CSE 667 Cryptography - Assignment 2

Sukran Kulecki - Spring 2025
Caleb Alexander, Matthew O'Connor, Ryan Holthouse, Huy Tran, Mathan Patel

---

Abstract:
This report documents our team's implementation of a custom cryptographic algorithm developed for our class. Our approach utilizes a 32-bit initial key expanded via SHA-256, a 16-round Feistel network structure, carefully designed S-boxes based on bent functions, and strategic initial and final permutation operations. The algorithm successfully balances security and performance while adhering to the assignment constraints.

Through our implementation, we demonstrated practical applications of fundamental cryptographic principles including confusion, diffusion, and the avalanche effect. Our algorithm incorporates multiple security features that provide protection against common cryptanalytic attacks, including linear and differential cryptanalysis.

1. Key Length

Our algorithm initially uses a 32-bit key length. We'll increase key size to 256-bit by using SHA-256, which will make our algorithm more secure as we are changing our key entirely. It is easy to break it by brute force as SHA-256 is deterministic and the key strength remains the same.(Alternatively we can use salt to make the key randomized but it's non-deterministic. To make it deterministic we need to use the same nuance in decryption that we used in encryption. To use same nuance we can randomly put nuance in decrypted text so the attacker doesn't know the location of nuance but we have to make it deterministic so while decrypting we know where the nuance is so we will use AES-CTR to randomly generate a location where to put nuance. We can add 32-bit (or 96-bit for 128-bit strength and so on) nuance for salt to make key length 64-bit).

2. Key Scheduling Algorithm

We divided the 256-bit key into 16 sub keys. Then for each round the key will shift by some bit(s), which is random and based on a 16-bit subkey for that round. By using AES-CTR we will generate a random number between 0 and 15 and the key will shift by that value.

The key scheduling algorithm transforms the initial 16-bit user key into multiple round keys through several steps:

- Master key generation: The 16-bit user key is passed to the SHA-256 to produce the 256-bit master key

- AES-CTR setup: The first 128 bits of the master key served as the AES encryption key, the next 96 bits (bits 128-223) form the nonce/IV for CTR mode. Finally, an initial counter value of 0 is used.
- Subkey derivation: The 256-bit master key is divided into 16 subkeys of 16 bits each:
- Round key generation: For each of the 16 rounds:
    - AES-CTR generates a pseudo random value using the current counter.
    - This value modulo 16 determines a shift amount (0-15)
    - The appropriate subkey (round mod 16) is **rotated left** by this shift amount
    - The resulting 16-bit value serves as the round key
    - The counter is incremented after each round key generation

3. Permutations

At the start an initial permutation (IP) is used to shuffle every other bit with its pair. In other words, splitting the key into pairs, each first and second bit is flipped. In the case of an 8-bit string “10101010”, the resulting bit string would be “01010101,” for example. This permuted set of bits then goes along through the middle phases of our encryption algorithm involving the round key.

Afterwards, the resulting bits are split into 4 even groups, where then the 1st & 3rd and 2nd & 4th groups of bits are swapped. This then has an XOR function applied to it with the original left side of the 32 bits.

At the very end, in the final permutation, the initial permutation is run once more, flipping each pair of bits in the final resulting output. The permutations utilized cause reverse-engineering to become much more complex due to the shuffled nature of the bits, along with an XOR to further mask the plaintext.

4. S-boxes

Our algorithm implements a non-linear S-box constructed from 16 distinct bent functions. Bent functions provide optimal nonlinearity and resistance to cryptanalysis.

- **Bent function:** A bent function is a Boolean function that exhibits maximum nonlinearity with respect to linear functions. Formally, a bent function f has the property that the Hamming distance H(f(x), l(x)) is maximized across all inputs x and all linear functions l. This nonlinearity makes bent functions excellent building blocks for cryptographically strong S-boxes. We utilized compositions of bent functions of the form:

$$f(x_1, x_2, \ldots, x_n) = x\_ax\_b \oplus x\_cx\_d \oplus \ldots \oplus x\_ix\_j$$

Where $a, b, c, d, \ldots, i, j \in \{1, \ldots, n\}$ and $a \neq b \neq c \neq \ldots \neq j$.

- **S-box architecture:** Our bent function maps our 16 bit input to a 16 bit output. The S-boxes are constructed as followed:

    1. First Layer: Bent function: We defined 16 bent functions $\{f_1, f_2, \ldots, f_{16}\}$ where each function operates on 16-bit inputs:
       $f_1(x) = x_1x_2 \oplus x_3x_4 \oplus x_5x_6 \oplus x_7x_8$
       $f_2(x) = x_1x_3 \oplus x_2x_4 \oplus x_5x_7 \oplus x_6x_8$
       …
       $f_{16}(x) = x_9x_{15} \oplus x_{10}x_{16} \oplus x_{11}x_{13} \oplus x_{12}x_{14}$
    2. Vector formation: We combine these functions to form a vector:
       $S(x) = (f_1(x), f_2(x), f_3(x), \ldots, f_{16}(x))$
    3. To ensure balanced output and avoid weak output (all 0s / all 1s), we apply a linear transformation
       $S'(x) = A \cdot S(x) + b \bmod 2$
       Where
        - A is a 16×16 binary MDS (Maximum Distance Separable) matrix derived from Hadamard matrices
        - b is a 16×1 random non-repeating vector
        - The multiplication is standard matrix-vector multiplication in GF(2)
    4. Notice that while applying A and b are linear transformations, and thus very invertible (A is specifically chosen to be invertible); the result of applying the S-Box to an input remains non-invertible because each bent function has no inverse.

5. Encryption Algorithm

Our encryption algorithm follows a 16-round Feistel network structure, combining elements from our key schedule, permutations, and S-boxes:

1. Upon receiving a 32-bit message (M) and 32-bit user key (K), the key is expanded to 256 bits using SHA-256 and divided into 16 different round keys as described in the key scheduling section.
2. The message M undergoes an initial permutation as described in the permutations section.
3. For each of the 16 encryption rounds:
    - The current state is split into left (L) and right (R) halves of 16 bits each
    - The new right half is computed as: R' = S(R XOR K_n) XOR L
        - R is XORed with the round key
        - The result passes through our S-box
        - The output is XORed with L to become R'
    - The new left half is simply L' = R
    - L' and R' are used as the left and right halves for the next round

4. After 16 rounds, the modified message undergoes a final permutation which is the inverse of our initial permutation (flipping pairs of two bits)

6. Decryption Algorithm

The decryption process is the reverse of encryption, which is a key advantage of the Feistel network structure:

1. The ciphertext first undergoes the inverse of the final permutation.
2. The round keys are either generated again (as in encryption) or can be pre-computed and stored for efficiency.
3. The rounds are processed in reverse order (15 down to 0), with each round:
    a. Taking the current left half (L) and right half (R)
    b. Computing L_prev = R XOR S(L XOR K_round)
    c. Setting R_prev = L
    d. Using L_prev and R_prev as the inputs for the previous round
4. After all rounds are reversed, the inverse of the initial permutation is applied to recover

7. Security Analysis

Our algorithm incorporates multiple security features that collectively provide resistance to common cryptanalytic attacks:

- While starting with a 32-bit key (the maximum allowed in the assignment), the expansion to 256 bits through SHA-256 significantly increases the computational resources required for a brute force attack, making it practically infeasible.
- Our permutation operations thoroughly mix the bits, destroying patterns that might otherwise be exploited by statistical analysis.
- Our non-linear S-box design provides strong protection against differential and linear attacks. The complex matrix transformation ensures that linear relationships between input and output bits are minimized.
- The use of 16 rounds ensures thorough mixing of data and keys, with each round adding to the complexity an attacker must overcome. It means it is really difficult to reverse engineering the process and the key, with many points of error for the attacker.

8. Pseudocode & Diagram

Pseudocode for encryption:

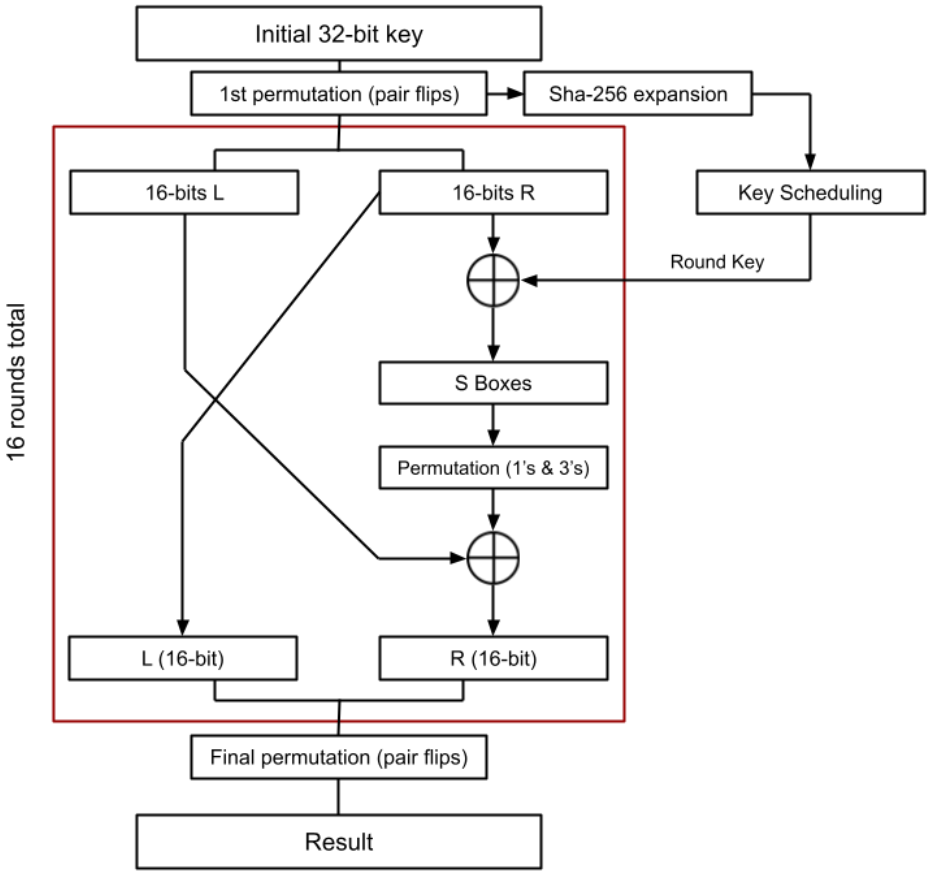
```
function Encrypt(plaintext, userKey):
    // Key Expansion
    masterKey = SHA256(userKey)                        // Generate 256-bit master
key
    aesKey = masterKey[0:127]                          // First 128 bits for AES key
```

```
nonceIV = masterKey[128:223]                    // Next 96 bits for nonce/IV
subkeys = divideInto16BitSubkeys(masterKey)     // Divide into 16 subkeys

// Initial Permutation
processed = pairPermutation(plaintext)

// Split into Left and Right halves (16 bits each)
L = processed[0:15]
R = processed[16:31]

// Initialize AES-CTR
counter = 0

// 16 Rounds of processing
for round = 0 to 15:
    // Generate round key
    randomValue = AES_CTR(aesKey, nonceIV, counter)
    shiftAmount = randomValue mod 16
    subkeyIndex = round mod 16
    roundKey = rotateLeft(subkeys[subkeyIndex], shiftAmount)
    counter++

    // Round operations
    temp = R
    R_xor_key = R XOR roundKey
    S_box_output = applySubstitution(R_xor_key)   // S-box operation
    R = S_box_output XOR L
    L = temp

// Combine L and R for final result
result = concatenate(L, R)

// Final Permutation
ciphertext = pairPermutation(result)

return ciphertext
```

Pseudocode for decryption:

```
function Decrypt(ciphertext, userKey):
    // Key Expansion (identical to encryption)
    masterKey = SHA256(userKey)
    aesKey = masterKey[0:127]
    nonceIV = masterKey[128:223]
    subkeys = divideInto16BitSubkeys(masterKey)

    // Generate all round keys in advance (same as encryption)
    roundKeys = []
    counter = 0
    for round = 0 to 15:
        randomValue = AES_CTR(aesKey, nonceIV, counter)
        shiftAmount = randomValue mod 16
        subkeyIndex = round mod 16
        roundKey = rotateLeft(subkeys[subkeyIndex], shiftAmount)
        roundKeys.append(roundKey)
        counter++

    // Apply inverse of final permutation
    processed = inverseFinalPermutation(ciphertext)

    // Split into Left and Right halves
    L = processed[0:15]
    R = processed[16:31]

    // 16 Rounds of processing in reverse order
    for round = 15 to 0 (decrement):
        temp = L
        L = R XOR applySubstitution(L XOR roundKeys[round])
        R = temp

    // Combine L and R
    result = concatenate(L, R)

    // Apply inverse of initial permutation
    plaintext = inverseInitialPermutation(result)

    return plaintext
```

Diagram of our encryption algorithm:



References:

1. Federal Information Processing Standards Publication 197, "Advanced Encryption Standard (AES)," November 2001.
2. M. Tolga Sakallı and Bora Aslan, "On the algebraic construction of cryptographically strong Boolean functions," available at http://www.singacom.uva.es/~edgar/cactc2012/trabajos/CACT2012_Sakalli.pdf
3. Federal Information Processing Standards Publication 46-3, "Data Encryption Standard (DES)," October 1999.
4. Federal Information Processing Standards Publication 180-4, "Secure Hash Standard (SHS)," August 2015.
5. Howard M. Heys, "A Tutorial on Linear and Differential Cryptanalysis," Cryptologia, vol. XXVI, no. 3, pp. 189-221, July 2002.