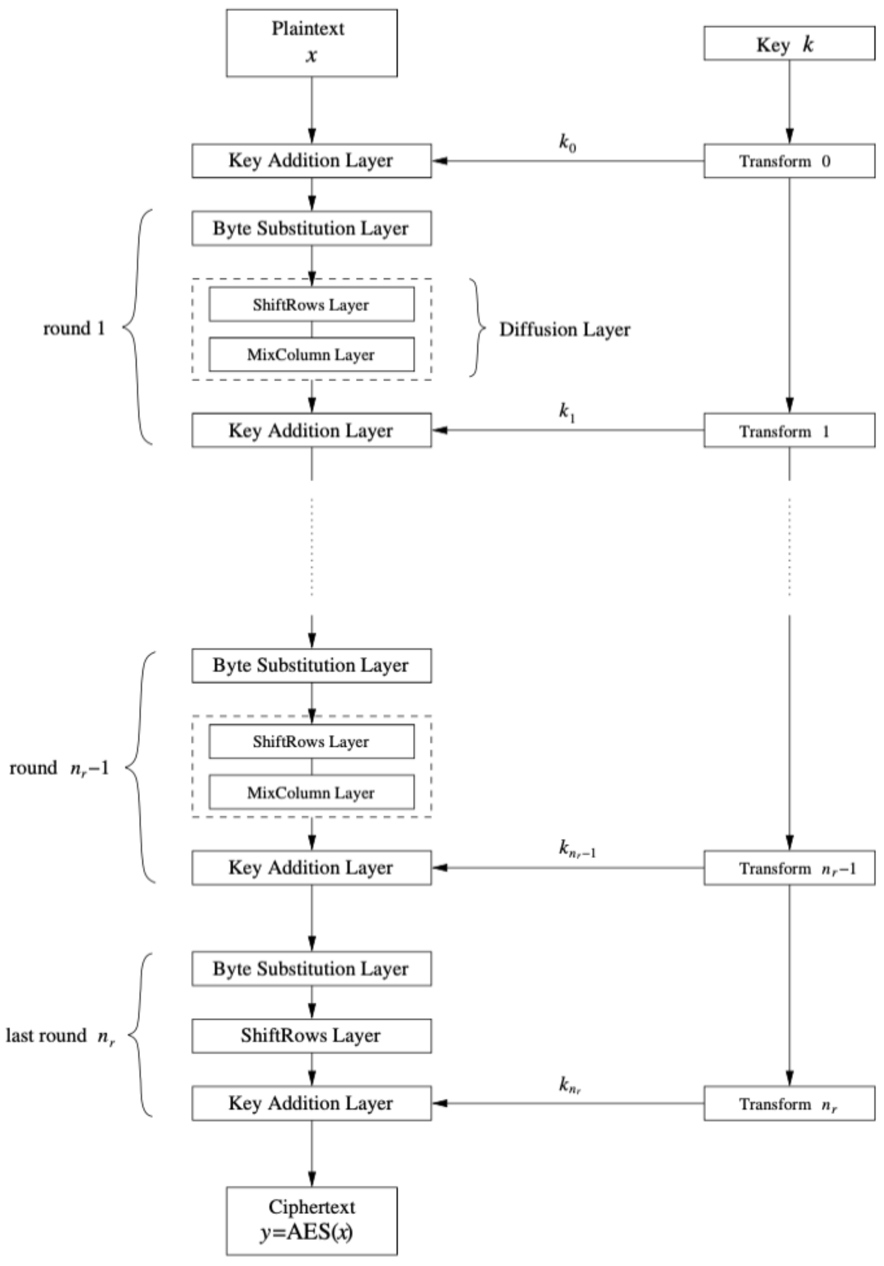


Figure 2: AES Encryption Block Diagram [8]. The plaintext is denoted as x, the ciphertext as y, key as k, and the number of rounds as n_r.

3.3.2 Decryption process

AES decryption is the process of reversing encryption steps to retrieve the original plaintext from a given ciphertext. Since AES is a symmetric block cipher, it uses the same secret key for both encryption and decryption [5].

The input data is handled in 128-bit blocks and processed through multiple transformation rounds. The number of rounds depends on the key size, as outlined in Table 1.

To decrypt, AES performs the inverse of each encryption step, but in reverse order. These inverse operations are:

1. **AddRoundKey**: This step `XOR` the block with the corresponding round key. Since `XOR` is its own inverse, this operation is identical in both encryption and decryption.
2. **InvMixColumns**: Reverses the mixing of bytes in each column. This step is skipped in the final round.
3. **InvShiftRows**: Reverses the row shifts that were applied during encryption.
4. **InvSubBytes**: Applies the inverse S-box substitution to each byte, undoing the non-linear transformation.

Decryption starts by XORing the ciphertext with the final round key. Then, each round applies the inverse transformations using the appropriate round key from the key schedule. After all rounds are completed, the original plaintext is recovered.

3.4 Key sizes and rounds

In AES, the block size is fixed at 128 bits, but the key size can vary between 128, 192, and 256 bits. The key length determines the number of rounds used in the encryption process, as outlined in Table 1.

Key length (bit)	**# rounds (n_r)**
128	10
192	12
256	14

Table 1: Key lengths and number of rounds for AES [8].

4 Why does AES work?

The 16-byte input $A_0, \ldots, A_{15}$ is fed byte-wise into the S-Box in the Byte Substitution layer (Section 4.2). The 16-byte output $B_0, \ldots, B_{15}$ is permutated twice in the ShiftRows (Section 4.3) and mixed by the MixColumn transformation (Section 4.4), both in the Diffusion layer. Finally, the 128-bit subkey k_i is `XOR` with the immediate result in the Key Addition layer (Section 4.5). Figure 3 shows the graph of a single AES round.

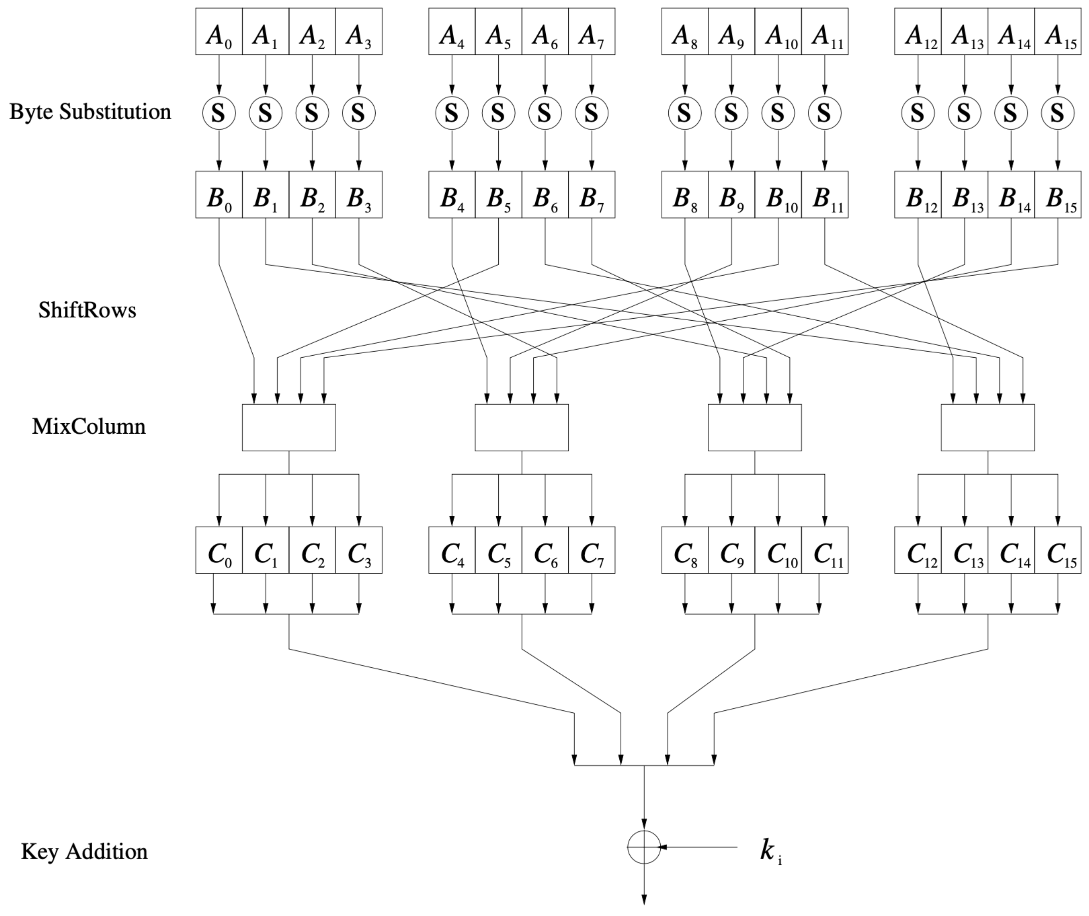

A_0 A_1 A_2 A_3 A_4 A_5 A_6 A_7 A_8 A_9 A_{10} A_{11} A_{12} A_{13} A_{14} A_{15}
Byte Substitution
S
B_0 B_1 B_2 B_3 B_4 B_5 B_6 B_7 B_8 B_9 B_{10} B_{11} B_{12} B_{13} B_{14} B_{15}
ShiftRows
MixColumn
C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9 C_{10} C_{11} C_{12} C_{13} C_{14} C_{15}
Key Addition
k_i

Figure 3: AES round function for rounds $1, 2, \ldots, n_r - 1$ [8].

The mathematical foundations discussed in subsequent subsections are based on the AES specification published by NIST [5] and the textbook by Paar and Pelzl [8].

4.1 Mathematics foundations

As all bytes in the AES algorithm are treated as elements of a finite field, operations such as addition and multiplication are performed according to the rules of the Galois

Field $GF(2^8)$, rather than standard arithmetic. These operations use polynomial representations and are essential to AES transformations.

4.1.1 Galois Field $GF(2^8)$

A Galois Field is a field that contains a finite number of elements, denoted as

$$GF(p^n) \tag{1}$$

where p is a prime number (the characteristic of the field), and $n \in \mathbb{N}$ (the degree of extension).

In the AES algorithm, $GF(2^8)$ is used as its elements are 8-bit binary numbers (ranging from 0 to 255), and it supports both addition and multiplication operations fundamental for the algorithm's transformations. The bytes can be interpreted as polynomial representation:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 = \sum_{i=0}^{7} b_ix^i \text{ with } b_i \in \{0,1\} \tag{2}$$

.

4.1.2 Addition

The addition of two elements in a finite field $GF(2^8)$ is performed with the bitwise `XOR` operation, denoted by $\oplus$. Formally, this can be expressed as

$$c_i = a_i \oplus b_i \quad \forall i \in [0,7] \tag{3}$$

where a_i, b_i, and c_i represents the bits of the input elements and the result respectively.

Example: Addition of $(57)_{\text{hex}}$ and $(83)_{\text{hex}}$

$$\begin{array}{cccccccccc} & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & = & \{57\} \\ \oplus & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & = & \{83\} \\ \hline & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & = & \{d4\} \end{array}$$

Therefore, $(57)_{\text{hex}} \oplus (83)_{\text{hex}} = (d4)_{\text{hex}}$.

4.1.3 Multiplication

Multiplication in a finite field $GF(2^8)$, denoted by $\bullet$, is defined as the multiplication of two polynomials modulo an irreducible polynomial of degree 8. Formally, the operation is expressed as:

$$c(x) = a(x) \bullet b(x) \mod m(x) \tag{4}$$

where $a(x)$ and $b(x)$ are polynomials representing the field elements, and $m(x)$ is the reducible polynomial defining the field.

For AES algorithm, this irreducible polynomial $m(x)$ is fixed and given as follows:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{5}$$

This polynomial corresponds to the binary representation $(100011010)_{\text{bin}}$.

Example: Multiplication of $(53)_{\text{hex}}$ and $(83)_{\text{hex}}$

The polynomial multiplication before modulo reduction is

$$\begin{aligned}\overbrace{(x^6 + x^4 + x^2 + x^1 + 1)}^{(53)_{\text{hex}}} \bullet \overbrace{(x^7 + x + 1)}^{(83)_{\text{hex}}} \mod m(x) \quad = \quad & (x^{13} + x^{11} + x^9 + x^8 + x^7 \\ & + x^7 + x^5 + x^3 + x^2 + x \\ & + x^6 + x^4 + x^2 + x + 1) \mod m(x)\end{aligned}$$

The resulting polynomial sum is computed using the bitwise `XOR` operation over the coefficients:

$$\begin{array}{cccccccccccccc}
 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\oplus & & & & & & & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 \\
\oplus & & & & & & & & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1
\end{array}$$

Next, the modulo operation by the irreducible polynomial $m(x)$ is performed:

$$\begin{array}{ccccccccccccccc}
 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\
\oplus & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & & & & & \\
\hline
 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\
\oplus & & & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & & & \\
\hline
 & & & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}$$

.

The final result corresponds to the binary value $(11000001)_{\text{bin}} = (c1)_{\text{hex}}$.

Thus, $(57)_{\text{hex}} \bullet (83)_{\text{hex}} = (c1)_{\text{hex}}$.

4.2 SubBytes transformation

The SubBytes transformation is a non-linear byte substitution step that operates independently on each byte of the AES state. It can be viewed as the application of 16 parallel S-Boxes, each processing 8-bit input to produce an 8-bit output, as illustrated in Figure 4. For every byte A_i in the AES state, the transformation produces a substituted byte B_i, defined by the function $B_i = S(A_i)$.

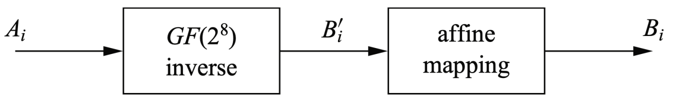


Figure 4: The two operations within the AES S-Box which computes the function $B_i = S(A_i)$ [8].

The construction of the S-Box is based on two sequential transformations:

1. **Galois Field Inversion**: Each byte is interpreted as an element in the finite field $GF(2^8)$ and is mapped to its multiplicative inverse in this field. This operation is defined by:

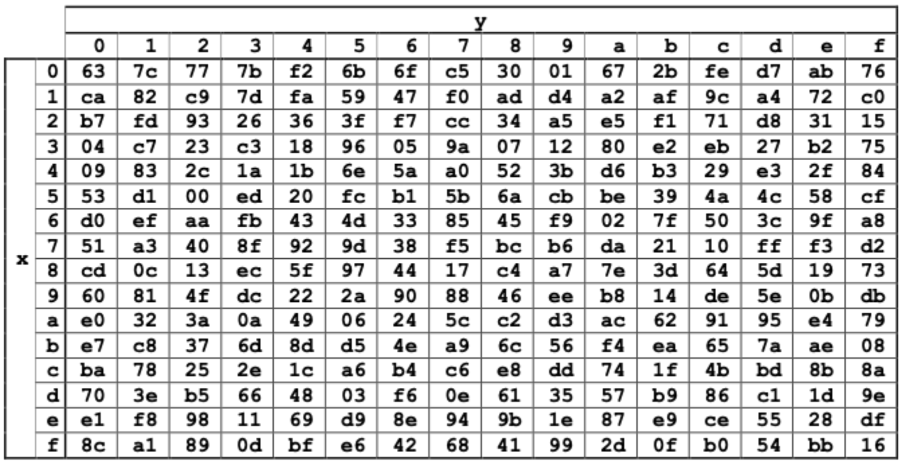
$$A_i \cdot B'_i = 1 \mod m(x) \tag{6}$$

   where $m(x)$ is the irreducible polynomial defining the field, as discussed in Section 4.1.3.
   The element 00, which has no inverse, is mapped to itself.
2. **Affine Mapping**:
   Each resulting byte B'_i undergoes a bitwise affine transformation over $GF(2)$. This transformation is defined as:

$$b_i = {b'}_i \oplus {b'}_{(i+4 \mod 8)} \oplus {b'}_{(i+5 \mod 8)} \oplus {b'}_{(i+6 \mod 8)} \oplus {b'}_{(i+7 \mod 8)} \oplus c_i \tag{7}$$

   for $0 \leq i \leq 8$, where b_i is the i-th output bit of the byte, ${b'}_i$ are the bits of the inverted byte, and c_i are bits from a fixed constant byte.

A complete S-Box can be precomputed by applying these transformations to all 256 possible input bytes (from `00` to `FF`), allowing efficient lookup during AES encryption. The full substitution table is shown in Figure 5.



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

Figure 5: S-Box: substitution values for the byte $\{xy\}$ [5].

Example: SubByte transformation of $(53)_{\text{hex}}$

$$\text{SubByte}((53)_{\text{hex}}) = (ed)_{\text{hex}} \tag{8}$$

4.3 ShiftRows transformation

The ShiftRows transformation cyclincally shifts the second row of the state matrix three bytes to the right, the third row by two bytes to the right, and the fourth row by one byte to the right (Eq. 9). The first row is not changed.

The purpose of this is to increase the diffusion properties of AES.

$$\begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_1 & B_5 & B_9 & B_{13} \\ \hline B_2 & B_6 & B_{10} & B_{14} \\ \hline B_3 & B_7 & B_{11} & B_{15} \\ \hline \end{array} \longrightarrow \begin{array}{|c|c|c|c|} \hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_5 & B_9 & B_{13} & B_1 \\ \hline B_{10} & B_{14} & B_2 & B_6 \\ \hline B_{15} & B_3 & B_7 & B_{11} \\ \hline \end{array} \tag{9}$$

4.4 MixColumns transformation

The MixColumn transformation is a linear transformation which mixes each column of the state matrix. It provides diffusion by mixing the bytes within each column using a fixed matrix over the Galois Field GF(2^8).

The transformation can be expressed as:

$$MixColumn(B) = C \tag{10}$$

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \cdot \begin{pmatrix} B_{0,c} \\ B_{1,c} \\ B_{2,c} \\ B_{3,c} \end{pmatrix} = \begin{pmatrix} C_{0,c} \\ C_{1,c} \\ C_{2,c} \\ C_{3,c} \end{pmatrix}, \quad \text{for } 0 \leq c < N_b \tag{11}$$

with B represents the input state matrix after the ShiftRows transformation, and C is the resulting state after MixColumns transformation. Each column vector of B is multiplied by a constant 4×4 matrix, where each element is a byte in hexadecimal notation and the multiplication is carried out in GF(2^8).

The output bytes in the resulting state matrix C are computed using the following equations, where multiplication and addition are performed in GF(2^8):

$$C_{0,c} = (\{02\} \bullet B_{0,c}) \oplus (\{03\} \bullet B_{1,c}) \oplus B_{2,c} \oplus B_{3,c} \tag{12}$$
$$C_{1,c} = B_{0,c} \oplus (\{02\} \bullet B_{1,c}) \oplus (\{03\} \bullet B_{2,c}) \oplus B_{3,c} \tag{13}$$
$$C_{2,c} = B_{0,c} \oplus B_{1,c} \oplus (\{02\} \bullet B_{2,c}) \oplus (\{03\} \bullet B_{3,c}) \tag{14}$$
$$C_{3,c} = (\{03\} \bullet B_{0,c}) \oplus B_{1,c} \oplus B_{2,c} \oplus (\{02\} \bullet B_{3,c}) \tag{15}$$

with $c \in [1, 4]$ and $\{02\} = (02)_{\text{hex}}$ and $\{03\} = (03)_{\text{hex}}$ correspond to fixed multipliers in the finite field GF(2^8).

4.5 AddRoundKey transformation

The AddRoundKey transformation combines the current state matrix with a round key derived from the key expansion process (Section 4.6).

This transformation operates by performing a bitwise `XOR` operation between each byte of the state matrix and the corresponding byte of the round key, represented as:

$$C_{i,\text{out}} = C_{i,\text{in}} \oplus K_i \tag{16}$$

where $C_{i,\text{in}}$ and $C_{i,\text{out}}$ denote the input and output bytes of the state respectively, and K_i represents the corresponding byte of the round key.

This step introduces key-dependent confusion into the encryption process, ensuring that the state is uniquely modified at each round based on the key material.

4.6 Key Expansion

The key expansion algorithm in AES is responsible for generating the round keys from the original cipher key. These round keys are required at each round of the encryption and decryption processes. For an N_r-round AES cipher, a total of $N_r + 1$ round keys are needed (refer to Table 1).

AES employs a word-oriented key schedule, where one word consists of 32 bits (4 bytes). The generated round keys are stored in a one-dimensional key expansion array W, which holds all the words produced during the expansion process.

In AES-128, the cipher key is 128 bits in length, which corresponds to 4 words. The key expansion produces 11 round keys, amounting to a total of 44 words ($W[0], \ldots, W[43]$) (Figure 6). The initial key forms the first four words of the key expansion array:

$$W[0], W[1], W[2], W[3] = \text{Original AES Key} \tag{17}$$

For each subsequent group of 4 words, the first word is computed using a non-linear transformation function `g()`, and the remaining words are generated through a recursive `XOR` operation:

$$W[4i] = W[4(i-1)] \oplus g(W[4i-1]) \tag{18}$$
$$W[4i+j] = W[4i+j-i] \oplus W[4(i-1)+j] \tag{19}$$

where $i = 1, \ldots, 10$ corresponds to the round index and $j = 1, 2, 3$ correspond to the word index of round key at i.

The function `g()` introduces non-linearity and diffusion into the key expansion process. It operates on a single 4-byte word and comprises three steps:

1. **RotWord**: Performs a cyclic left shift on the input word by one byte.

$$\Big(B_0, B_1, B_2, B_3\Big) \to \Big(B_1, B_2, B_3, B_0\Big) \tag{20}$$

2. **SubWord**: Applies the SubBytes transformation to each byte of the word, substituting them using the AES S-Box (see Section 4.2).

3. **Round Constant (Rcon)**: Adds a round-dependent constant to the most significant byte of the word. Rcon[i] is defined as:

$$\text{Rcon}[i] = (r_i, 0, 0, 0) \tag{21}$$

where $r_i = 2^{i-1}$ is computed in the finite field $GF(2^8)$

The transformation `g()` plays a crucial role in strengthening the cipher by adding confusion and preventing linear relationships between the round keys and the original cipher key.

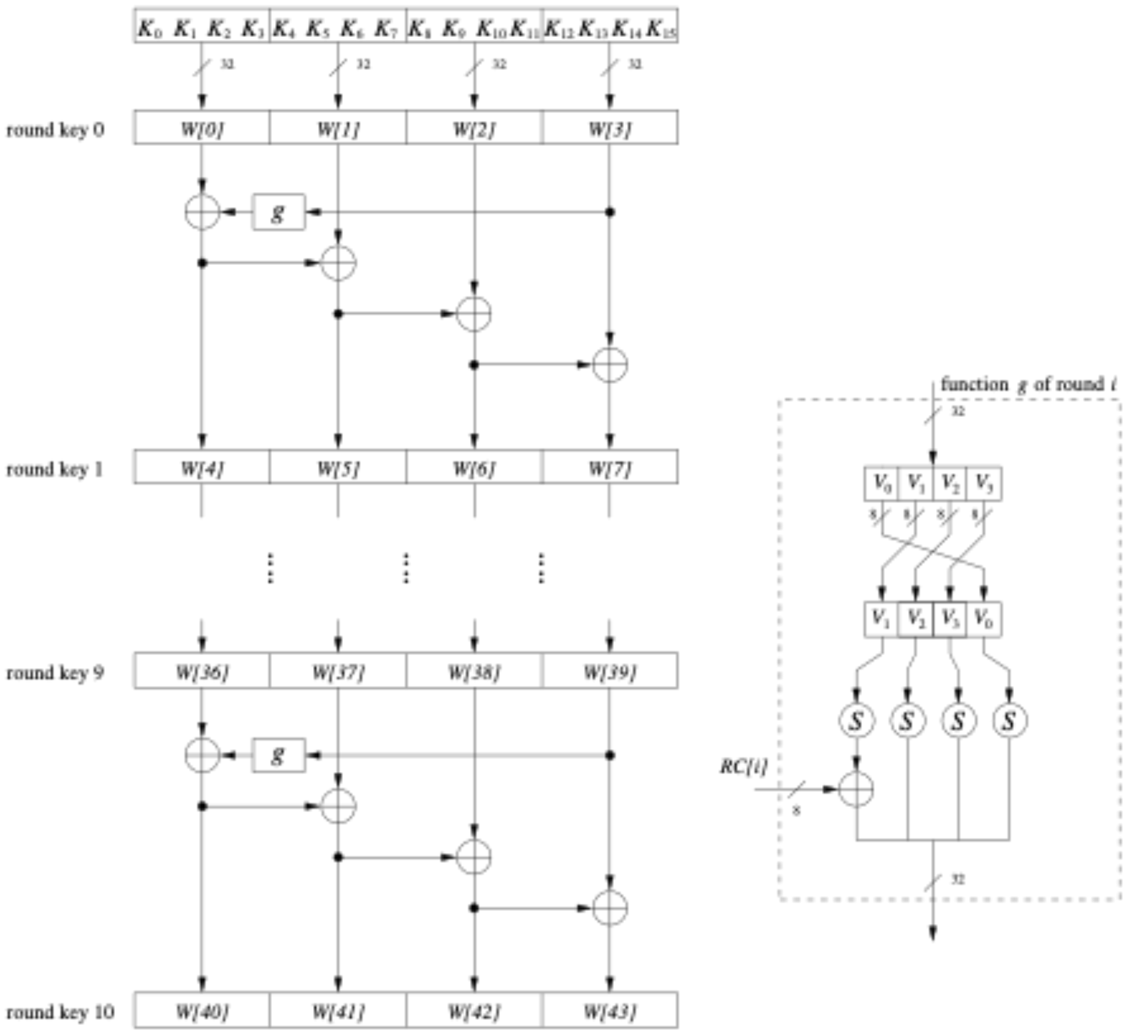


Figure 6: Key schedule for AES-128 [8].

AES-192 and AES-256 differ from AES-128 primarily in terms of their key length and the number of rounds executed during encryption. For AES-192, the key expansion process generates 13 round keys, totaling 52 words (Figure 7). The computation of the key expansion array elements follows a procedure similar to AES-128; however, each iteration produces 6 new words instead of 4. In the case of AES-256, 15 round keys are generated, amounting to 60 words (Figure 8). Each iteration of the key expansion calculates 8 words. Additionally, AES-256 employs an extra function, denoted as `h()`, which performs a SubByte transformation between the third and fourth words of each round iteration, thereby increasing the complexity of the key schedule.

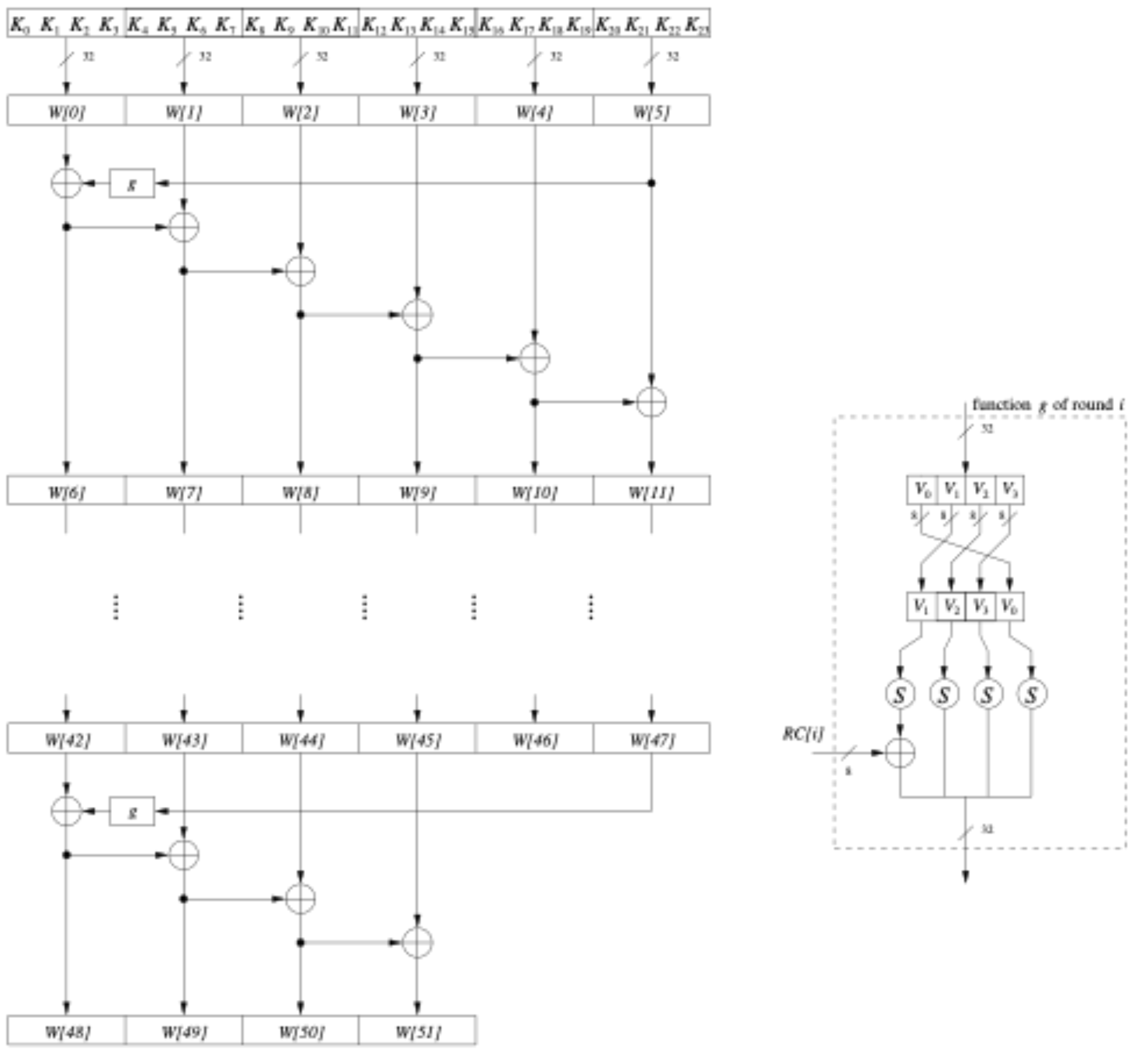

K_0 K_1 K_2 K_3 K_4 K_5 K_6 K_7 K_8 K_9 K_{10} K_{11} K_{12} K_{13} K_{14} K_{15} K_{16} K_{17} K_{18} K_{19} K_{20} K_{21} K_{22} K_{23}
32
W[0] W[1] W[2] W[3] W[4] W[5]
g
W[6] W[7] W[8] W[9] W[10] W[11]
W[42] W[43] W[44] W[45] W[46] W[47]
g
W[48] W[49] W[50] W[51]
function g of round i
32
V_0 V_1 V_2 V_3
8
V_1 V_2 V_3 V_0
S S S S
RC[i]
8
32

Figure 7: Key schedule for AES-192 [8].

4.7 Security properties

Modes of operation define how a block cipher like AES is applied to variable-length data. Each mode introduces distinct security implications, including confidentiality, error propagation, resistance to structural analysis, and resilience against future threats such as quantum computing.

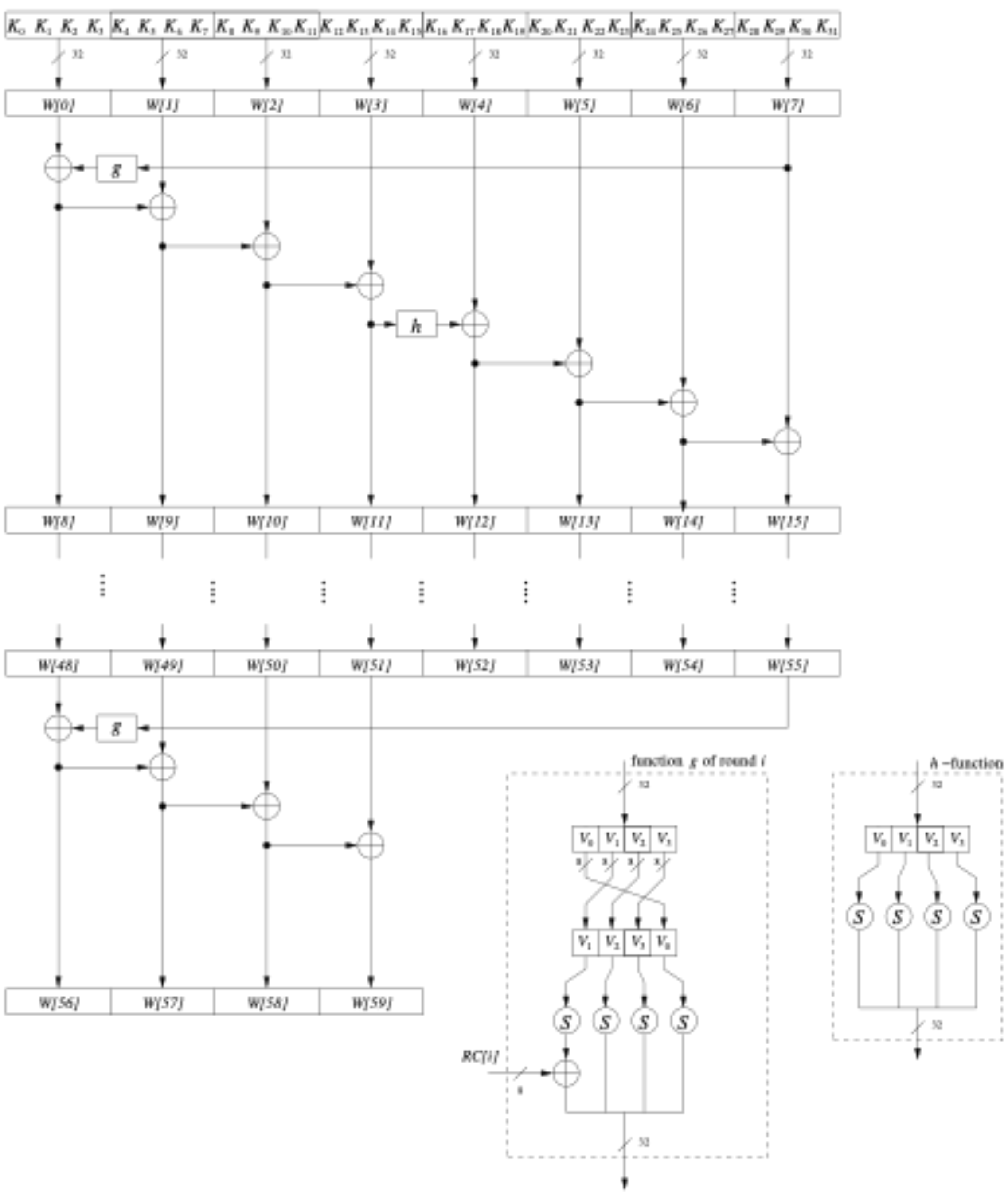


Figure 8: Key schedule for AES-256 [8].

The security of a mode depends not only on AES itself but also on how the blocks are processed. For instance:

- **Electronic Codebook (ECB) Mode** encrypts each block independently, leading to pattern leakage. Identical plaintext blocks produce identical ciphertexts, compromising confidentiality, especially in structured data like images.
- **Cipher Block Chaining (CBC) Mode** introduces randomisation through

an Initialisation Vector (IV), ensuring that identical messages yield different ciphertexts. However, encryption is inherently sequential and more sensitive to bit-flip propagation.

- **Cipher Feedback (CFB) and Output Feedback (OFB) Modes** convert the block cipher into a stream cipher. OFB offers better error isolation, while CFB is self-synchronising but more susceptible to feedback-based error propagation.
- **Counter (CTR) Mode** enables full parallelism and random access by encrypting incrementing counters. It avoids feedback loops and supports high-throughput use cases, making it robust in environments requiring speed and scalability.
- **XEX-based Tweaked CodeBook (XTS)-AES** is optimised explicitly for disk encryption. It uses a tweakable block cipher based on sector position to ensure that the same data encrypted at different locations yields different ciphertexts, offering strong structural and positional integrity.

Table 2 describes the security characteristics of AES operations.

Mode	Parallel	Error Propagation	Random Access	Best Use Case
ECB	Yes	None	Yes	Toy cases, testing only
CBC	No (Enc) / Yes (Dec)	Yes (next block)	No	File encryption
CFB	No	Yes (next block)	No	Streaming data with feedback
OFB	Yes	None	No	Resilient byte-wise streaming
CTR	Yes	None	Yes	High-throughput systems
XTS	Yes	None	Yes	Secure disk encryption

Table 2: Security characteristics of AES modes of operation.

With the anticipated rise of quantum computing, AES modes must also be evaluated under quantum threat models. Grover's algorithm reduces brute-force complexity from 2^k to approximately $2^{k/2}$.

Jang et al. [6] present refined circuit depth and gate count estimates under realistic quantum constraints. Their study indicates the following effective quantum security levels:

- AES-128: $2^{156.26}$
- AES-192: $2^{221.58}$
- AES-256: $2^{286.07}$

These values, derived using optimised Grover-based search circuits, are well above the estimated practical capabilities of near-term quantum computers [6].

Importantly, parallelisable modes like CTR and XTS are more amenable to low-depth quantum circuit implementations. Their compatibility with constraints such as NIST's `MAXDEPTH` parameter makes them promising candidates for quantum-resilient applications [6].

5 Practical demonstration of AES

5.1 Implementation details

Building on the mathematical foundation outlined in Section 4, the AES algorithm was implemented in Python to demonstrate its practical application. This implementation supports only AES-128, and requires both the encryption key and input data to be provided in hexadecimal format. The complete source code is provided in Appendix A.

The implementation is structured across three Python modules: `functions.py`, `bin_math.py`, and `aes.py`, each responsible for a specific aspect of the algorithm.

The `functions.py` module (see Python Code 1) contains general-purpose helper functions used throughout the AES implementation. These functions include:

- `process_str(string)`: Removes all whitespace from a given string.
- `process_key(key, Nk=4)`: Processes a hexadecimal key string into a 4×4 `NumPy` array of bytes.
- `process_input_hex(input_hex)`: Processes a hexadecimal input (e.g., plaintext or ciphertext) into a 4×4 AES state matrix.
- `bytes_to_word(bytes_list)`: Converts a list of 4 bytes into a 32-bit word.
- `word_to_bytes(word)`: Converts a 32-bit word into a list of 4 bytes.
- `state_to_hex(state)`: Converts a 4×4 AES state matrix into a hexadecimal string representation.

The `bin_math.py` module (see Python Code 2) implements the arithmetic operations required for AES's finite field calculations. This module includes:

- `gf_mult(a, b)`: Performs Galois Field multiplication of two bytes, as explained in Section 4.1.3.
- `generate_gf(primitive_element)`: Constructs the extension field $GF(2^8)$ using a given primitive element, as described in Section 4.1.3.

The core AES encryption logic is implemented in the `aes.py` module (see Python Code 3). This module includes the main algorithmic transformations and key scheduling logic, as described below:

- `load_properties(key_length)`: Loads AES parameters such as the number of rounds based on the specified key length.

- `generate_sbox()`: Generates the AES S-Box used in the SubBytes step (see Section 4.2).
- `generate_rc(Nr=10)`: Produces a list of round constants used in the key expansion process (see Section 4.6).
- `g(word, rc)`: Implements the transformation function used during key expansion (see Section 4.6).
- `KeyExpansion(key, Nb=4, Nk=4, Nr=10)`: Performs AES key expansion to generate round keys (see Section 4.6).
- `enter_key(key)`: Prepares and expands the encryption key by processing it and initializing the key expansion.
- `SubBytes(state)`: Performs the SubBytes transformation to the AES state (see Section 4.2).
- `ShiftRows(state)`: Performs the ShiftRows transformation (see Section 4.3).
- `MixColumns(state)`: Performs the MixColumns transformation (see Section 4.4).
- `AddRoundKey(state, keys)`: Adds the round key to the current state using `XOR` (see Section 4.5).
- `encrypt(input_)`: Executes the full AES encryption procedure on the given input.

5.2 Results and analysis

The AES implementation was tested using the script aes.py with a predefined key and plaintext input. The parameters used for this test are detailed below:

```
key:        000102030405060708090a0b0c0d0e0f
plaintext:  00112233445566778899aabbccddeeff
```

Upon execution, the resulting ciphertext output was:

```
ciphertext:  69C4E0D86A7B0430D8CDB78070B4C55A
```

This output matches the expected result defined in the official AES specification published by the NIST [5]. The consistency of this result validates the correctness of the AES encryption implementation in `aes.py`.

5.3 Possible future work

Future work could focus on extending the current implementation to include AES-192 and AES-256, which would involve handling their respective key lengths and more complex key expansion procedures. Additionally, implementing the decryption process would provide a complete encryption-decryption cycle, enabling full functionality of the AES algorithm. Another potential enhancement is to allow input keys and plaintext data in formats other than hexadecimal, such as ASCII or binary, by incorporating automatic conversion methods. Finally, a comparative analysis of runtime complexity and execution time for encryption and decryption across AES-128, AES-192, and AES-256 could provide valuable insights into the performance and efficiency trade-offs among these variants.

6 Applications and Limitations

6.1 Common use case

Wireless Security (Wi-Fi): AES is commonly used together with Wi-Fi security protocols (such as WPA2 and WPA3) to encrypt data transmitted over wireless networks. This is to ensure that sensitive information, like passwords and personal data, remains protected from unauthorised access [9].

Encrypted Browsing (HTTPS): Websites use AES encryption within HTTPS protocols to ensure data transmitted between browsers and servers. This encryption helps protect user information, such as login credentials and payment details, from interception by malicious actors [9].

Virtual Private Networks: The job of a virtual private network (VPN) is to securely connect a user to another server online, only the best encryption can be considered so that the user's data will not be leaked. The VPNs that use AES with 256-bit keys include NordVPN, Surfshark, and ExpressVPN [10].

File and Disk Encryption: Operating systems like Windows and macOS offer AES-based encryption options (e.g. BitLocker and FileVault) for securing entire hard drives or individual files. This is particularly useful for safeguarding personal or sensitive business information stored on physical devices [9].

Cloud Storage: AES encryption is essential for securing files stored in cloud environments. Services like Google Drive, Dropbox, and others use AES to ensure that uploaded files remain confidential and protect against unauthorised access [9].

Mobile Applications: Many mobile apps, especially those dealing with financial transactions or personal data, use AES encryption to secure data on devices and in transit. This includes banking apps, social media platforms, and messaging apps, providing users with a peace of mind that their data is protected [9].

Secure Messaging: Many encrypted messaging applications, like Signal and WhatsApp, use AES to secure messages end-to-end, ensuring that only the sender and recipient can read contents of their conversations [9].

Password Managers: These are the programs that carry a lot of sensitive information. Hence, password managers like LastPass and Dashlane include the important step of AES implementation [10].

6.2 Strengths

Cryptanalytic Strength and Security

AES's greatest strength lies in its cryptographic security. As discussed in Section 2.3, its three key lengths make it highly resistant to brute-force attacks with existing and foreseeable computational resources. Even with theoretical advances such as quantum computing, AES-256 is believed to remain secure. Moreover, after more than two decades of intense public and academic scrutiny, no feasible attack has been discovered that compromises the full version of AES more efficiently than exhaustive key search.

Performance and Resource Efficiency

AES excels in performance across software and hardware implementations. Its design allows efficient realisation in embedded devices, consumer hardware, and high-speed servers. In software, techniques such as T-Box-based lookup tables and processor instruction sets like Intel's AES-NI enable quick block cipher operations, with throughput often measured in hundreds of megabits or gigabits per second. In hardware, AES is compact enough for low-power chips (e.g., in smartcards, Radio-Frequency Identification (RFID), or Internet of Things (IoT) devices) but also scales up for multi-gigabit networking and storage applications via pipelining and parallelisation.

Global Standardization

AES has earned a reputation for deep trust and widespread acceptance, thanks to its transparent and internationally recognized selection process. It is officially acknowledged in major standards and has been certified by the U.S. NSA for the protection of classified information up to the TOP SECRET level, specifically with 192 and 256-bit keys. This extensive regulatory endorsement guarantees that AES solutions remain compatible and consistent across various fields and organizations, from finance and business to government and healthcare [9].

6.3 Limitations and considerations:

Despite its many strengths, using AES in practice has several important challenges that can affect security and effectiveness of cryptographic solutions.

Key Management

No symmetric cipher, including AES, can guarantee security if key management practices are poor. The entire method fundamentally depends on the secrecy, integrity, and lifecycle management of keys. Secure generation, distribution, rotation, storage (often using physical security modules), and destruction are critical.

Despite AES's mathematical strength, mistakes such as using weak keys, allowing key reuse, failing to rotate compromised keys, or transmitting keys over insecure channels can undermine the confidentiality of encrypted data.

Implementation Vulnerabilities

AES's theoretical resilience is not immune to practical attacks targeting its real-world implementation. Side-channel attacks, including timing analysis, cache behaviour exploitation, power consumption monitoring, and electromagnetic radiation assessment, pose significant threats to the confidentiality of cryptographic key material. If the implementation is not carefully handled, these attacks can effectively disclose sensitive information. Mitigating these risks requires additional programming practices like constant-time implementations or mathematically secure curves.

Lack of Built-In Authentication

AES offers confidentiality, but it does not include a way to verify that the encrypted data hasn't been tampered with. Data integrity and authenticity must be added using Message Authentication Codes (MACs) or by adopting combined authenticated encryption modes like Galois/Counter Mode (GCM) or Counter with CBC-MAC (CCM) [11]. Neglecting to supplement basic AES with integrity checks can expose the system to forgery and modification attacks.

7 Conclusion

The Advanced Encryption Standard proved to be the backbone of 21st-century cryptographic security. It demonstrates remarkable resilience and adaptability across diverse applications.

AES operates as a symmetric block cipher, processing data in 128-bit blocks using key lengths of 128, 192, or 256 bits. This flexibility in key sizes, combined with its mathematically strong design, provides scalable security levels suitable for efficient execution in embedded devices, consumer hardware, and high-speed servers. The algorithm's strength lies in its systematic round-based structure, where each round applies four transformations: SubBytes, ShiftRows, MixColumns, and AddRoundKey, creating a robust encryption framework.

Our analysis of AES demonstrates its versatility and effectiveness in applications ranging from wireless security and virtual private networks to cloud storage and secure messaging.

Despite its strengths, successful implementation requires careful attention to potential vulnerabilities. Key management remains crucial, as poor key handling practices can compromise even the strongest encryption. Additionally, while AES provides excellent confidentiality, it should be complemented with appropriate authentication mechanisms for complete security.

AES continues demonstrating resilience against known and emerging threats, including potential quantum computing challenges. Its mathematical foundation and successful stability suggest it will remain a fundamental component of cybersecurity infrastructure for years ahead. As digital security needs evolve, AES's robust design and adaptability position it to continue serving as a cornerstone in protecting sensitive information across the digital world.

This comprehensive research on AES has highlighted why it remains the global standard for symmetric encryption. It balances security, performance, and practical implementation needs in an increasingly connected world.

A Appendix: Python Code

The following Python code files implement the components described in Section 5.

functions.py

```python
import numpy as np

def process_str(string):
    """
    Removes all whitespace from a given string.
    """

    return string.replace(" ", "")

def process_key(key, Nk=4):
    """
    Processes a hexadecimal key string into a NumPy array of bytes.
    """

    try:
        key = process_str(key)  # Remove spaces

        # Convert hex string to list of integers
        bytes_list = [int(key[i:i+2], 16) for i in range(0, len(key), 2)]

        # Reshape into a (Nk, 4) NumPy array
        return np.array(bytes_list).reshape((Nk, 4))
    except:
        raise Exception("Key must be hexadecimal.")

def process_input_hex(input_hex):
    """
    Processes a hexadecimal input (e.g., plaintext or ciphertext) into a 4x4 AES state matrix.
    """
    try:
        return process_key(input_hex).T
    except:
        raise Exception("Input must be hexadecimal.")


def bytes_to_word(bytes_list):
    """
    Converts a list of 4 bytes into a 32-bit word.
    """
    if len(bytes_list) != 4:
        raise Exception("Use case only valid for AES-128 (4-byte word)")

    return (bytes_list[0] << 24) | (bytes_list[1] << 16) | (bytes_list[2] << 8) | bytes_list[3]

def word_to_bytes(word):
    """
    Converts a 32-bit word into a list of 4 bytes.
    """
    return [
        (word >> 24) & 0xFF,     # Most significant byte
        (word >> 16) & 0xFF,
        (word >> 8) & 0xFF,
        word & 0xFF              # Least significant byte
    ]

def state_to_hex(state):
    """
    Convert AES state matrix (4x4 bytes) to a hex string
    """
    state = np.array(state)
    flat = state.T.flatten()
    return "".join(f"{byte:02X}" for byte in flat)
```

Python Code 1: Helper functions for AES operations bin_math.py

```python
from aes import aes_properties

IRRED_POLY = aes_properties["irred_poly"]

def gf_mult(a, b):
    """
    Galois Field (GF) multiplication
    """
    result = 0
    for i in range(8):
        if b & 1:
            result ^= a
        a <<= 1
        if a & 0x100:
            a ^= IRRED_POLY
        b >>= 1
    return result & 0xFF  # Ensure the result is within 0x00
      ↪ to 0xFF


def generate_gf(primitive_element):
    """
    Extension field of 2^8, multiplicative of
      ↪ primitive_element
    """
    field = [0]  # Start with 0
    element = primitive_element

    for _ in range(1, 256):
        field.append(element)
        element = gf_mult(element, primitive_element)

    return field
```

Python Code 2: Bitwise operations for finite field arithmetic aes.py

```
import numpy as np
from BitVector import *
from functions import *
from bin_math import *

aes_properties = {
    "Nr": {          # Number of internal rounds
        128: 10,
        192: 12,
        256: 14
    },
    "irred_poly": 0b100011011, # irredicible polynomial
}


class AES():

    def __init__(self, key_length=128):
        self.load_properties(key_length)
        self.generate_sbox()
        self.gfp2 = generate_gf(2)
        self.gfp3 = generate_gf(3)
        self.generate_rc(self.Nr)


    def load_properties(self, key_length):
        """
        Load properties of AES related to its key length
        """
        if key_length == 128:
            self.irred_poly = aes_properties["irred_poly"]  #
              ↪  Irreducible polynomial of AES
            self.Nr = aes_properties["Nr"][key_length]      #
              ↪  Number of internal rounds
            self.Nk = key_length // 32                      #
              ↪  Number of 32-bit words in key, e.g. [0x--,
              ↪ 0x--, 0x--, x---] for AES-128
            self.Nb = 4                                     #
```

```python
                    Number of columns in state matrix

        elif key_length in [192, 256]:
            raise Exception(f"AES-{key_length} is not implemented in the code, except for AES-128.")

        else:
            raise Exception("Please enter a valid key length: 128, 192, 256.")

    def generate_sbox(self):
        """
        Generate S-Box
        """
        irred_poly = BitVector(bitstring='100011011') # irreducible polynomial
        subBytesTable = []

        c = BitVector(bitstring='01100011')

        for i in range(0, 256):
            # Initialise bit
            b = BitVector(intVal = i, size=8)

            # GF(2) Inverse
            b = b.gf_MI(irred_poly, 8) if i != 0 else BitVector(intVal=0)

            # Affine mapping
            b1,b2,b3,b4 = [b.deep_copy() for x in range(4)]
            b ^= (b1 >> 4) ^ (b2 >> 5) ^ (b3 >> 6) ^ (b4 >> 7) ^ c

            subBytesTable.append(int(b))

        self.sbox = subBytesTable

    def generate_rc(self, Nr=10):
        """
        Generate list of round coefficient (RC) for use in KeyExpansion
        """
        rc = [0x01]
        for _ in range(1, Nr):
            rc.append(gf_mult(rc[-1], 0x02))

        self.rc = rc

    def g(self, word, rc):
        """
        g() function in KeyExpansion
        """
        bytes = word_to_bytes(word)      # Convert word to list of 4 bytes

        # 1. RotWord: left-rotate by 1 byte
        bytes = bytes[1:] + bytes[:1]

        # 2. SubWord: substitute using S-box
        bytes = [self.sbox[byte] for byte in bytes]

        # 3. # XOR first byte with round constant
        bytes[0] ^= rc

        return bytes_to_word(bytes)      # Convert back to a 32-bit word

    def KeyExpansion(self, key, Nb=4, Nk=4, Nr=10):
        """
        Key Expansion
        """
        # Initialise first round of keys
        w = [0] * (Nb + Nk * Nr)
        for i in range(Nb):
            w[i] = bytes_to_word(key[i])

        # Calculate subsequent round of keys
```

```python
        for i in range(1, Nr+1):
            for j in range(4):
                if j == 0:
                    w[4*i] = w[4*(i-1)] ^ self.g(w[4*i-1], rc=self.rc[i-1])
                else:
                    w[4*i + j] = w[4*i + j - 1] ^ w[4*(i-1) + j]

        return w

    def enter_key(self, key):
        """
        Prepares and expands the encryption key for AES.
        """
        self.key = process_str(key)

        key = process_key(key, self.Nk)
        self.W = self.KeyExpansion(key, self.Nb, self.Nk, self.Nr)

    def SubBytes(self, state):
        """
        SubBytes transformation
        """
        return [[self.sbox[byte] for byte in word] for word in state]

    def ShiftRows(self, state):
        """
        ShiftRows transformation
        """
        n = [word[:] for word in state] # temp state

        for i in range(self.Nb):
            for j in range(4): # shift rows by i
                n[i][j] = state[i][(i + j) % self.Nb]

        return n

    def MixColumns(self, state):
        """
        MixColumns transformation
        """
        n = [[0] * self.Nb for _ in state]

        for c in range(self.Nb):
            s0 = state[0][c]
            s1 = state[1][c]
            s2 = state[2][c]
            s3 = state[3][c]

            n[0][c] = gf_mult(2, s0) ^ gf_mult(3, s1) ^ s2 ^
               ↪ s3
            n[1][c] = s0 ^ gf_mult(2, s1) ^ gf_mult(3, s2) ^
               ↪ s3
            n[2][c] = s0 ^ s1 ^ gf_mult(2, s2) ^ gf_mult(3,
               ↪ s3)
            n[3][c] = gf_mult(3, s0) ^ s1 ^ s2 ^ gf_mult(2,
               ↪ s3)

        return n

    def AddRoundKey(self, state, keys):
        """
        AddRoundKey transformation
        """
        s_ = [[None for _ in range(4)] for i in range(self.Nb
           ↪ )]

        k_ = [word_to_bytes(word) for word in keys]

        for c in range(self.Nb):
            for i in range(4):
                s_[i][c] = state[i][c] ^ k_[c][i]

        return s_

    def encrypt(self, input_):
        """
        Encryption algorithm of AES
        """
        self.input_ = input_
        initial_state = process_input_hex(input_)

        # Initial round (Round 0)
        state = self.AddRoundKey(initial_state, self.W[:4])

        # Round 1 to Nr-1
        for r in range(1, self.Nr):
            state = self.SubBytes(state)
            state = self.ShiftRows(state)
            state = self.MixColumns(state)
            state = self.AddRoundKey(state, self.W[4*r : 4*r
                ↪ + 4])

        # Final round (Round Nr)
        state = self.SubBytes(state)
        state = self.ShiftRows(state)
        state = self.AddRoundKey(state, self.W[-4:])

        return state


if __name__ == "__main__":
    key = '000102030405060708090a0b0c0d0e0f'
    input_ = '00112233445566778899aabbccddeeff'

    aes = AES()
    aes.enter_key(key)
    final = aes.encrypt(input_)

    print(f"key:\t\t{key}")
    print(f"plaintext:\t{input_}")
    print(f"ciphertext:\t{state_to_hex(final)}")
```

Python Code 3: Implementation of the AES algorithm

References

[1] Center for Strategic and International Studies. *Significant Cyber Incidents.* Accessed: 2025-05-12. 2025. URL: `https://www.csis.org/programs/strategic-technologies-program/significant-cyber-incidents`.

[2] Rajdeep Bhanot and Rahul Hans. "A Review and Comparative Analysis of Various Encryption Algorithms". In: *International Journal of Security and Its Applications* 9 (Apr. 2015), pp. 289–306. DOI: `10.14257/ijsia.2015.9.4.27`.

[3] Lily Chen and Matthew Scholl. *The Cornerstone of Cybersecurity - Cryptographic Standards and a 50-Year Evolution.* Accessed: 2025-05-12. May 2022. URL: `https://www.nccoe.nist.gov/news-insights/cornerstone-cybersecurity-cryptographic-standards-and-50-year-evolution`.

[4] National Institute of Standards and Technology. *NIST Releases First 3 Finalized Post-Quantum Encryption Standards.* Accessed: 2025-05-12. Aug. 2024. URL: `https://www.nist.gov/news-events/news/2024/08/nist-releases-first-3-finalized-post-quantum-encryption-standards`.

[5] National Institute of Standards et al. *Advanced Encryption Standard (AES).* en. Nov. 2001. DOI: `https://doi.org/10.6028/NIST.FIPS.197`. URL: `https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=901427`.

[6] Kyungbae Jang et al. "Quantum Analysis of AES". In: *IACR Communications in Cryptology* 2.1 (Apr. 8, 2025). ISSN: 3006-5496. DOI: `10.62056/ay11zo-3y`.

[7] SistlaVasundhara Devi and Harika Kotha. "AES encryption and decryption standards". In: *Journal of Physics: Conference Series* 1228 (May 2019), p. 012006. DOI: `10.1088/1742-6596/1228/1/012006`.

[8] Christof Paar and Jan Pelzl. *Understanding Cryptography: A Textbook for Students and Practitioners.* Heidelberg, Dordrecht, London, New York: Springer-Verlag Berlin Heidelberg, 2010, pp. 87–121. ISBN: 978-3-642-04100-6. DOI: `10.1007/978-3-642-04101-3`.

[9] Verena Cooper. *AES Encryption Explained: How It Works, Benefits, and Real-World Uses.* Accessed: 2025-05-12. Apr. 2025. URL: `https://www.splashtop.com/blog/aes-encryption`.

[10] Rūta Rimkienė. *What is AES Encryption and How Does It Work?* Accessed: 2025-05-12. Aug. 2022. URL: `https://cybernews.com/resources/what-is-aes-encryption/`.

[11] Radha Poovendran, Junhyuk Song, and Jicheol Lee. *The AES-CMAC-96 Algorithm and Its Use with IPsec*. RFC 4494. June 2006. DOI: `10.17487/RFC4494`. URL: `https://www.rfc-editor.org/info/rfc4494`.