Faculty of Natural and Mathematical Sciences
Department of Information

King's College London
Strand Campus, London,
United Kingdom

7CCSMPRJ

Individual Project Submission 2025

Name:	Angelica Thiline Weerasinghe
Student Number:	K24081313
Degree Programme:	MSc Cyber Security
Project Title:	Follow the standard! Attacks on poorly designed cryptography and how to fix them.
Supervisor:	Dr Eamonn Postlethwaite
Word Count:	Word count goes here

RELEASE OF PROJECT

Following the submission of your project, the Department would like to make it publicly available via the library electronic resources. You will retain copyright of the project.

Check the appropriate box below

☐ I **agree** to the release of my project

☐ I **do not** agree to the release of my project

Signature: Signature **Date:** July 28, 2025

Department of Information
King's College London
United Kingdom

7CCSMPRJ Individual Project

Follow the standard! Attacks on poorly designed cryptography and how to fix them.

Name: **Angelica Thiline Weerasinghe**
Student Number: K24081313
Course: MSc Cyber Security

Supervisor: Dr Eamonn Postlethwaite

This dissertation is submitted for the degree of MSc in MSc Cyber Security.

1 Template Descriptions

1.1 Template Folder Structure

After unzip the Latex template, you will find the following folders:

- **"root" folder:** The file "thesis.tex" is the master file of the whole document, which defines the structure of the chapters and other settings.
- **"contents" folder:** This is the folder for all chapters. Use the path

  `contents/chapter_filename`

  when including chapters. The following files can be found in this folder.

  - acknowledgements.tex: the contents of the Acknowledgement chapter.
  - abstract.tex: the contents of the Abstract chapter.
  - nomenclature.tex: the contents of the Nomenclature chapter.
  - introduction.tex: the contents of the Introduction chapter.
  - background.tex: the contents of the Background Theories chapter.
  - main.tex: the contents of the chapter regarding the main results.
  - conclusion.tex: the contents of the Conclusion chapter.
  - app_1.tex: the contents of the Appendix chapter.
  - sample_1.bib: the bibtex file for references. Remember to run "bibtex" to update the list of reference section.
  - TemplateDescriptions.tex: this is the content of this chapter (Template Description). Remove the line

    ```
    \include{contents/TemplateDescriptions}
    ```

    from "thesis.tex" which will remove the Chapter "Template Descriptions" from the document.

  Create a new chapter file if necessary.

- **"figures" folder:** This is the folder for all figures. Use the path

  `contents/figure_filename`

  when including figures.

contents/sample1 bibtex, run bibtex

1.2 Student and Project Information

The master file of this template is "thesis.tex". Change the following lines accordingly in "thesis.tex" to include your information.

```
%=============================================================
% Change below accordingly and remove``\red{}''
%=============================================================
\modulecode{\red{7CCSMPRJ/7CCSMUIP}} %Use the right module code
\submissiontitle{Individual Project Submission \red{20XX/XX (Replace XX by year)}}
\studentnumber{\red{Student number goes here}}
\programme{\red{Programme title goes here}}
\supervisor{\red{Supervisor's name goes here}}
\wordcount{\red{Word count goes here}}
\title{\red{Project title goes here}}
\author{\red{Your name goes here}}
\ReleaseProject{0} %Replace 0 by 1 if release project; Replace 0 by 2 if not release proj
\department{\red{Engineering/Information}} %use the right department
%=============================================================
```

Remove the line

```
\include{contents/TemplateDescriptions}
```

from "thesis.tex" which will remove the Chapter "Template Descriptions" from the document.

Replace the image "signature.png" in the folder "figures" by your signature image in "png" format.

The content of “Acknowledgements” is in “\contents\acknowledgements.tex”

Acknowledgements

It is a short paragraph to thank those whose have contributed to the project work.

The content of “Abstract” is in “\contents\abstract.tex”

Abstract

It is a precis of the report (normally in one page), which should include:

- A brief introduction to the project objectives
- A brief description of the main work of the project
- A brief description of the contributions, major findings, results achieved and principal conclusion of the project

Preliminaries.

AES	Advanced Encryption Standard
CMAC	Cipher-based Message Authentication Code
Plaintext	Original unencrypted data
Ciphertext	Encrypted version of the plaintext
State	Intermediate 4x4 byte matrix used in AES
Round	One iteration of AES transformations
in	Input data block for AES encryption
Nr	Number of rounds in AES algorithm (depends on key size)
$\oplus$	XOR (exclusive-ORed)
OFB	Output FeedBack mode
CFB	Cipher FeedBack mode
CBC	Cipher Block Chaining mode
CTR	Counter mode
ECB	Electronic Code Book mode
CMAC	Cipher-based Message Authentication Code
MAC	Message Authentication Code
IV	Initialisation vector
Chosen plaintext attack (CPA)	Attacks in which the attacker chooses a set of plaintexts and is able to obtain respective ciphertexts
Chosen Ciphertext Attack (CCA)	Attacks where the attacker chooses a set of ciphertexts and is able to obtain respective plaintexts
Non-malleability	Prevents the ciphertext from being altered in a way that changes the plaintext
Padding Oracle	Outputs if the ciphertext is correctly padded
Confidentiality	Protection of information from unauthorised access
Integrity	Ensures information has not been altered in an unauthorised manner
Authentication	Verifies the identity of entities participating
m	Message
k	Secret key
E	Encryption algorithm
c	Ciphertext
D	Decryption algorithm
K	Set of keys
M	Message space
C	Ciphertext space
GCM	Galois/Counter Mode
GMAC	Galois Message Authentication Code

Contents

List of Figures

List of Tables

2 Introduction

This project will explore vulnerabilities and potential attacks on inadequately designed AES (Advanced Encryption Standard) systems. While AES is a widely recognised encryption standard known for its robustness and efficiency, the quality of its implementation can vary significantly across different applications. Improper cryptographic practices and weak key management can introduce security flaws that adversaries can exploit.

The motivation behind this research stems from the growing reliance on encryption in securing sensitive data across digital platforms. As cyber threats evolve, even minor deviations from cryptographic best practices can lead to serious consequences. Therefore, understanding how implementation flaws affect AES security is essential for developers, security professionals, and researchers alike.

This section of the report will present the aims and objectives that would offer a framework of the research direction. Additionally, the background will provide insight into AES. The literature review will analyse existing research on AES vulnerabilities and attacks.

2.1 Aims and Objectives

The aim of this project is to highlight the critical importance of adhering to AES standards to ensure robust security and mitigate vulnerabilities in encryption systems. By analysing common weaknesses in AES implementations, this project seeks to emphasise the necessity of strict compliance with standards. By intentionally deviating from best practices in AES, the research will identify specific vulnerabilities that arise due to incorrect configurations.

To achieve this goal, several objectives have been outlined to examine how deviations from AES standards affect security. One of the crucial objectives is to select a specific mode of operation to illustrate the errors and security flaws that can occur when it is incorrectly implemented, such as improper handling of initialisation vectors (IV).

The project would also involve developing an AES encryption/decryption system using a chosen programming language. Within this implementation, intentional errors will be introduced to deviate from the standards, such as flawed padding schemes and misused initialisation vectors. The purpose of this simulation is to investigate how such variances affect encryption results and compromise confidentiality and data integrity.

Through a series of experiments and analyses, the project will assess how these deviations impact security. This would include identifying how and why certain errors weaken cryptographic resilience and what type of attacks they might expose the system to. This will measure the security deterioration that occurs when standards are abandoned by documenting each vulnerability along with its corresponding impact severity.

Finally, after documenting the observed vulnerabilities, the project will implement solutions to fix the identified vulnerabilities. This would include updating the flawed implementation to align with the AES standards and verifying its robustness against potential attacks. Corrective measures will include the usage of CMAC (Cipher-Based Message Authentication Code), the proper use of initialisation vectors, secure key generation, and appropriate padding schemes.

By achieving these goals, the project hopes to close the knowledge gap between theoretical and practical cryptographic security, providing important insights into how bad execution can compromise even well-known algorithms like AES.

3 Background and Literature Review.

To establish a solid foundation for this project, this section introduces the core cryptographic concepts relevant to AES, with a primary focus on security notation, encryption notation, and block cipher fundamentals. Understanding these principles is essential for evaluating the security posture of AES implementations and identifying potential vulnerabilities.

The section begins by outlining symmetric encryption and distinguishing block ciphers from stream ciphers, before presenting the structure and historical development of AES, including its design goals and supported modes of operation. Special attention is given to the internal mechanics of AES, such as its key expansion strategy and multi-round structure. A technical explanation of each encryption round, including the SubBytes, ShiftRows, MixColumns, and AddRoundKey transformations, provides insight into the layered operations that contribute to the cryptographic strength of AES. In addition, the section introduces authentication primitives, such as CMAC, which extend AES to provide data integrity and authenticity alongside its core encryption capabilities.

3.1 Symmetric encryption.

According to [1], data encryption is the process of converting plaintext into hidden text in order to enable secure communication, whereas decryption is the process of obtaining the original text from the hidden ciphertext. Every encryption method aims to make the process of decrypting data without the encryption key as difficult as possible. Both encryption and decryption are accomplished via keys. There are numerous encryption algorithms used in information security. Key encryption can be divided into two main categories: symmetric (private) and asymmetric (public) [2].

Symmetric encryption, also known as secret key encryption, uses a single key called a private key to encrypt and decrypt data. A secure channel is required to share the private key between the sender and the receiver. To securely transmit a message to Bob, for instance, Alice and Bob must first decide on a secret key, *k*. Alice first uses the encryption algorithm E and key k to encrypt her message m, producing the ciphertext $c = E(k,m)$. After that, Bob uses the same key k and the decryption method D to decrypt it, yielding the plaintext $m = D(k,c)$ [3]. Since encryption and decryption both rely on the same key, maintaining its secrecy is critical. Security depends on keeping k a secret, although the encryption and decryption methods E and D are known by all.

Mathematically, symmetric encryption consists of a mapping $E : K \times M \to C$ where K is the set of keys, M is the message space, and C is the ciphertext space [3]. The function $E_k : M \to C, \quad m \to E(k, m)$ must be invertible for each key $k \in K$. This guarantees that the decryption function $D_k = E_k^{-1}$ can retrieve the original message. The symmetric encryption's basic correctness property, which can be mathematically

stated as follows [4]:

$$\forall k \in K, \forall m \in M : \mathrm{Dec}(k, \mathrm{Enc}(k, m)) = m$$

It ensures that any encrypted message will contain the exact original plaintext when it is decrypted using the same secret key.

3.2 Block ciphers.

Generally, symmetric encryption schemes fall into two categories: stream ciphers, which handle data as continuous streams of bits, and block ciphers, which encrypt data in blocks, which are defined length groups of bits [1]. A block cipher operates on a fixed size input block, commonly 64 bits or 128 bits, transforming it into ciphertext using a secret key [5]. When the same plaintext block is encrypted using the same key, this transformation for a given key yields the same ciphertext block.

The formal expression for a block cipher is $F : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$, where k is the key length, n is the block length, $\{0,1\}^k$ denotes all possible keys, and $\{0,1\}^n$ denotes the space of input and output blocks [6]. The cipher, functions as a keyed permutation $F_K : \{0,1\}^n \to \{0,1\}^n$ for a fixed key K. This permutation property maintains invertibility, which is crucial for decryption, by guaranteeing that each block of plaintext maps uniquely to a block of ciphertext.

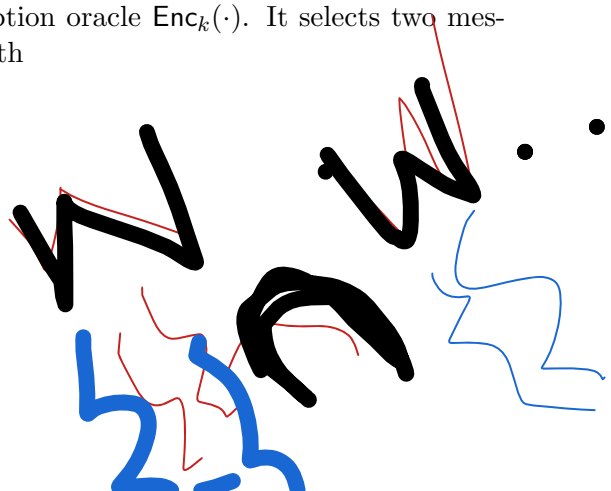
3.3 IND-CPA and IND-CCA Security.

The formal evaluation of cryptographic systems is dependent on how well they withstand adversarial models, especially chosen-input attacks. Indistinguishability under Chosen Plaintext Attack (IND-CPA) and Indistinguishability under Chosen Ciphertext Attack (IND-CCA) are two well-known security concepts in the symmetric scenario. These notions define how effectively an encryption algorithm hides information under active and adaptive threat scenarios.

If an adversary is unable to differentiate between two ciphertexts of equal length despite adaptive querying an encryption oracle, then the encryption scheme is IND-CPA secure. The adversary can submit multiple plaintexts to the encryption oracle and receive their corresponding ciphertexts. The adversary should not be able to determine which selected message was encrypted in order for the scheme to remain secure. [7].

Formally, the IND-CPA experiment for a symmetric encryption scheme $\Pi = (\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ proceeds as follows [7]:

1. A secret encryption key k is created using $\mathsf{Gen}(1^n)$, where n defines the desired security level.
2. The adversary $\mathcal{A}$ makes use of the encryption oracle $\mathsf{Enc}_k(\cdot)$. It selects two messages, m_0 and m_1, that are the same length



3. A bit b is randomly picked (either 0 or 1). The message m_b is encrypted to produce a ciphertext $c^* = \mathsf{Enc}_k(m_b)$, which is sent back to $\mathcal{A}$.

4. After further interaction with the encryption oracle, $\mathcal{A}$ makes a final guess b' as to which message was encrypted

5. The experiment returns 1 if b' equals b (meaning the adversary guessed correctly), and 0 otherwise

The encryption method is deemed secure against CPA if no computationally efficient adversary can consistently do better than random guessing when trying to distinguish between ciphertexts.

In CCA, the adversary has the ability to encrypt any messages of its choice and also has the ability to decrypt any ciphertexts of its choice with the restriction that the challenge ciphertext may not be submitted for decryption. This setting models a powerful attacker who can manipulate ciphertexts and observe system responses—particularly relevant for modes like CBC, which are vulnerable without integrity checks.

Formally, the IND-CCA security experiment $\mathrm{PrivK}^{\mathcal{A},\Pi}_{\mathsf{cca}}(n)$ unfolds as follows [7].

1. A key k is produced using the key-generation algorithm $k \leftarrow \mathsf{Gen}(1^n)$

2. The adversary $\mathcal{A}$ is granted interactive access to two oracles:

   - An encryption oracle that returns $\mathsf{Enc}_k(m)$ for messages m.
   - A decryption oracle that returns $\mathsf{Dec}_k(c)$ for ciphertexts c

   Using this access, $\mathcal{A}$ selects a pair of equal-length messages (m_0, m_1).

3. A bit b is sampled uniformly at random from $\{0, 1\}$. The challenger encrypts m_b to create the challenge ciphertext $c^* = \mathsf{Enc}_k(m_b)$ and returns this to $\mathcal{A}$.

4. The adversary continues to query both oracles after receiving c^* but $\mathcal{A}$ is not allowed to send c^* to the decryption oracle.

5. Eventually, $\mathcal{A}$ produces a guess b'. If b' = b, the result is 1. Otherwise, the result is 0.

An encryption scheme achieves CCA security if no adversary—despite having computational resources and access to a decryption oracle (excluding the challenge ciphertext), can reliably distinguish which of two plaintexts was encrypted, performing no better than random guessing.

3.4 AES (Advanced Encryption Standard).

The National Institute of Standards and Technology (NIST) released AES, one of the symmetric block ciphers, in 2000 [8]. Due to its security flaws, the primary goal of this

method was to replace DES (Data Encryption Standard). The 56-bit length of the secret key was one of the primary issues. Concerns were expressed about the ability to resist brute-force attacks by well-funded adversaries [9]. The block length of 64 bits, even after Triple-DES was implemented, was still short enough to protect against collision attacks. Later, NIST invited professionals from all over the world who specialise in data security and encryption to present an innovative block cipher algorithm for data encryption and decryption.

Block lengths of 128 bits, key lengths of 128, 192, and 256 bits, and efficient implementation across a range of hardware and software were the primary requirements for the proposed algorithm [9]. After several criteria and security assessments, the Rijndel algorithm, now known as the AES algorithm, was selected in 2000 as the encryption method proposed by Belgian cryptographers Joan Daeman and Vincent Rijmen.

AES consists of a 128 bit fixed block size and a key size of 128, 192 or 256 bits. Its design is based on a substitution permutation network, which is a combination of substitution and permutation [10]. AES operates on a 4x4 column-major-order matrix of bytes called the *state*. These states have round fixed transformations [11]. To generate an output, the state passes through several tiers of permutation boxes (P-boxes) and substitution boxes (S-boxes). The round key required for each round is determined by an expansion process that uses the block cipher key as input. The number of rounds and key lengths of the three block ciphers vary, as shown in Table 1.

	Key length (bits)	**Number of rounds**
AES 128	128	10
AES 192	192	12
AES 256	256	14

Table 1: Key lengths and number of keys.

As mentioned above, the AES encryption algorithm requires three inputs: the input data (in), the number of rounds (Nr) and the round keys [11]. The following can be used to illustrate this.

$$AES(in, key) = Encrypt(in, Nr, KeyExpansion(key)$$

3.5 Encryption.

In order to guarantee the highest level of security, the AES algorithm depends on the number of rounds and has four subprocesses in each round: SubBytes, ShiftRows, MixColumns, and AddRoundKey, which can be seen in Figure 1.

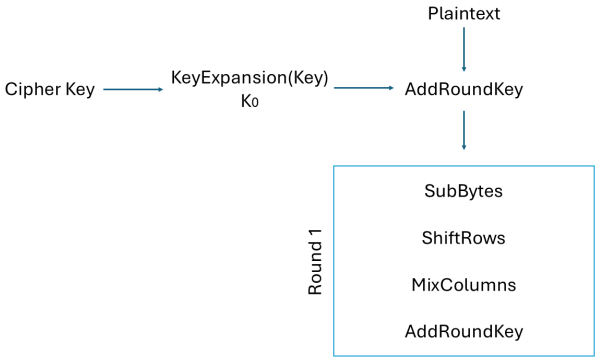


Figure 1: The first round in AES encryption, including subprocesses.

3.5.1 SubBytes

A nonlinear transformation of the state that is invertible, SubBytes applies an independent substitution table known as the S-Box to every byte in the state [11]. The AES S-Box was carefully designed to reduce the correlation between input and output, which means that input patterns are difficult to translate into output patterns [12]. In addition, it reduces the probability of various propagations, restricting the impact of modifying one aspect of the input on the output. Figure 2 illustrates how the values of an S-Box are used to replace each byte in the 4x4 state array.

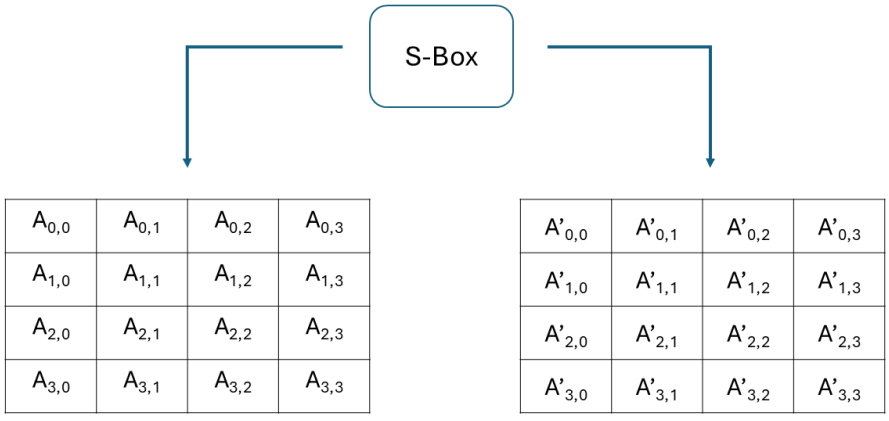


Figure 2: Illustration of SubBytes.

3.5.2 ShiftRows

ShiftRows transforms the state by clinically shifting the last three rows of the rows to the left. The number of positions shifted depends on the row index. Row 1 would shift 1 byte to the left, Row 2 would shift 2 bytes, and Row 3 would shift 3 bytes to the left [12]. This is illustrated below.

$$P = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \quad \text{ShiftRows}(P) = \begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,1} & A_{1,2} & A_{1,3} & A_{1,0} \\ A_{2,2} & A_{2,3} & A_{2,0} & A_{2,1} \\ A_{3,3} & A_{3,0} & A_{3,1} & A_{3,2} \end{bmatrix}$$

Figure 3: Effect of ShiftRows transformation on AES state matrix

3.5.3 MixColumns

MixColumns mixes the data within each vertical column of the 4x4 byte state. It takes four bytes in one column and combines them in a special way so that each byte affects all four results. This mixing is done using a system called GF(2^8) [12], which means that the bytes are added and multiplied with rules that help protect against hackers. AES distributes the data more uniformly by combining MixColumns and ShiftRows, so even slight modifications to the input can disrupt the entire outcome.

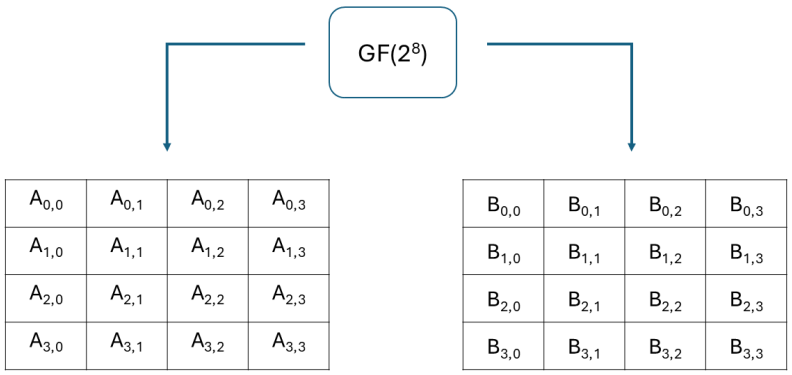


Figure 4: MixColumns Process.

3.5.4 AddRoundKey

The state and the subkey are combined in the AddRoundKey step. Using the KeyExpansion algorithm's key schedule, a subkey is generated from the main key on every

round. The size of each subkey matches that of the state. Bitwise XOR is used to combine the state and the subkey [10]. Figure 5 shows the AddRoundKey process of round A, with the subkey K.

$$\begin{bmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix} \oplus \begin{bmatrix} K_{0,0} & K_{0,1} & K_{0,2} & K_{0,3} \\ K_{1,0} & K_{1,1} & K_{1,2} & K_{1,3} \\ K_{2,0} & K_{2,1} & K_{2,2} & K_{2,3} \\ K_{3,0} & K_{3,1} & K_{3,2} & K_{3,3} \end{bmatrix} = \begin{bmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{bmatrix}$$

Figure 5: Illustration of the AddRoundKey process.

3.6 Key expansions.

As mentioned above, each round uses a unique round key via the AddRoundKey transformation. Using the Rijndael key schedule [12], the key expansion extends the cipher key into a broader range of round keys. It generates **4 x (Nr + 1)** words [11] using the original cipher key, where Nr is the number of rounds. The output is a linear array w[i] that contains all of the round keys. Rotate, Rcon, and Rijndael S-Box are among the main operations used by the key schedule.

- Rotation - takes a 32-bit word and rotates it to the left. For example, a0, a1, a2, a3 become a1, a2, a3, a0 [12].
- Rcon - a set of special values used during the process. Each Rcon value is a 4-byte word. Only the first byte holds the important data; the remaining three are always zeros [11].
- Rijndael S-Box - a value lookup table that generates an output by applying the S-Box to each of the input word's four bytes [12].

3.7 Modes of operation.

It should be possible for block cipher algorithms to encrypt plain texts of various sizes that would not fit on a predetermined block size. Using a mode of operation is one approach to accomplish this. To guarantee the confidentiality of lengthy messages, it is used to define how the cipher encrypts and decrypts blocks. NIST specified five modes of operation in 2001, which are applied to AES: OFB (Output FeedBack mode), CFB (Cipher FeedBack mode), CBC (Cipher Block Chaining mode), CTR (Counter mode), and ECB (Electronic Code Book mode). [13]. Since CBC mode will be the main focus of this project, it will be introduced in the next subsection.

3.7.1 Cipher Block Chaining mode (CBC).

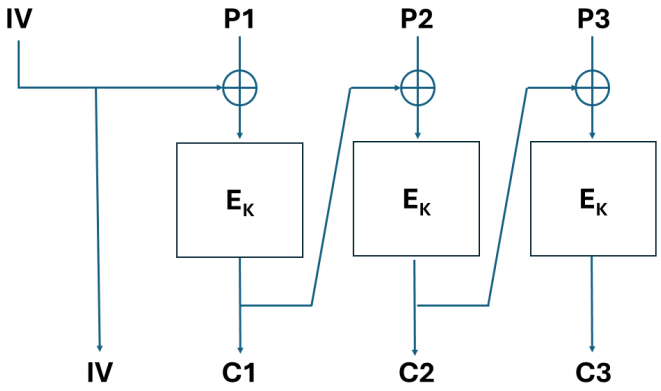


Figure 6: CBC mode.

The plaintext blocks are joined with the previous ciphertext blocks as part of the encryption process in CBC mode, which is a confidentiality mode. In order to combine the first plaintext block with CBC mode, an IV is required [14]. The IV does not have to be hidden, but it must be unpredictable. Patterns that attackers can take advantage of appear if repeatedly encrypting the same plaintext results in the same ciphertext. In order to attain cryptographic security, even when the same encryption key is used, every encryption of the same plaintext must produce a distinct ciphertext. This is accomplished with the use of the IV in CBC mode [13].

Figure 6 presents the encryption process. The plaintext (P1) undergoes an XOR operation with the IV before being encrypted using the key (k). Each block's (C1, C2, C3) encryption results are XORed with the subsequent plaintext block [13].In this manner, a distinct outcome is achieved when identical plaintext blocks are encrypted. In addition, an identical message will always be encrypted differently because a separate IV is used for each subsequent encryption. In CBC decryption, the first ciphertext block goes through the inverse CBC function. To recover the first plaintext block, the output is XORed with the IV [14]. The inverse CBC function is used for the corresponding ciphertext blocks, and the resultant block is XORed with the block that was generated before it.

CBC encryption and decryption are defined as follows [14]:

Encryption

$$C_1 = \text{Enc}_k(P_1 \oplus IV)$$
$$C_x = \text{Enc}_k(P_x \oplus C_{x-1}), \quad \text{for } x = 2, \ldots, n$$

Decryption

$$P_1 = \text{Dec}_k(C_1) \oplus IV$$
$$P_x = \text{Dec}_k(C_x) \oplus C_{x-1}, \quad \text{for } x = 2, \ldots, n$$

If the original plaintext does not meet the AES block size requirement, the system increases the message length by appending additional bits, a process known as padding. Padding adds a pattern of bits to the end of the message to fill out the final block of the message [14]. One example method involves appending a single '1' bit followed by the minimum number of '0' bits required to reach the block boundary. Padding ensures that every message, regardless of its original length, is properly processed by CBC encryption and can be unambiguously restored during decryption. To ensure that the receiver does not mistakenly remove bits from an unpadded message, padding is applied even when the last block is already full, ensuring consistency across all messages.

3.7.2 Characteristics of CBC mode.

The strengths and limits of CBC mode in applications are influenced by a number of unique features. Its sequential reliance is one of its primary characteristics. By guaranteeing that identical plaintext blocks produce distinct ciphertexts, chaining improves security. This means that while the decryption process can be parallelised, the encryption process cannot [14].

The vulnerability of CBC mode to birthday attacks is one of its drawbacks. When encrypting huge amounts of data with the same key, it takes advantage of the mathematical likelihood of collisions in ciphertext blocks [15]. Error propagation is another characteristic. The decryption of that block and the one that follows will be impacted if a bit in the plaintext is tampered with or corrupted during transmission [13]. This helps detect tampering but also reduces fault tolerance. CBC mode is not self synchronous meaning that it will not recover itself from insertions or deletions of ciphertext bits.

For security, the IV must be unpredictable and unique for each encryption. A reused or predictable IV can expose patterns in the ciphertext, leading to potential information leakage. Notably, while CBC is semantically secure under chosen plaintext attack (CPA) with a random IV, it fails this guarantee if the IV is simply a nonce (a non-repeating but potentially predictable value) [13].

CBC mode is vulnerable to padding oracle attacks, which will be demonstrated in this

project. Because CBC works on fixed-size blocks, padding is often required to align the plaintext to the cipher's block size. A padding oracle is an algorithm that provides an adversary with information about the authenticity of the plaintext padding when the ciphertext is submitted. If an attacker can observe whether the padding is valid during decryption, they can manipulate the ciphertexts to reconstruct the original plaintext [16].

While Cipher Block Chaining (CBC) mode provides confidentiality, it does not guarantee integrity. Although unauthorised parties may not be able to read the ciphertext, they might still alter it, and the receiver would unknowingly accept and process it, making CBC mode malleable [15]. Without pairing it with a Message Authentication Code (MAC), the ciphertext could be maliciously modified without detection. This makes CBC vulnerable to chosen-ciphertext attacks (CCA), where adversaries manipulate ciphertexts and observe system behaviour to recover information.

To address these vulnerabilities, modern encryption systems incorporate integrity checks directly into the encryption process, an approach known as Authenticated Encryption (AE), which combines confidentiality with robust protection against ciphertext tampering.

3.8 Authenticated Encryption (AE).

Authenticated encryption refers to encryption schemes that provide both confidentiality and authenticity simultaneously [17]. AE schemes play a crucial role in secure communication protocols, especially in settings vulnerable to active adversaries. In IPsec and Transport Layer Security (TLS), AE schemes are frequently used to encrypt messages so that only authorised parties can read them, to make sure the message hasn't been altered in transit, and to verify that the message's origin is authentic.

This structure ensures not only that an adversary cannot read the message but also that any alterations to the ciphertext are detected and rejected. The combination prevents attackers from forging or modifying messages without detection. It can be constructed using standard block ciphers like AES in combination with MACs. One of the approaches is to first generate the ciphertext using encryption and then use the MAC to authenticate it, which will be demonstrated in this project with the use of CMAC to provide the integrity component.

Alternatively, there are authenticated encryption with associated data schemes such as AES GCM (Galois/Counter Mode) that integrate encryption with authentication into a single algorithm, reducing computational overhead and simplifying implementation. AES GCM combines AES encryption in counter mode with GMAC (Galois Message Authentication Code) for authentication [18]. This mode of operation was standardised by NIST in Special Publication 800-38D and has since become a cornerstone in secure communication protocols such as TLS, IPsec, and SSH.

3.9 Cipher-based Message Authentication Code (CMAC).

To address the need for integrity and authentication alongside confidentiality, cryptographic systems often incorporate message authentication techniques. CMAC is a symmetric key message authentication algorithm designed to ensure both the integrity and the authenticity of data. It is based on AES and was published by NIST in the special publication 800-38B [19]. CMAC improves upon the earlier CBC-MAC by addressing its vulnerability to variable-length messages and providing a more robust keying mechanism. The CMAC algorithm generates a fixed length authentication tag known as the Message Authentication Code (MAC) from a message and a secret key, which is used to verify that the message has not been altered or originated from a legitimate source.

The following are the steps of the CMAC algorithm, according to NIST Special Publication 800-38B [19]:

1. Subkey generation.
    - Encrypt a block of zeros using the block cipher under key K: $x = E_K(0^n)$.
    - If the most significant bit is 0, generate the subkey K_1 by left shifting x one bit.
    - Else, left shift x and then XOR the result with a constant Y_b to form K_1.
    - Repeat the same logic to derive subkey K_2 from subkey K_1
2. Message preparation.
    - The message is segmented into blocks of 128 bits.
    - If the length of the final block is complete, no padding is required.
    - If the length of the final block is shorter than the block size, a single 1 bit is appended, followed by zeros until the block reaches 128 bits.
3. Block processing.
    - The final block is XORed with the subkey K_1 if it is a complete block.
    - The final block is XORed with the subkey K_2 if padding has been applied.
4. MAC computation.
    - The chaining variable c_0 is initialised with a block of zeros.
    - Each block is XORed with the preceding output c_{i-1} and the result is encrypted using E_K.
    - Once all blocks are processed, the tag is calculated by encrypting the XOR of the last chaining value with the final processed block.

- The resulting tag T serves as a cryptographic fingerprint, used to verify the authenticity and integrity of the original message

To verify the authenticity of a message, the recipient recomputes the MAC using the same secret key and message. The computed MAC is compared to the original MAC received with the message. If they match, the message is valid and unaltered. If they differ, the message has been tampered with or was sent by an unauthorised source.

4 Objectives, Specification and Design

This section presents a comprehensive framework for the project, outlining its core goals and technical direction. It begins by detailing the project objectives that would guide the research into flawed AES implementations and their associated vulnerabilities. The section then introduces the software tools and development environment selected to carry out the project, followed by a clear specification of the system's functional requirements.

4.1 Project objectives.

This project aims to explore how flawed implementation of AES can introduce severe vulnerabilities into secure systems. This research aims to demonstrate that deviation from best practices, such as predictable IVs, weak key generation, and improper padding schemes, can create exploitable security gaps. The approach to the development and design of this encryption system is directly shaped by the aim and objectives established earlier in the report.

To fulfil the first objective, the project selects CBC mode as the operational focus due to its sensitivity to misconfigured IVs and padding routines. CBC's reliance on chaining blocks makes it a prime candidate to demonstrate how improper practices such as predictable IVs and inaccurate padding can undermine confidentiality. Using Python, the goal of creating a configurable AES encryption/decryption system will be accomplished. This system will be deliberately engineered to support both secure and insecure configurations. Intentional deviations will be embedded in the implementation to simulate realistic system security issues.

The third objective involves conducting experiments and attack simulations to assess the impact of these deviations. Techniques such as a padding oracle will be used to evaluate how incorrect error handling and padding verification can leak sensitive information. The objective to apply and validate corrective security measures in the same system will include randomised IVs and integration of CMAC to verify message authenticity and integrity.

4.2 Software and tool selection.

To support the development of the AES encryption and decryption system, this project made use of several software tools selected for their accessibility, cryptographic capabilities, and practical suitability.

The primary programming language was Python, chosen for its simplicity, flexibility, and widespread adoption in academic and security research. Python is free to use and open source, so it is broadly used for varied functions [20]. Due to its vast number of libraries and third-party packages, the programmer is able to execute complex functions easily. Its readable syntax also enabled rapid development and clearer debugging throughout all phases of the project.

Coding was performed within PyCharm, which is an integrated development environment (IDE) optimised for Python workflows. Developed by JetBrains, PyCharm offers intelligent code completion, real-time error detection, and visual debugging [21]. Additionally, it supports virtual environments and incorporates integrated testing and profiling tools, which greatly improved code quality and development productivity.

To implement cryptographic functionality, the project utilises the PyCryptodome library, which is a comprehensive Python package offering secure cryptographic primitives. A key focus was the use of CMAC to ensure message authenticity and integrity. This was achieved by leveraging two core modules:

- Crypto.Cipher - supplied the AES algorithm as the underlying block cipher, aliased as CryptoAES for streamlined integration.
- Crypto.Hash - provided the interface for generating CMAC values using AES.

Together, these components enabled the secure creation of CMAC digests, which were essential to follow the standards of maintaining integrity within the AES. Os.urandom was used to generate cryptographically secure random values, which was essential for generating 16-byte keys and IVs, according to the standards.

4.3 System requirements.

This section defines the functional requirements for AES encryption and decryption system to explore how flawed implementation of AES can introduce severe vulnerabilities into secure systems. The design supports both secure and insecure configurations to evaluate implementation risks.

The system must perform AES encryption and decryption. Given a plaintext message, cryptographic key, and an IV, it should correctly produce a ciphertext using AES CBC mode and also decrypt accordingly. The key must be configurable, with support for 128 bit, 192-bit, and 256-bit options. These sizes align with the official NIST AES specification and provide flexibility in testing how key length impacts performance and security [11]. The corresponding decryption function must accurately reverse this process, reconstructing the original plaintext from the ciphertext using the same key and IV.

To maintain randomness and eliminate predictability, the system uses cryptographically secure generation of IVs and keys. Specifically, the function **os.urandom** is employed to produce values with high entropy. Each encryption session will be using a freshly generated IV and key to eliminate attacks that exploit repeated or low entropy values.

To facilitate compliance with AES structure, the system must correctly expand the key using AES's key schedule. This ensures that the encryption rounds are correctly initialised for each key size. The system should include the rounds in the specification,

which are composed of SubBytes, ShiftRows, MixColumns, and AddRoundKey mentioned in section 3 [11].

When the original message length does not align with the block size requirement in AES, the system should apply a padding routine to extend the message appropriately. Padding is essential not only for structural compatibility but also to ensure unambiguous message recovery during decryption.

The padding routine used in this system appends a sequence of bytes where each byte's value equals the number of padding bytes added. For consistency and structural reliability, a full block of padding may be added even when the message already confirms the block boundary, ensuring that the padding can always be inferred or validated accurately.

In addition to core encryption and decryption capabilities, the system should incorporate CMAC to ensure the authenticity and integrity of ciphertexts. This function would compute a fixed length digest, enabling verification of message validity before decryption is performed. It would serve as a defence mechanism against tampering and replay attacks.

For the system to simulate and assess vulnerabilities, it would be necessary to switch between secure and insecure modes. Secure configurations would apply randomised IVs, strong keys, valid padding, and CMAC verification. In contrast, insecure configurations allow fixed IVs, weak keys, and padding errors to be introduced, replicating real-world flaws and enabling testing of how such deviations affect security.

4.4 Security deviations.

To effectively investigate how flawed AES implementations can compromise the integrity of encryption systems, the project intentionally would incorporate insecure configurations into its design. These deviations would replicate common errors found in real world applications, highlighting the practical consequences of neglecting standards.

4.4.1 Static or predictable IVs.

One critical deviation in this project involves the use of static or predictable IVs. NIST 800-38A specification on block cipher modes of operation [14], mentions that the IV should not be a secret, but for the CBC mode, the IV should be unpredictable. This requirement stems from the way CBC processes the first block of plaintext by XORing the block with the IV before encrypting. A static or predictable IV allows attackers to anticipate how the first block is transformed. Chosen Plaintext Attacks (CPAs) are based on the attacker's ability to predict IV, which allows them to create specific plaintexts to abuse the encryption process.

In CPA, the adversary selects the plaintexts and observes the resulting ciphertexts. The attacker will manipulate the first block of the plaintext to produce the desired ciphertext using the predictable IV and confirm guesses to identify the original plaintext. This

compromises the property known as semantic security, which guarantees that the ciphertexts do not reveal any information about the plaintext [15]. When identical plaintexts are encrypted, identical IVs generate identical ciphertexts. This allows adversaries to infer patterns across encrypted messages.

4.4.2 Weak key generation.

A fundamental requirement for secure encryption is a sufficiently large and unpredictable key space. AES supports key sizes of 128, 192 and 256 bits, which translates to 2^{128}, 2^{192}, and 2^{256} possible key combinations, respectively. AES is protected against brute force attacks since the key space should be larger than 100 bits to ensure that such attacks are impossible [22]. However, the strength of AES also heavily depends on how keys are generated and managed. When encryption keys are derived from low entropy sources or reused across sessions, the effective key space shrinks, undermining the security guarantees of AES.

Weak key generation is a critical security deviation that occurs when keys are constructed from predictable patterns or poor random generators. In such cases, attackers reduce the number of guesses required to break the encryption. If a key is used without key expansion, the system becomes vulnerable to dictionary and brute-force attacks. Similarly, when developers reuse keys across multiple sessions, attacks can correlate ciphertexts and launch targeted cryptanalysis. Because it takes advantage of the algorithm's implementation rather than its flaws, this deviation is more dangerous. Simulating low entropy keys demonstrates how easily encrypted data can be compromised. These scenarios highlight the importance of using cryptographically secure random number generators.

4.4.3 Bit-flipping attack.

A significant security deviation in AES CBC mode arises from its vulnerability to bit-flipping attacks, a direct consequence of its malleability. This vulnerability arises because the decryption of a ciphertext block in CBC mode involves XORing it with the next ciphertext block. So as a result, tampering with one ciphertext block leads to predictable changes in the subsequent block, which is a property that attackers can exploit [15].

A bit flipping attack allows an adversary to modify specific bits in a ciphertext without the encryption key to manipulate the plaintext. If an adversary intercepts a ciphertext sequence, they can deliberately flip a specific bit in a block and reintroduce the altered ciphertext into the communication channel [23]. This mechanism enables the attacker to induce controlled changes in the decrypted plaintext without access to the encryption key. The first affected block will be corrupted due to the tampering, but the second block where the bit flip lands will reflect the attacker's intended modification. This attack does not require knowledge of the encryption key and succeeds purely by manipulating ciphertext structure

This attack highlights AES CBC's lack of non-malleability, which prevents the ciphertext

from being altered in a way that changes the plaintext [15]. AES CBC mode is known to not provide built-in integrity or authenticity checks, as it accepts ciphertexts even if the data is tampered with [24]. As a result, systems relying solely on AES-CBC for encryption are exposed to data manipulation, unauthorised control flow changes, and malicious payload injection. The decrypted output may violate confidentiality and integrity by seeming syntactically accurate while being semantically changed.

This deviation underscores the importance of pairing CBC-mode encryption with robust integrity checks, such as MACs or authenticated encryption schemes, to prevent attackers from exploiting its malleable structure.

4.4.4 Padding oracle attack.

Inadequate padding practices lead to security flaws in AES CBC mode. According to NIST [14], padding is required when the length of the plaintext is not a multiple of the block size. One of the padding methods is to append padding bytes to the plaintext, each indicating the number of padding bytes added. However, improperly applied or validated padding creates weaknesses that allow specific cryptographic attacks.

The most notable exploitation of this weakness is the padding oracle attack, first introduced by Serge Vaudeney in his 2002 paper [15]. Vaudeney shows how the usage of a valid padding oracle (an interface that reveals whether the decrypted ciphertext has correct padding) can decrypt arbitrary ciphertexts that were encrypted using 'Pad-then-CBC' mode of operation. In this mode of operation, if the final block of plaintext is not completely filled, extra bytes are added to fill it [25]. These extra bytes all have the same value, which indicates how many padding bytes were added. If the message already fits perfectly into the block size, a full block of padding is added, with each byte set to the block size value. This ensures that during decryption, the system can always tell where the actual message ends and padding begins.

The Vaudenay attack works by prepending a crafted block to a target ciphertext and submitting it to the oracle. If the padding is valid, the attacker can infer the last byte of the plaintext. If it's invalid, they iteratively modify the crafted block until the response changes. Repeating this strategy enables the recovery of the entire plaintext, one byte at a time [25]. Subsequent works reinforced Vaudenay's findings, emphasising how widespread the vulnerability is. Black and Urtubia expanded the scope by analysing seven padding schemes and successfully breaking five [15]. Canvel, Hiltgen, Vaudenay, and Vuagnoux applied the attack to SSL/TLS implementations. In order to illustrate this security deviation, this project would also employ Vaudenay's attack to retrieve the plaintext without the secret key and the IV.

5 Methodology and Implementation

The methodology adopted in this project is experimental and simulation-based, designed to explore how deliberate deviations from AES standards lead to exploitable vulnerabilities. Through progressive stages of implementation, the system simulates both correct and incorrect cryptographic practices to highlight security flaws in AES-CBC mode. This approach is especially relevant considering the widespread assumption that using a standard algorithm such as AES guarantees security. In reality, the resilience of any cryptographic system is not solely dependent on the strength of the algorithm but critically on how that algorithm is applied.

5.1 AES CBC Implementation.

5.1.1 AES CBC Encryption.

The complete source code referenced in this section can be found in Appendix A where each component is presented in full for further analysis and reference.

The implementation begins with the import of necessary modules. It retrieves `'urandom'` function from Python's standard library that generates cryptographically secure random bytes [26]. These bytes are important to produce strong encryption keys and IVs, ensuring unpredictability and security in AES CBC mode. The subsequent lines create a 16-byte random key and a 16-byte random IV using `'urandom(16)'`, both of which adhere to AES's default block size of 128 bits. However, as AES also supports key sizes of 192 and 256 bits, it can be specified in the implementation by adjusting the key length accordingly.

The substitution table `'S_BOX'` is used to perform nonlinear substitution of bytes via the `'sub_bytes'` transformation [11], increasing confusion by breaking linear relationships between the plaintext and the ciphertext. This table follows the original Rijndael specification and is applied during the SubBytes transformation. By embedding round-specific uniqueness, the round constants in the `'R_CON'` array are utilised during key expansion, specifically to break symmetry and reinforce key derivation. The values are derived from successive powers of 2 in the Galois field $\mathcal{GF}(\in^{\forall})$, starting with 0x01 [11].

The system proceeds to define several utility functions that abstract common cryptographic operations. In order to apply transformations like SubBytes, ShiftRows, and MixColumns, the `to_matrix` function transforms a byte array into a 4×4 matrix structure that represents the AES internal state. The `'from_matrix'` reconstructs the original byte stream from a matrix, maintaining fidelity with the block format required for encryption. `'xor_bytes'` function performs element-wise XOR operations between byte blocks, which is a fundamental operation in both the `'AddRoundKey'` transformation and CBC mode chaining.

To accommodate arbitrary lengths of input data, the encryption algorithm applies the PKCS#7 (Public Key Cryptography Standards #7) padding scheme via the

‘apply_padding()’ function [27]. This scheme makes sure that the plaintext complies with the block size of AES. As stated in subsection 3.7.1, additional bytes are appended to the end of the plaintext to ensure its total length aligns with the block size of 16 bytes [11]. Each padding byte carries a value equal to the number of bytes added, enabling precise and reliable removal during the decryption process.

The ‘apply_padding()’ function performs two critical operations. First, it calculates the number of bytes that are left in the last incomplete block to determine how many padding bytes are required and sets it to ‘pad_len’ [27]. Next, ‘pad_len’ amount of bytes are added to the plaintext. The padded message that results from this is a perfect multiple of the block size in length. For example, if 5 bytes are missing, it sets the variable ‘pad_len’ to 5. Then the function will append ‘0x05 0x05 0x05 0x05 0x05’, making it straightforward to remove during decryption. The padded plaintext is then divided into distinct blocks that correspond to AES's operational structure using ‘divide_blocks()’, allowing the result to be segmented into consistent 16-byte chunks.

The implementation then introduces the ‘sub_bytes’ function, which uses the ‘S_BOX’ to apply the nonlinear substitution to every byte in the state matrix. The function iterates over all rows and columns of the matrix using the nested loops, and for each ‘[row][col]’, it replaces the existing byte value with its corresponding entry from the ‘S_BOX’. This is illustrated in Figure 2.

Following the substitution, the bytes within each row of the state matrix are rearranged structurally by the ‘shift_rows(state)’ function [11]. In particular, the second, third, and fourth rows are moved to the left by one, two, and three places, respectively, but the first row stays the same. This step moves each row of the matrix by a different offset, which is illustrated in Figure 3. By spreading byte patterns throughout columns, this procedure improves diffusion.

Next, column mixing is introduced within the state matrix using the ‘mix_columns’ transformation. This process efficiently leverages the xtime() function to perform polynomial multiplication over the Galois field $GF(2^8)$ [11] which is illustrated in Figure 4 The algorithm begins by calculating x, the XOR of all four bytes in a column, and saving the first byte as y. For the first three positions in the column, each byte is updated by XOR-ing it with x and the result of xtime() applied to the XOR of that byte and its immediate successor. The fourth byte is similarly updated using x and the result of xtime() applied to the XOR of itself and y. This transformation effectively mixes the bytes in each column and reinforces diffusion across the AES state.

The ‘add_round_key()’ function performs the final step in each AES encryption round by combining the current state matrix with a round specific key [11]. It loops through each position in the 4×4 matrix and XORs each byte in the state with its corresponding byte in the round key, which is illustrated in Figure 5. Together with ‘sub_bytes’, ‘shift_rows’ and ‘mix_columns’, it completes the core internal transformations for each AES round.

The implementation encapsulates AES logic within an 'AES' class, allowing structured access to encryption capabilities and key scheduling. This class coordinates the transformation and utility functions that were previously mentioned, combining them into an efficient encryption process that complies with AES guidelines. First the class determines the number of rounds based on the input key length: 10 rounds for 16 bytes, 12 rounds for 24 bytes, and 14 rounds for 32 bytes, as per the AES specifications [11]. As stated in subsection 3.6, the `'expand_key_schedule()'` function creates round keys by converting the original key through a number of fundamental operations. These include byte rotation (`RotWord`), substitution using the Rijndael S-box (`SubWord`), and the incorporation of round constants (`RCON`) at defined intervals to ensure cryptographic strength.

For 256 bit keys, an additional substitution step is applied to every fourth word to further enhance diffusion. To retain a predictable yet secure link to the original key, each freshly generated word is derived via an XOR operation with a word positioned `'iter_size'` steps earlier. This process continues until the required number of words is produced, calculated as $(n_{rounds}+1)\times 4$, after which the words are grouped into distinct round keys [11]. The encryption procedure then employs these round keys in a sequential manner, closely following AES guidelines and guaranteeing a strong key for every round.

The `'encrypt_single_block()'` method carries out AES encryption on a single 16 byte block of plaintext. It begins by validating the block size and converting the input into a 4×4 state matrix. The method first applies the initial `'add_round_key()'` transformation, which combines the state with the first round key using bitwise XOR. It then iterates through the main encryption rounds, `'sub_bytes'`, `'shift_rows'`, `'mix_columns'`, and another `'add_round_key()'`. In the final round, `'mix_columns'` is omitted, following the standard AES specification [11], and the resulting state matrix is converted back into a byte array for output

The `'encrypt_cbc_mode()'` method facilitates AES encryption in CBC mode, which was illustrated in Figure 6. It begins by validating the IV using `'assert len(iv) == 16'`, ensuring that the IV conforms to the AES block size requirements. The method applies padding to the input data via `'apply_padding(plaintext)'`, after which the padded message is divided into blocks using `'divide_blocks()'`. Each block is then XORed with the previous ciphertext block and IV using `'xor_bytes()'`, and then encrypted with `'encrypt_single_block()'`. The result is collected in `'encrypted_blocks'`, and the final ciphertext is returned by concatenating the blocks using `b''.join(encrypted_blocks)`.

The `'create_cmac()'` function is designed to produce a CMAC using AES as the underlying cipher. `'CMAC.new(key, ciphermod=CryptoAES)'` initialises a new CMAC object using the provided key and AES as the encryption algorithm [28].
`'cmac_obj.update(message)'` feeds the plaintext into the CMAC algorithm. Then, `'cmac_obj.digest()'` finalises the CMAC calculation and returns the authentication tag. The function follows the steps defined by NIST Special Publication 800-38B [19]

which was also mentioned in the subsection 3.9.

The implementation involves the input plaintext, along with outputs including the ciphertext, key, IV, and the CMAC tag, each displayed using a formatted print statement.

5.1.2 AES CBC Decryption.

The complete source code referenced in this section can be found in Appendix B where each component is presented in full for further analysis and reference. Several functions from the encryption implementation, such as the `'to_matrix()'`, `'from_matrix()'`, `'xor_bytes()'`, `'add_round_key()'`, `'xtime()'`, and `'divide_blocks()'` are reused in the decryption code to maintain consistency across AES operations and reduce redundancy in logic. The inputs used in this AES CBC decryption and CMAC verification process include the encryption key (my_key), the encrypted message (ciphertext), the IV, and the CMAC value, used to authenticate the integrity of the decrypted message.

At its core, the AES class validates the key length and expands it into a series of round keys using `'expand_key_schedule()'`. This expansion creates a schedule of keys by utilising the `'S_BOX'` for substitution, key rotation, and round constant integration `'R_CON'` [11]. Conversion functions like `'to_matrix'` and `'from_matrix'` which were also mentioned in the previous section, facilitate transitions between linear byte arrays and structured AES state matrices.

The `'decrypt_block()'` method, which serves as the inverse of the `'encrypt_single_block()'` function described in the encryption implementation, performs AES decryption on a single 16-byte ciphertext block. It begins by validating the block size and converting the ciphertext into a state matrix using `'to_matrix()'`. Decryption is initiated with `'add_round_key()'` using the final round key, followed by `'inv_shift_rows()'` and `'inv_sub_bytes()'` to reverse the row shifts and byte substitutions. The method then enters the main round loop, applying `'add_round_key()'`, `'inv_mix_columns()'`, `'inv_shift_rows()'`, and `'inv_sub_bytes()'` in reverse round order to progressively unwind the encryption transformations. After completing all rounds, `'add_round_key()'` is applied once more using the initial round key. The resulting matrix is then converted back to bytes via `'from_matrix'` and returned as the decrypted output.

The inverse transformations, `'inv_sub_bytes()'`, `'inv_shift_rows()'` and `'inv_mix_columns()'`, are fundamental to restore the encrypted data to its original state within each decryption round. In order to ensure precise state value reversal, `'inv_sub_bytes()'` use the inverse substitution box (`'INV_S_BOX'`) to reverse non-linear byte replacements made during encryption [11]. By executing right circular shifts on each row, `'inv_shift_rows()'` essentially reverses the left shifts used during encryption and returns the rows to their initial places. The `'inv_mix_columns()'` function, which reconstructs each column using field arithmetic, inverts the diffusion effect created by `'mix_columns()'`, which was discussed in the previous section.

The ‘decrypt_cbc(self,ciphertext,iv)’ function is designed to reverse the ciphertext that was encrypted using AES CBC mode. It first validates the IV making sure the length is 16 bytes using ‘if len(ciphertext) != 16’. Since CBC uses the previous ciphertext block or IV for the first block in decryption [14], the decryption procedure initialises an empty list called ‘decrypted_blocks’ to store plaintext segments and sets ‘previous_block’ to the IV. The ciphertext is split into blocks by employing ‘divide_blocks()’, and each block is decrypted by calling ‘self.decrypt_block (current_block)’. The decrypted output is then processed with ‘xor_bytes (previous_block, decrypted)’ to apply the CBC chaining mechanism. This result is added to the ‘decrypted_blocks’ list, and ‘previous_block’ is updated to the current ciphertext block for the next iteration. After all blocks are handled, they are joined into a single byte string and finalised with ‘remove_padding()’ to produce the clean plaintext.

To eliminate any extra bytes that were introduced during encryption, a procedure to remove the padding must be taken [27]. In this implementation, padding is removed post decryption using the ‘remove_padding()’ function. This function targets PKCS#7 padding, a widely used scheme where the value of each padding byte indicates how many padding bytes were added. After decrypting the ciphertext, the function extracts the padding value from the last byte of the data using ‘pad_value = padded_data[-1]’. It then validates that the padding length is within acceptable bounds using ‘pad_value < 1 or pad_value > len(padded_data)’, preventing malformed or corrupted input from slipping through. The function isolates the padded segment with ‘pad_section = padded_data[-pad_value:]’ and checks each byte in this region using a generator expression. If any byte fails this check, the message is rejected via ‘raise ValueError("Padding validation failed")’, flagging potentially tampered ciphertext. Once verified, the function trims the padding using ‘return padded_data[:-pad_value]’, cleanly restoring the original message.

The ‘verify_CMAC(key, message, mac)’ function serves to authenticate the decrypted message by validating its integrity against the provided CMAC. With the original key, the message, and its corresponding CMAC, the function performs a secure integrity check. It begins by initialising a CMAC object with the key using ‘CMAC.new(key, ciphermod=CryptoAES)’ [28] and updates it with the message content via ‘Auth_Code.update(message)’. This regenerates the expected CMAC based on the message and key. Then, ‘Auth_Code.verify(mac)’ compares the regenerated CMAC with the provided one. If they match, the function returns “The message is authentic”, confirming that the message is unchanged and the correct key was used. If the values differ, it raises a ValueError, and the function returns “The message or the key is wrong”, indicating either a mismatch in the key, a modified message, or a corrupted CMAC. This approach is crucial for verifying data integrity and authenticity, particularly in secure communications where tampering must be detected reliably.

5.2 Bit-flipping attack.

6 Results, Analysis and Evaluation

It summarises the results obtained from the proposed design and methodology. The way to obtain the results should be described in detail. Analysis and evaluation have to be performed. Comparisons should be made. It should justify if the project's aims, objectives, requirements, and specifications have been achieved.

7 Legal, Social, Ethical, and Professional Issues

A chapter gives a reasoned discussion about legal, social, ethical, and professional issues within the context of your project problem. You should also demonstrate that you are aware of the Code of Conduct & Code of Good Practice issued by the British Computer Society (BSC) (`https://www.bcs.org/membership/become-a-member/bcs-code-of-conduct/`) for computer science projects and the Rule of Conduct issued by The Institution of Engineering and Technology (IET) (`https://www.theiet.org/about/governance/rules-of-conduct/`) for engineering projects. You should have applied their principles, where appropriate, as you carried out your project. You could consider aspects like the effects of your project on the public well-being, security, software trustworthiness and risks, Intellectual Property and related issues, etc.

The content of “Conclusion” is in “\contents\conclusion.tex”

8 Conclusion

It is a chapter to sum up the main points and findings of the work; how you achieve the project aims and address the research questions; the contributions and results you have achieved. Future plan and development can be mentioned in this section as well. It is normally in one or two pages.

References

[1] M. N. Alenezi, H. Alabdulrazzaq, and N. Q. Mohammad, "Symmetric encryption algorithms: Review and evaluation study," *International Journal of Communication Networks and Information Security*, vol. 12, no. 2, pp. 256–272, 2020. Available at: https://www.proquest.com/scholarly-journals/symmetric-encryption-algorithms-review-evaluation/docview/2440677681/se-2?accountid=11862.

[2] D. S. Abd Elminaam, H. M. Abdual-Kader, and M. M. Hadhoud, "Evaluating the performance of symmetric encryption algorithms.," *International Journal of Network Security*, vol. 10, no. 3, pp. 216–222, 2010. Available at: https://www.academia.edu/download/105571997/ijns-2010-v10-n3-p213-219.pdf.

[3] H. Delfs and H. Knebl, "Symmetric-key cryptography," in *Introduction to Cryptography: Principles and Applications*, pp. 11–48, Springer, 2015. Available at: https://doi.org/10.1007/978-3-662-47974-2_2.

[4] J. Duan, J. Hurd, G. Li, S. Owens, K. Slind, and J. Zhang, "Functional correctness proofs of encryption algorithms," in *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pp. 519–533, Springer, 2005. Available at: https://doi.org/10.1007/11591191_36.

[5] L. R. Knudsen and M. Robshaw, *The block cipher companion*. Springer Science & Business Media, 2011.

[6] M. Rosulek, *The joy of cryptography*. Oregon State University, 2021. Available at: http://164.52.217.92:8080/jspui/bitstream/123456789/969/1/Cyrptography.pdf.

[7] J. Katz and Y. Lindell, *Introduction to modern cryptography: principles and protocols*. Chapman and hall/CRC, 2007. Available at: https://doi.org/10.1201/9781420010756.

[8] A. M. Abdullah, "Advanced encryption standard (aes) algorithm to encrypt and decrypt data," *Cryptography and Network Security*, vol. 16, no. 1, p. 11, 2017. Available at: https://www.researchgate.net/profile/Ako-Abdullah/publication/317615794_Advanced_Encryption_Standard_AES_Algorithm_to_Encrypt_and_Decrypt_Data/links/59437cd8a6fdccb93ab28a48/Advanced-Encryption-Standard-AES-Algorithm-to-Encrypt-and-Decrypt-Data.pdf.

[9] A. Menezes and D. Stebila, "The advanced encryption standard: 20 years later," *IEEE Security Privacy*, vol. 19, no. 6, pp. 98–102, 2021. Available at: https://doi.org/10.1109/MSEC.2021.3107078.

[10] S. Rawal, "Advanced encryption standard (aes) and it's working," *International Research Journal of Engineering and Technology*, vol. 3, no. 8, pp. 1165–1169, 2016. Available at: https://www.academia.edu/download/54504660/IRJET-V3I8214.pdf.

[11] N. I. of Standards and T. (NIST), "Advanced encryption standard (aes)," standard FIPS PUB 197, U.S. Department of Commerce, 2001. Available at: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[12] J. Ridgway, "Understanding the advanced encryption standard," *steganosaur*, 2013. Available at: https://steganosaur.us/files/002_Understanding_the_Advanced_Encryption_Standard.pdf.

[13] D. Blazhevski, A. Bozhinovski, B. Stojchevska, and V. Pachovski, "Modes of operation of the aes algorithm," *The 10th Conference for Informatics and Information Technology (CIIT 2013)*, 2013. Available at: https://ciit.finki.ukim.mk/data/papers/10CiiT/10CiiT-46.pdf.

[14] N. I. of Standards and T. (NIST), "Recommendation for block cipher modes of operation: Methods and techniques," standard SP 800-38A, U.S. Department of Commerce, 2001. Available at: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38a.pdf.

[15] P. Rogaway, "Evaluation of some blockcipher modes of operation," *Cryptography Research and Evaluation Committees (CRYPTREC) for the Government of Japan*, 2011. Available at: https://www.cryptrec.go.jp/exreport/cryptrec-ex-2012-2010r1.pdf.

[16] R. Fedler, "Padding oracle attacks," *Network Architectures and Services*, vol. 83, 2013. Available at: https://www.net.in.tum.de/fileadmin/TUM/NET/NET-2013-08-1/NET-2013-08-1_11.pdf.

[17] M. A. Jimale, M. R. Z'aba, M. L. B. M. Kiah, M. Y. I. Idris, N. Jamil, M. S. Mohamad, and M. S. Rohmad, "Authenticated encryption schemes: A systematic review," *IEEE Access*, vol. 10, pp. 14739–14766, 2022.

[18] N. I. of Standards and T. (NIST), "Recommendation for block cipher modes of operation: Galois/counter mode (gcm) and gmac," standard SP 800-38D, U.S. Department of Commerce, 2007. Available at: https://doi.org/10.6028/NIST.SP.800-38D.

[19] N. I. of Standards and T. (NIST), "Recommendation for block cipher modes of operation: the cmac mode for authentication," standard SP 800-38B, U.S. Department of Commerce, 2005. Available at: https://doi.org/10.6028/NIST.SP.800-38B.

[20] S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram, and T. D. Singh, "Comparative analysis of python and java for beginners," *International Research Journal of Engi-*

neering and Technology (IRJET), vol. 7, no. 8, pp. 4384–4407, 2020. Available at: https://www.academia.edu/download/94738677/IRJET-V7I8755.pdf.

[21] Q. Nguyen, *Hands-on application development with PyCharm: Accelerate your Python applications using practical coding techniques in PyCharm*. Packt Publishing Ltd, 2019.

[22] M. Boussif, "On the security of advanced encryption standard (aes)," in *2022 8th International Conference on Engineering, Applied Sciences, and Technology (ICEAST)*, pp. 83–88, IEEE, 2022. Available at: https://doi.org/10.1109/ICEAST55249.2022.9826324.

[23] K. G. Paterson and A. K. Yau, "Cryptography in theory and practice: The case of encryption in ipsec," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 12–29, Springer, 2006. Available at: https://doi.org/10.1007/11761679_2.

[24] V. D. Gligor and P. Donescu, "Integrity-aware pcbc encryption schemes," in *International Workshop on Security Protocols*, pp. 153–168, Springer, 1999. Available at: https://doi.org/10.1007/10720107_22.

[25] R. Bardou, R. Focardi, Y. Kawamoto, L. Simionato, G. Steel, and J.-K. Tsay, "Efficient padding oracle attacks on cryptographic hardware," in *Annual Cryptology Conference*, pp. 608–625, Springer, 2012. Available at: https://inria.hal.science/hal-00691958/file/RR-7944.pdf.

[26] Python, "random — generate pseudo-random numbers." Available at:https://docs.python.org/3/library/random.html. Accessed: July 26, 2025.

[27] K. Haria, R. Shah, V. Jain, and R. Mangrulkar, "Enhanced image encryption using aes algorithm with cbc mode: a secure and efficient approach," *Iran Journal of Computer Science*, vol. 7, no. 3, pp. 589–605, 2024. Available at: https://doi.org/10.1007/s42044-024-00191-y.

[28] PyCryptodome, "CMAC." Available at:https://pycryptodome.readthedocs.io/en/latest/src/hash/cmac.html. Accessed: July 26, 2025.

[29] B. Zhu, "A pure python implementation of AES, with optional CBC, PCBC, CFB, OFB and CTR cipher modes." Available at: https://github.com/boppreh/aes, 2014. Accessed: July 19, 2025.

A Appendix — AES CBC Encryption

[29] [28]

```
from os import urandom
from Crypto.Hash import CMAC
from Crypto.Cipher import AES as CryptoAES

my_key = urandom(16)
iv = urandom(16)

S_BOX = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67,
        0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2,
        0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5,
        0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80,
        0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6,
        0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE,
        0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02,
        0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA,
        0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E,
        0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8,
        0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC,
        0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4,
        0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74,
        0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57,
        0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87,
        0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D,
        0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

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,
)
```

```
to_matrix = lambda data : [list(data[i:i+4]) for i in range(0,
    len(data), 4)]

from_matrix = lambda matrix: bytes(sum(matrix, []))

xor_bytes = lambda block1, block2 : bytes(i^j for i, j in zip(block1,
    block2))

def apply_padding(data):
    pad_len = 16 - (len(data) % 16)
    return data + bytes([pad_len] * pad_len)

divide_blocks = lambda msg : [msg[i:i+16] for i in range(0, len(msg),
    16)]

def sub_bytes(state):
    for row in range(4):
        for col in range(4):
            state[row][col] = S_BOX[state[row][col]]

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

def add_round_key(state, round_key):
    for row in range(4):
        for col in range(4):
            state[row][col] ^= round_key[row][col]

def xtime(byte):
    return (((byte << 1) ^ 0x1B) & 0xFF) if (byte & 0x80) else (byte <<
        1)


def mix_columns(state):
    for col in state:
        x = col[0] ^ col[1] ^ col[2] ^ col[3]
        y = col[0]
        for i in range(0,3): col[i] ^= x ^ xtime(col[i] ^ col[i + 1])

        col[3] ^= x ^ xtime(col[3] ^ y)

class AES:
    rounds = {16: 10, 24: 12, 32: 14}
    def __init__(self, key):
        assert len(key) in self.rounds
        self.n_rounds = self.rounds[len(key)]
        self._key_matrices = self.expand_key_schedule(key)
```

```
    def expand_key_schedule(self, key):
        columns = to_matrix(key)
        iter_size = len(key) // 4
        i = 1
        while len(columns) < (self.n_rounds + 1) * 4:
            word = list(columns[-1])

            if len(columns) % iter_size == 0:
                word.append(word.pop(0))
                word = [S_BOX[b] for b in word]
                word[0] ^= R_CON[i]
                i += 1

            elif len(key) == 32 and len(columns) % iter_size == 4:
                word = [S_BOX[b] for b in word]

            word = xor_bytes(word, columns[-iter_size])
            columns.append(word)

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

    def encrypt_single_block(self, block):
        assert len(block) == 16
        state = to_matrix(block)
        add_round_key(state, self._key_matrices[0])

        for i in range(1, self.n_rounds):
            sub_bytes(state)
            shift_rows(state)
            mix_columns(state)
            add_round_key(state, self._key_matrices[i])

        sub_bytes(state)
        shift_rows(state)
        add_round_key(state, self._key_matrices[-1])
        return from_matrix(state)

    def encrypt_cbc_mode(self, plaintext, iv):
        assert len(iv) == 16
        encrypted_blocks = []
        prev = iv
        for block in divide_blocks(apply_padding(plaintext)):
            encrypted = self.encrypt_single_block(xor_bytes(block, prev))
            encrypted_blocks.append(encrypted)
            prev = encrypted
        return b''.join(encrypted_blocks)

def create_cmac(key, message):
    cmac_obj = CMAC.new(key, ciphermod=CryptoAES)
    cmac_obj.update(message)
    return cmac_obj.digest()

aes_instance = AES(my_key)
```

```
plaintext = b""
ciphertext = aes_instance.encrypt_cbc_mode(plaintext, iv)
MAC_code = create_cmac(my_key, plaintext)

print(f"""Ciphertext: {ciphertext.hex()}
Key: {my_key.hex()}
IV:" {iv.hex()}
CMAC: {MAC_code.hex()}""")
```

B Appendix - AES CBC Decryption

[29] [28]

```
from Crypto.Hash import CMAC
from Crypto.Cipher import AES as CryptoAES

S_BOX = (
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67,
        0x2B, 0xFE, 0xD7, 0xAB, 0x76,
    0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2,
        0xAF, 0x9C, 0xA4, 0x72, 0xC0,
    0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5,
        0xF1, 0x71, 0xD8, 0x31, 0x15,
    0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80,
        0xE2, 0xEB, 0x27, 0xB2, 0x75,
    0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6,
        0xB3, 0x29, 0xE3, 0x2F, 0x84,
    0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE,
        0x39, 0x4A, 0x4C, 0x58, 0xCF,
    0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02,
        0x7F, 0x50, 0x3C, 0x9F, 0xA8,
    0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA,
        0x21, 0x10, 0xFF, 0xF3, 0xD2,
    0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E,
        0x3D, 0x64, 0x5D, 0x19, 0x73,
    0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8,
        0x14, 0xDE, 0x5E, 0x0B, 0xDB,
    0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC,
        0x62, 0x91, 0x95, 0xE4, 0x79,
    0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4,
        0xEA, 0x65, 0x7A, 0xAE, 0x08,
    0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74,
        0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57,
        0xB9, 0x86, 0xC1, 0x1D, 0x9E,
    0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87,
        0xE9, 0xCE, 0x55, 0x28, 0xDF,
    0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D,
        0x0F, 0xB0, 0x54, 0xBB, 0x16,
)

INV_S_BOX= (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3,
        0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x43,
        0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x95,
        0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2,
        0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C,
        0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46,
```

```
        0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x58,
        0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD,
        0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF,
        0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37,
        0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x62,
        0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC0,
        0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10,
        0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A,
        0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xBB,
        0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x14,
        0x63, 0x55, 0x21, 0x0C, 0x7D,
)

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,
)

to_matrix = lambda data : [list(data[i:i+4]) for i in range(0,
    len(data), 4)]

from_matrix = lambda matrix : bytes(sum(matrix, []))

xor_bytes = lambda block1, block2 : bytes(i^j for i, j in zip(block1,
    block2))

def remove_padding(padded_data):
    pad_value = padded_data[-1]
    if pad_value < 1 or pad_value > len(padded_data):
        raise ValueError("Invalid padding length")
    pad_section = padded_data[-pad_value:]
    if any(byte != pad_value for byte in pad_section):
        raise ValueError("Padding validation failed")
    return padded_data[:-pad_value]

def divide_blocks(data, block_size=16, require_padding=True):
    if require_padding and len(data) % block_size != 0:
        raise ValueError("Data length must be a multiple of block size
            when padding is required.")
    return [data[i:i + block_size] for i in range(0, len(data),
        block_size)]
```

```
def inv_sub_bytes(state):
    for i in range(4):
        for j in range(4):
            state[i][j] = INV_S_BOX[state[i][j]]

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

def add_round_key(state, round_key):
    for row in range(4):
        for col in range(4):
            state[row][col] ^= round_key[row][col]

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

def mix_single_column(col):
    x = col[0] ^ col[1] ^ col[2] ^ col[3]
    y= col[0]
    col[0] ^= x ^ xtime(col[0] ^ col[1])
    col[1] ^= x ^ xtime(col[1] ^ col[2])
    col[2] ^= x ^ xtime(col[2] ^ col[3])
    col[3] ^= x ^ xtime(col[3] ^ y)

def inv_mix_columns(state):
    for i in range(4):
        x = xtime(xtime(state[i][0] ^ state[i][2]))
        y = xtime(xtime(state[i][1] ^ state[i][3]))
        state[i][0] ^= x
        state[i][1] ^= y
        state[i][2] ^= x
        state[i][3] ^= y

    for col in state: mix_single_column(col) #mix columns

class AES:
    rounds = {
        16: 10,
        24: 12,
        32: 14
        }

    def __init__(self, key):
        assert len(key) in self.rounds
        self.n_rounds = self.rounds[len(key)]
        self._key_matrices = self.expand_key_schedule(key)
```

```
    def expand_key_schedule(self, key):
        columns = to_matrix(key)
        iter_size = len(key) // 4
        x = 1
        while len(columns) < (self.n_rounds + 1) * 4:
            word = list(columns[-1])
            if len(columns) % iter_size == 0:
                word.append(word.pop(0))
                word = [S_BOX[b] for b in word]
                word[0] ^= R_CON[x]
                x += 1
            elif len(key) == 32 and len(columns) % iter_size == 4:
                word = [S_BOX[b] for b in word]
            word = xor_bytes(word, columns[-iter_size])
            columns.append(word)
        return [columns[4 * x: 4 * (x + 1)] for x in range(len(columns)
            // 4)]

    def decrypt_block(self, ciphertext):

        if len(ciphertext) != 16: raise ValueError("Ciphertext block
            must be 16 bytes")

        state = to_matrix(ciphertext)
        add_round_key(state, self._key_matrices[-1])
        inv_shift_rows(state)
        inv_sub_bytes(state)

        for round in range(self.n_rounds - 1, 0, -1):
            add_round_key(state, self._key_matrices[round])
            inv_mix_columns(state)
            inv_shift_rows(state)
            inv_sub_bytes(state)
        add_round_key(state, self._key_matrices[0])

        return from_matrix(state)

    def decrypt_cbc(self, ciphertext, iv):
        if len(iv) != 16: raise ValueError("Initialization vector must
            be 16 bytes.")

        decrypted_blocks = []
        previous_block = iv

        for current_block in divide_blocks(ciphertext):
            decrypted = self.decrypt_block(current_block)
            plaintext_block = xor_bytes(previous_block, decrypted)
            decrypted_blocks.append(plaintext_block)
            previous_block = current_block
        return remove_padding(b''.join(decrypted_blocks))

def verify_CMAC(key, message, mac):
    Auth_Code = CMAC.new(key, ciphermod=CryptoAES)
```

```
    Auth_Code.update(message)

    try:
        Auth_Code.verify(mac)
        return "The message is authentic"

    except ValueError:
        return "The message or the key is wrong"

my_key =  bytes.fromhex("")
aes_instance = AES(my_key)
ciphertext = bytes.fromhex("")
iv = bytes.fromhex("")
MAC = bytes.fromhex("")

decrypted_data = aes_instance.decrypt_cbc(ciphertext, iv)
check = verify_CMAC(my_key, decrypted_data, MAC)

if check != "The message is authentic":
    print("CMAC Verification failed     message may have been tampered
        with.")
else:
    print(f"""The message is {decrypted_data}
The message is authentic.""")
```