128 Bit Key AES Implementation

Done By:

Enjy Ramadan 223385

Mazen Ashraf 221875

Omar Sherif 222595

Osama Hossam 223741

Introduction:

This document provides an in-depth explanation of the implemented Advanced Encryption Standard (AES) algorithm, its structure, and how it handles encryption and decryption. It also describes the GUI, designed for encrypting and decrypting data, illustrated with placeholders for screenshots.

Algorithm Overview:

AES (Advanced Encryption Standard) is a symmetric block cipher standardized by the National Institute of Standards and Technology (NIST) in 2001. It has become a global standard for secure data encryption.

The implementation uses a 16-byte key size (128-bit AES). It consists of several main steps:

- **Key Expansion:** Generates multiple round keys from the original key.
- **Initial Add Round Key:** The plaintext is XORed with the initial round key.
- **Rounds:** A series of transformations (SubBytes, ShiftRows, MixColumns, and AddRoundKey).
- **Final Round:** Same as a regular round, but without the MixColumns step.

Implementation Details:

Key Functions in the Algorithm:

1. **Key Expansion** Key expansion generates all the round keys from the initial key using transformations such as rotations, substitution (S-box), and XOR with round constants.

```
def _expand_key(self, master_key):
    key_columns = bytes2matrix(master_key)
    iteration_size = len(master_key) // 4
    i = 1

    while len(key_columns) < (self.n_rounds + 1) * 4:
        word = list(key_columns[-1])

        if len(key_columns) % iteration_size == 0:
            word.append(word.pop(0))  # Rotate
            word = [s_box[b] for b in word]  # SubBytes
            word[0] ^= r_con[i]
            i += 1
        elif len(master_key) == 32 and len(key_columns) % iteration_size == 4:
            word = [s_box[b] for b in word]  # SubBytes

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

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

2. **Encryption Steps** Encryption involves applying transformations over multiple rounds.

```
def encrypt_block(self, plaintext):
    assert len(plaintext) == 16
    state = bytes2matrix(plaintext)

    # Round 0
    add_round_key(state, self._key_matrices[0])
    self._log_step(state, 0, "AddRoundKey")

    # Main Rounds
    for i in range(1, self.n_rounds):
        sub_bytes(state)
        self._log_step(state, i, "SubBytes")

        shift_rows(state)
        self._log_step(state, i, "ShiftRows")

        mix_columns(state)
        self._log_step(state, i, "MixColumns")

        add_round_key(state, self._key_matrices[i])
        self._log_step(state, i, "AddRoundKey")

    # Final Round
    sub_bytes(state)
    self._log_step(state, self.n_rounds, "SubBytes")

    shift_rows(state)
    self._log_step(state, self.n_rounds, "ShiftRows")

    add_round_key(state, self._key_matrices[-1])
    self._log_step(state, self.n_rounds, "AddRoundKey")

    return matrix2bytes(state)
```

3. **Decryption Steps** The decryption process reverses the transformations in the reverse order, using inverse functions (e.g., inv_sub_bytes, inv_shift_rows).

```
def decrypt_block(self, ciphertext):
    assert len(ciphertext) == 16
    state = bytes2matrix(ciphertext)
    self._log_step(state, self.n_rounds, "Initial State")

    # Final Round
    add_round_key(state, self._key_matrices[-1])
    self._log_step(state, self.n_rounds, "AddRoundKey")

    inv_shift_rows(state)
    self._log_step(state, self.n_rounds, "InvShiftRows")

    inv_sub_bytes(state)
    self._log_step(state, self.n_rounds, "InvSubBytes")

    # Main Rounds
    for i in range(self.n_rounds - 1, 0, -1):
        add_round_key(state, self._key_matrices[i])
        self._log_step(state, i, "AddRoundKey")

        inv_mix_columns(state)
        self._log_step(state, i, "InvMixColumns")

        inv_shift_rows(state)
        self._log_step(state, i, "InvShiftRows")

        inv_sub_bytes(state)
        self._log_step(state, i, "InvSubBytes")

    # Round 0
    add_round_key(state, self._key_matrices[0])
    self._log_step(state, 0, "AddRoundKey")

    return matrix2bytes(state)
```

Supporting Functions

- **SubBytes and Inverse SubBytes** These functions perform non-linear substitution using the AES S-box.

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


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

- **ShiftRows and Inverse ShiftRows** Circularly shift rows of the state matrix to ensure diffusion.

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


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

- **MixColumns** Performs matrix multiplication to further mix the data within columns.
- **AddRoundKey** XORs the state matrix with the round key.

Padding Mechanism:

To handle messages of arbitrary lengths, PKCS#7 padding is used.

```
def pad(plaintext):
    """
    Pads the given plaintext with PKCS#7 padding to a multiple of 16 bytes.
    """
    padding_len = 16 - (len(plaintext) % 16)
    padding = bytes([padding_len] * padding_len)
    return plaintext + padding
```

Key Validation

To ensure the security of the encryption and decryption process, the key is validated using specific rules. These rules enforce the complexity and strength of the key:

1. **Length Check: The key must be between 8 and 64 characters long.**
2. **Content Validation: The key must include at least one number, one uppercase letter, and one special character (non-alphanumeric).**

The validation function:

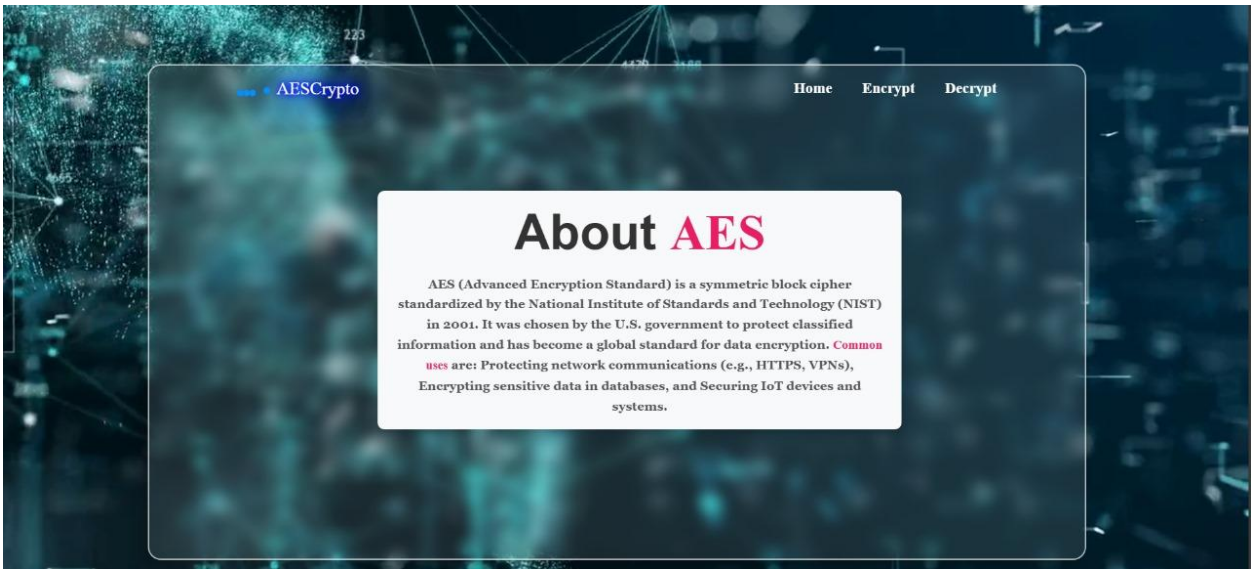
```
def validate_key(key):
    # Check if the key length is between 8 and 64 characters
    if len(key) < 8 or len(key) > 64:
        return False, "Key must be between 8 and 64 characters long."

    # Check if the key contains at least one number, one uppercase letter, and one special character
    if not re.search(r'[0-9]', key):  # At least one number
        return False, "Key must contain at least one number."
    if not re.search(r'[A-Z]', key):  # At least one uppercase letter
        return False, "Key must contain at least one uppercase letter."
    if not re.search(r'[\W_]', key):  # At least one special character (non-alphanumeric)
        return False, "Key must contain at least one special character."

    return True, "Key is valid."
```

GUI Description:

The GUI is designed to allow users to encrypt and decrypt plaintext or images using AES. Each component of the interface is described below.

Home Page:

The home page provides an introduction to AES and its uses.



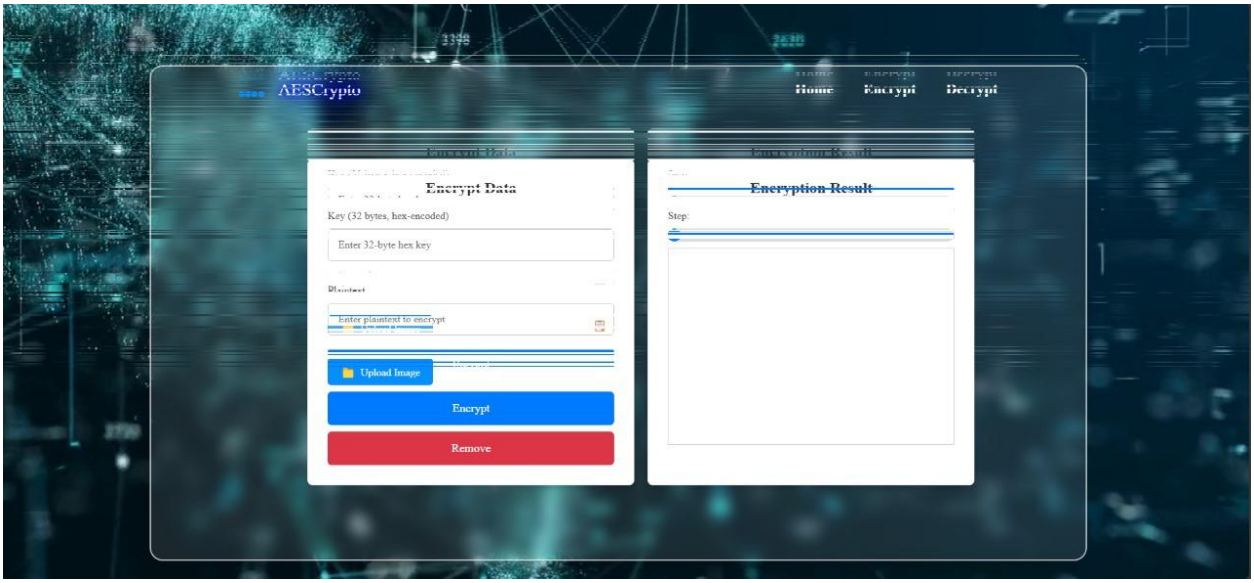
Encrypt Page:

This page allows users to:

1. Input a plaintext message or upload an image for encryption.
2. Enter a 32-byte hexadecimal key.
3. View the step-by-step logs of the encryption process.

Code for Integration:

```
def encrypt(plaintext, key):
    key = ensure_key_size(key.encode('utf-8'), AES_KEY_SIZE)
    if isinstance(key, str):
        key = key.encode('utf-8')
    if isinstance(plaintext, str):
        plaintext = plaintext.encode('utf-8')

    plaintext = pad(plaintext)

    aes = AES(key)
    blocks = split_blocks(plaintext)
    ciphertext_blocks = []
    for idx, block in enumerate(blocks):
        ciphertext_blocks.append(aes.encrypt_block(block))

    ciphertext = b''.join(ciphertext_blocks)
    ciphertext = ciphertext.hex()
    # Get the logs from AES instance
    logs = aes.get_logs()
    return ciphertext, logs
```



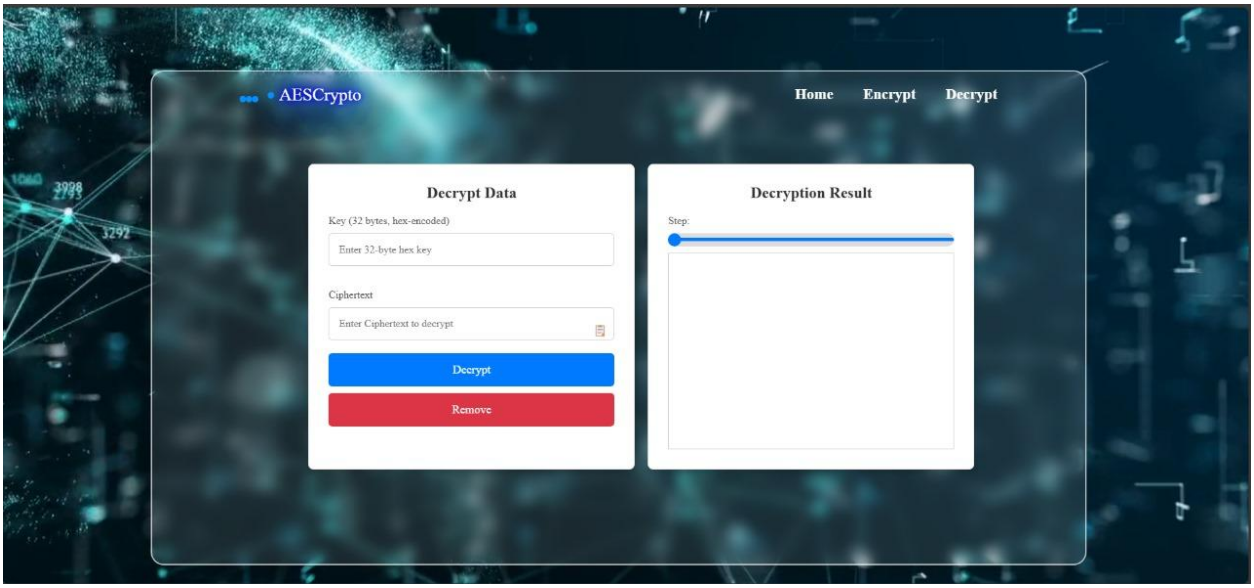
Decrypt Page:

This page allows users to:

1. Input a ciphertext for decryption.
2. Enter a 32-byte hexadecimal key.
3. View the step-by-step logs of the decryption process.

Code for Integration:

```
def decrypt(ciphertext, key):

    if isinstance(ciphertext, str):
        ciphertext = ciphertext.encode('utf-8')

    key = ensure_key_size(key.encode('utf-8'), AES_KEY_SIZE)




    aes = AES(key)
    blocks = split_blocks(ciphertext)
    plaintext_blocks = []
    for idx, block in enumerate(blocks):
        plaintext_blocks.append(aes.decrypt_block(block))

    plaintext = b''.join(plaintext_blocks)
    plaintext = unpad(plaintext)

    # Get the logs from AES instance
    logs = aes.get_logs()
    return plaintext, logs
```



Summary:

This implementation of AES provides a secure and efficient way to encrypt and decrypt data. The detailed logs generated during each step of encryption and decryption provide transparency into the process. The user-friendly GUI enhances accessibility, enabling users to seamlessly use AES for secure communications.