Implementation of the ECDH–AES System

Objective

The objective of this implementation is to establish a complete cryptographic system that:

1. Uses Elliptic Curve Diffie-Hellman (ECDH) to derive a shared secret key
2. Implements AES-128 encryption using a custom-generated S-box based on user-defined parameters

AES Overview

The Advanced Encryption Standard (AES) is a symmetric block cipher that encrypts data in 128-bit blocks using a 128-bit key through 10 rounds of substitution, permutation, and mixing operations. Each round applies four transformations: SubBytes (non-linear substitution), ShiftRows (transposition), MixColumns (mixing), and AddRoundKey (XOR with round key).

In our implementation, the SubBytes transformation uses a custom-generated S-box derived from a user-selected irreducible polynomial in GF(2^8), rather than the standard AES S-box.

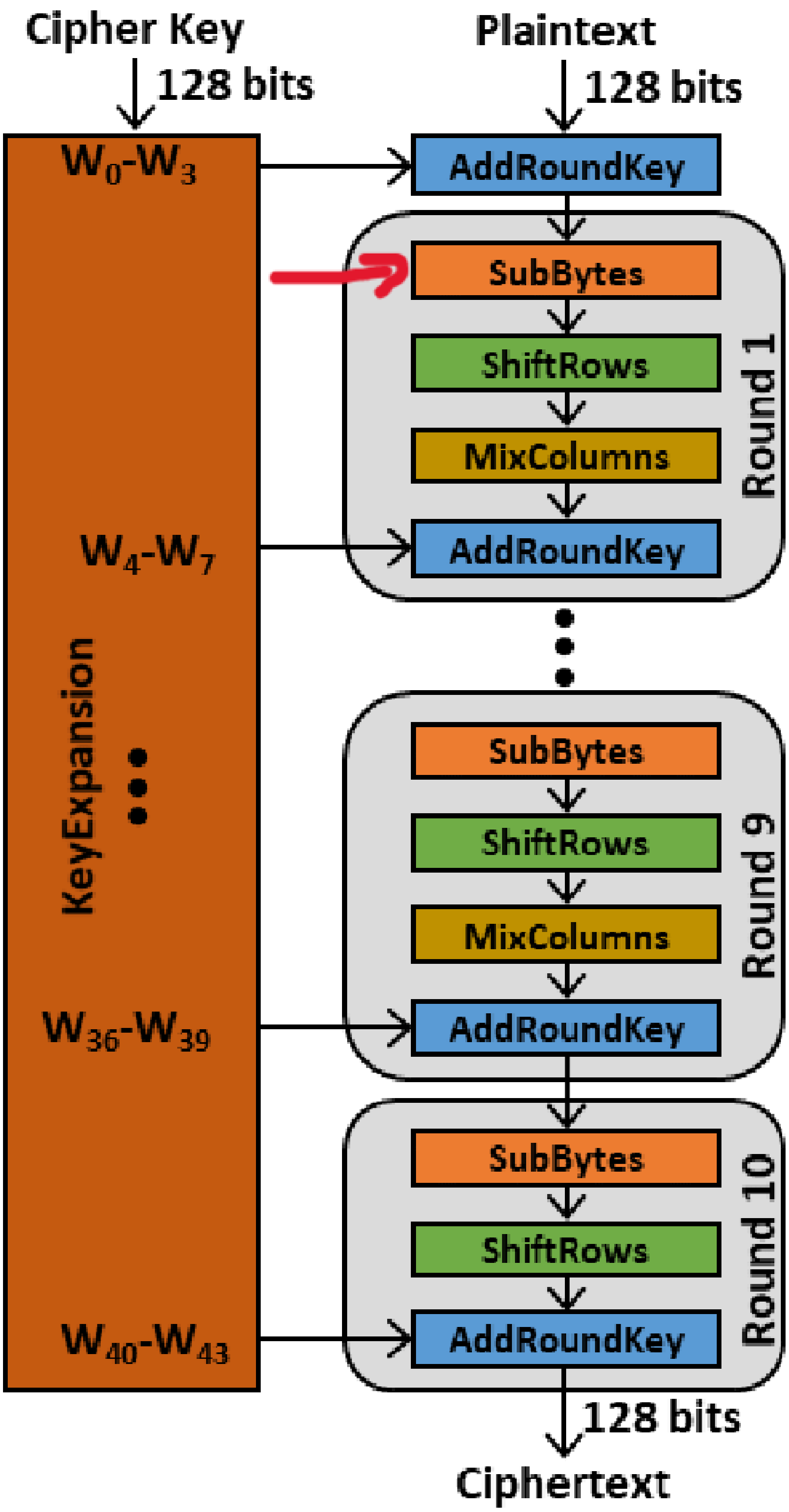


Figure 1: AES-128 Structure

Elliptic Curve Function

This section details the implementation of the elliptic curve function used in the ECDH key exchange.

1. **Elliptic Curve Construction**
   The code explicitly constructs an elliptic curve over a finite prime field.

   The curve equation used is:

   $$y^2 = x^3 + ax + b \pmod{p}$$

   In the code:

   The user manually inputs:

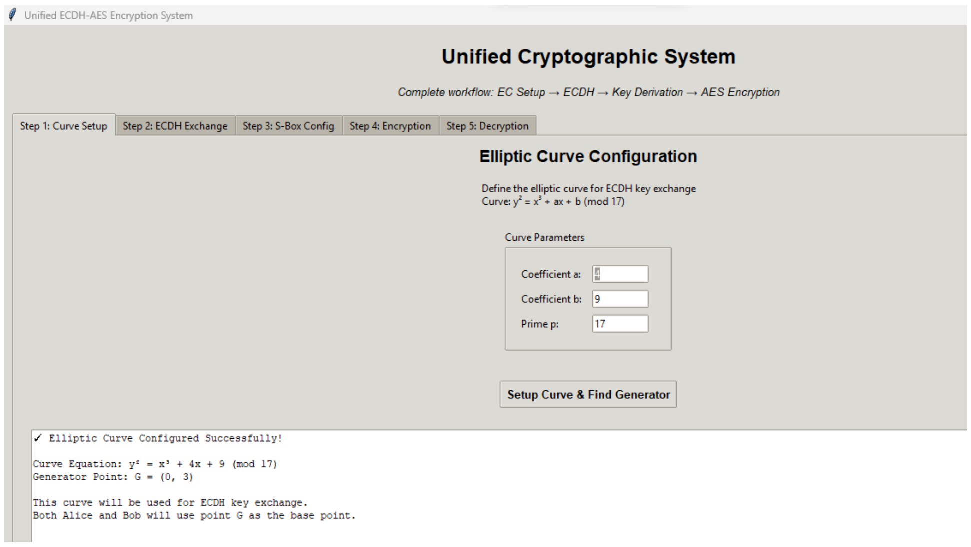
   - coefficient a
   - coefficient b

   The prime p is fixed (default: 17).

   The Curve class verifies the curve is non-singular using the discriminant:

   $$4a^3 + 27b^2 \not\equiv 0 \pmod{p}$$

   This ensures the curve is mathematically valid for cryptographic operations.


Unified ECDH-AES Encryption System
Unified Cryptographic System
Complete workflow: EC Setup → ECDH → Key Derivation → AES Encryption
Step 1: Curve Setup
Step 2: ECDH Exchange
Step 3: S-Box Config
Step 4: Encryption
Step 5: Decryption
Elliptic Curve Configuration
Setup Curve & Find Generator

Define the elliptic curve for ECDH key exchange
Curve: y² = x³ + ax + b (mod 17)

Curve Parameters

Coefficient a:	4
Coefficient b:	9
Prime p:	17

```
✓ Elliptic Curve Configured Successfully!

Curve Equation: y² = x³ + 4x + 9 (mod 17)
Generator Point: G = (0, 3)

This curve will be used for ECDH key exchange.
Both Alice and Bob will use point G as the base point.
```

Figure 2: Elliptic Curve Configuration Interface

2. **Generator Point Discovery**
   Instead of hardcoding a generator, the code searches for a valid point on the curve:

   - Iterates over all possible x and y
   - Checks whether (x, y) satisfies the curve equation
   - The first valid point found becomes the generator point G

   $\rightarrow$ This means the base point is not predefined and is computed dynamically from the curve parameters.

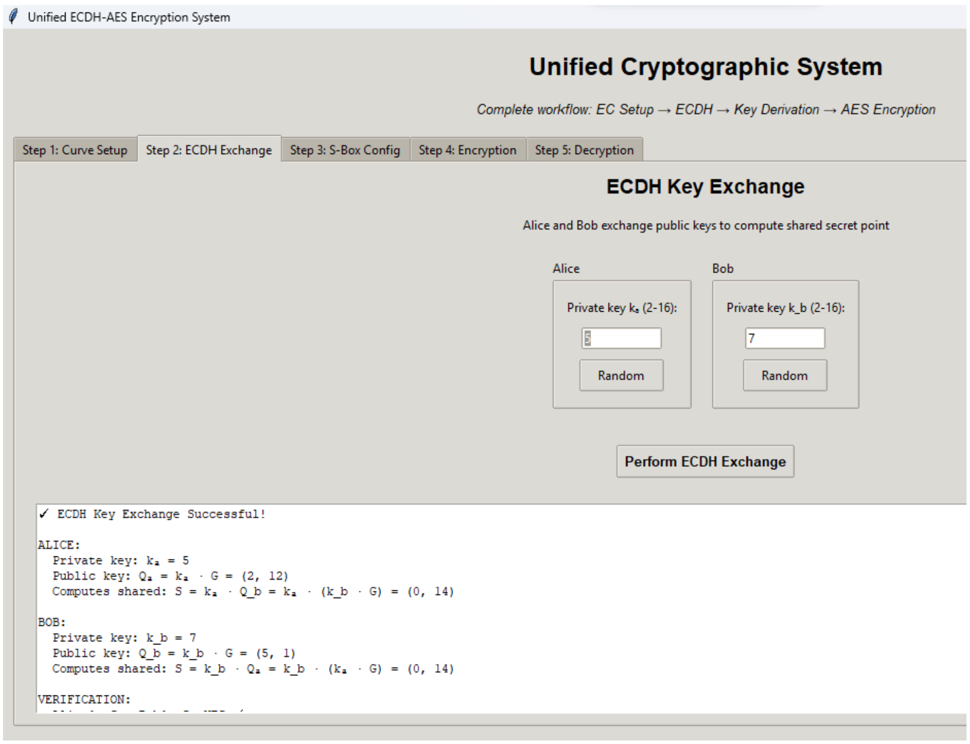
Manual Private Key Selection (Alice & Bob)

Unlike real-world systems where keys are fully random:
The code allows the user to manually enter:

- Alice's private key k_A
- Bob's private key k_B

Constraints:

- Values must be integers between 2 and 16
- These integers represent scalar values used in elliptic curve multiplication.


Unified ECDH-AES Encryption System

Unified Cryptographic System

Complete workflow: EC Setup → ECDH → Key Derivation → AES Encryption

Step 1: Curve Setup | Step 2: ECDH Exchange | Step 3: S-Box Config | Step 4: Encryption | Step 5: Decryption

ECDH Key Exchange

Alice and Bob exchange public keys to compute shared secret point

Alice
Private key kₐ (2-16): 5
Random

Bob
Private key k_b (2-16): 7
Random

Perform ECDH Exchange

```
✓ ECDH Key Exchange Successful!

ALICE:
  Private key: kₐ = 5
  Public key: Qₐ = kₐ · G = (2, 12)
  Computes shared: S = kₐ · Q_b = kₐ · (k_b · G) = (0, 14)

BOB:
  Private key: k_b = 7
  Public key: Q_b = k_b · G = (5, 1)
  Computes shared: S = k_b · Qₐ = k_b · (kₐ · G) = (0, 14)

VERIFICATION:
```

Figure 3: ECDH Key Exchange Interface

symmetric Key Generation

Using the manually chosen private keys:
Alice computes:

$$Q_A = k_A \cdot G$$

Bob computes:

$$Q_B = k_B \cdot G$$

These points are:

- Fully derived from the curve
- Safe to exchange publicly

ECDH Shared Secret Computation

The core ECDH logic is implemented exactly as defined mathematically:

Alice computes:

$$S_A = k_A \cdot Q_B = k_A \cdot (k_B \cdot G)$$

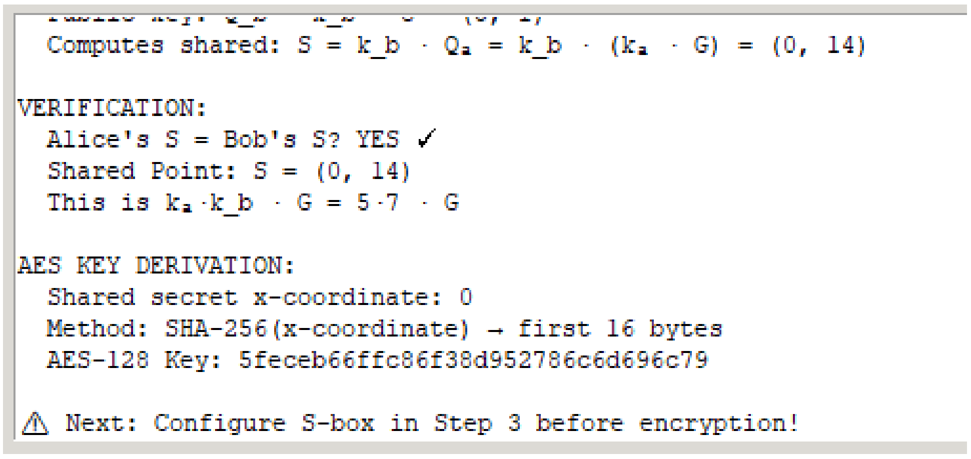
Bob computes:

$$S_B = k_B \cdot Q_A = k_B \cdot (k_A \cdot G)$$

The code verifies:

- $S_A.x = S_B.x$
- $S_A.y = S_B.y$

This confirms both parties derived the same elliptic curve point, without sharing private keys.



```
  Computes shared: S = k_b · Qₐ = k_b · (kₐ · G) = (0, 14)

VERIFICATION:
  Alice's S = Bob's S? YES ✓
  Shared Point: S = (0, 14)
  This is kₐ·k_b · G = 5·7 · G

AES KEY DERIVATION:
  Shared secret x-coordinate: 0
  Method: SHA-256(x-coordinate) → first 16 bytes
  AES-128 Key: 5feceb66ffc86f38d952786c6d696c79

⚠ Next: Configure S-box in Step 3 before encryption!
```

Figure 4: ECDH Key Exchange Results showing successful shared secret computation.

AES Key Derivation from Elliptic Curve Point

The elliptic curve point itself is not used directly as an AES key.

The code:

1. Extracts the x-coordinate of the shared point S
2. Converts it to bytes
3. Applies SHA-256:
$$\text{hash} = \text{SHA-256}(S.x)$$
4. Takes the first 16 bytes of the hash

$\rightarrow$ This produces a valid AES-128 symmetric key.
This step bridges public-key cryptography (ECDH) with symmetric encryption (AES).

Custom AES S-Box Generation

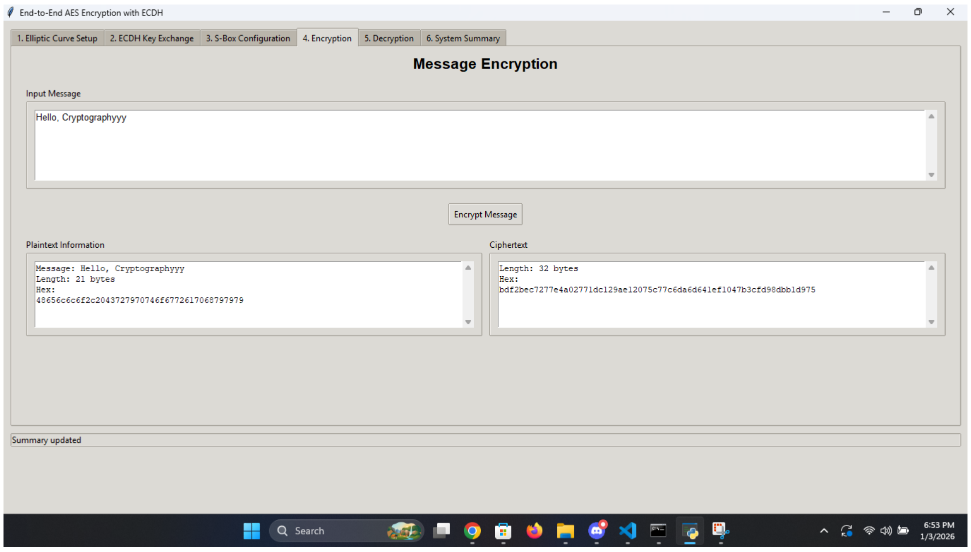
Instead of using a hardcoded AES S-box, the code generates the S-box from scratch:
Steps:

1. Compute multiplicative inverse in GF(2^8)
2. Apply the AES affine transformation
3. Generate the inverse S-box

This means:

- The S-box depends on the chosen polynomial
- AES behavior can be customized while remaining structurally correct

AES Encryption Process



```
Hello, Cryptographyyy
```

```
Message: Hello, Cryptographyyy
Length: 21 bytes
Hex:
48656c6c6f2c2043727970746f6772617068797979
```

```
Length: 32 bytes
Hex:
bdf2bec7277e4a02771dc129ae12075c77c6da6d641ef1047b3cfd98dbb1d975
```

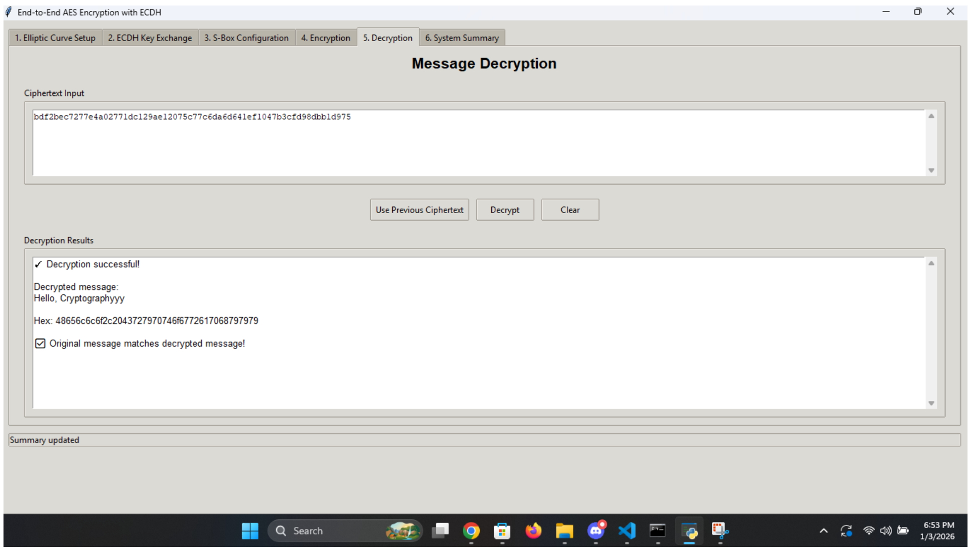
Figure 5: Encryption Process showing plaintext ”Hello, Cryptographyyy” being encrypted to ciphertext. The plaintext (21 bytes) is padded to 32 bytes and encrypted using the derived AES key.

The encryption logic follows the official AES-128 specification:

1. PKCS#7 padding
2. Initial AddRoundKey
3. 9 full rounds:
    - SubBytes (custom S-box)
    - ShiftRows
    - MixColumns (GF arithmetic)
    - AddRoundKey
4. Final round (no MixColumns)

The result is a ciphertext of length multiple of 16 bytes.

AES Decryption Process



Message Decryption

Ciphertext Input

```
bdf2bec7277e4a02771dc129ae12075c77c6da6d641ef1047b3cfd98dbb1d975
```

Decryption Results

```
✓ Decryption successful!

Decrypted message:
Hello, Cryptographyyy

Hex: 48656c6c6f2c2043727970746f6772617068797979

☑ Original message matches decrypted message!
```

Summary updated

Figure 6: Decryption Process showing successful recovery of original message. The ciphertext is decrypted back to "Hello, Cryptographyyy" with verification that original message matches decrypted message.

Decryption reverses encryption exactly:

Uses:

- Inverse S-box
- Inverse ShiftRows
- Inverse MixColumns
- Removes PKCS#7 padding
- Recovers the original plaintext

Successful decryption proves:

- The ECDH-derived key is correct
- AES operations are correctly implemented

Conclusion

Elliptic Curve Diffie-Hellman effectively solves the symmetric key exchange problem by enabling secure shared secret establishment over insecure channels, which is then used to derive AES keys for efficient encryption.