Term Paper - LED

Abhishek Singh Kushwaha[1], Rishit Agarwal[1] and Sourabh Dadore[1]

Indian Institute of Technology, Bhilai {abhisheksinghk,rishitag,sourabhd}@iitbhilai.ac.in

Abstract. In this paper we discuss about the design of LED Cipher and has Performed Differential & Integral cryptanalysis on it

Keywords: LED, AES-like, Lightweight, Block Cipher

1 Introduction

AES and its derivative ciphers are typically optimized for **high-speed software performance** but often fall short in delivering **lightweight hardware implementations**. In recent years, the rapid expansion of the **Internet of Things (IoT)** and **resource-constrained devices** has highlighted the need for a **lightweight block cipher** that is both **hardware-compact** and **software-efficient**. This is where the **LED cipher** becomes significant.

Beyond meeting the primary requirement of being lightweight, LED achieves two additional critical objectives:

1. A **ultra-light key schedule**.
2. **Resistance to related-key and single-key attacks**.

It is worth noting that many lightweight block ciphers are susceptible to **key-related attacks**. For example, the **`HIGHT` cipher** is vulnerable to a known related-key attack (**K+-2010**), while **`Hummingbird-1`** and **`KTANTAN`** have been compromised by practical related-key attacks (**S-2011** and **A-2011**, respectively). In contrast, **LED** demonstrates **resistance to such attacks** despite having an **almost negligible key-scheduling mechanism**.

We provide the reader with a **Github Repository** that contains all the implementations and attacks we have conducted.

2 Cipher Design

The paper presents two variants of the **64-bit LED block cipher**: one supporting a **64-bit key** and the other supporting a **128-bit key**. For simplicity, this discussion is limited to the **64-bit key variant**. **Derived from AES**, the LED cipher operates on a 4×4 state structure. It is a **64-bit block cipher** with two primary instances, each taking **64-bit** and **128-bit keys**. The cipher state is conceptually arranged in a $\mathbf{4 \times 4}$ matrix, where each nibble represents an element from $\mathbf{GF}(\mathbf{2^4})$, with the underlying polynomial for field multiplication given by $X^4 + X + 1$.

LED employs similar basic operations in each round as those in AES. The paper introduces changes that make LED hardware-friendly. We have made it easier to understand the similarities and difference between AES and LED by gathering all points under one heading.

2.1 Similarities with AES

2.1.1 Operations

The operations, **AddConstants**, **SubCells**, **ShiftRows**, and **MixColumnsSerial** (The concept), remain same as AES. The key operations, along with their descriptions, are outlined below:

1. **AddConstants**: In each round, the first step is to **XOR** a round-dependent constant with the first two columns of the state. The round constants are initialized with all zeroes and are then shifted one position to the left. For example, rc_0 is computed as $\mathrm{rc}_5 \oplus \mathrm{rc}_4 \oplus 1$. The 8 bits representing the key size, denoted as $\mathrm{ks}_7, \ldots, \mathrm{ks}_0$, are arranged for the round constants in a specific manner.

$$\begin{pmatrix} 0 \oplus (\mathrm{ks}_7\ \mathrm{ks}_6\ \mathrm{ks}_5\ \mathrm{ks}_4) & (\mathrm{rc}_5\ \mathrm{rc}_4\ \mathrm{rc}_3) & 0 & 0 \\ 1 \oplus (\mathrm{ks}_7\ \mathrm{ks}_6\ \mathrm{ks}_5\ \mathrm{ks}_4) & (\mathrm{rc}_2\ \mathrm{rc}_1\ \mathrm{rc}_0) & 0 & 0 \\ 2 \oplus (\mathrm{ks}_3\ \mathrm{ks}_2\ \mathrm{ks}_1\ \mathrm{ks}_0) & (\mathrm{rc}_5\ \mathrm{rc}_4\ \mathrm{rc}_3) & 0 & 0 \\ 3 \oplus (\mathrm{ks}_3\ \mathrm{ks}_2\ \mathrm{ks}_1\ \mathrm{ks}_0) & (\mathrm{rc}_2\ \mathrm{rc}_1\ \mathrm{rc}_0) & 0 & 0 \end{pmatrix}$$

2. **SubCells**: To achieve confusion, each nibble in the state is substituted using the **LED S-box**, which is identical to the S-box used in **PRESENT**.

X	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

3. **ShiftRows**: This operation involves a permutation achieved by shifting the rows. The nibbles in the state are rotated such that the i^{th} row is shifted by i positions to the left.

4. **MixColumns Serial**: For diffusion, the MixColumns operation is applied, but it differs from AES. In this case, the special construction of the **MDS matrix** is applied to each column independently. This process can be interpreted as applying a **hardware-friendly matrix** A four times to derive the **MDS matrix** M, which does not require additional memory cells for both encryption and decryption from a hardware perspective.

$$(A)^4 = \begin{pmatrix} 0\ 1\ 0\ 0 \\ 0\ 0\ 1\ 0 \\ 0\ 0\ 0\ 1 \\ 4\ 1\ 2\ 2 \end{pmatrix}^4 = \begin{pmatrix} 4\ 1\ 2\ 2 \\ 8\ 6\ 5\ 6 \\ B\ E\ A\ 9 \\ 2\ 2\ F\ B \end{pmatrix} = M$$

2.2 Differences from AES

2.2.1 Introduction of Step

The operation step(`STATE`) consists of four rounds of encryption of cipher state. Each of these four rounds uses, in sequece, the operations, `AddConstants`, `SubCells`, `ShiftRows`, and `MixColumnsSerial`. These operations are explained in detail going forward in this paper.

2.2.2 MDS Matrix

LED uses a hardware friendly MDS matrix that is suitable for serial implementation. The matrix **M** for MixColumnsSerial operation is introduced Above in the paper.

2.2.3 Loading the State

The state is loaded in a **row-wise** manner, rather than in the **column-wise** fashion typically used in AES. This choice is considered more **hardware-friendly**.

2.2.4 Ultra-light Key Schedule

The key schedule is referred to as **"Ultra-light"** because the term **"schedule"** has been omitted from **Key Schedule**. As stated in the paper, *"For the key schedule, we chose to eliminate the 'schedule'; instead, the user-provided key is used repeatedly as is."*

2.3 Pipeline of LED Encryption

In LED, a round consists of the four basic operations: `AddConstants`, `SubCells`, `ShiftRows`, and `MixColumns`. These operations are repeated four times to form a **step**, as explained in the *Difference from AES* section. Notably, the **step** itself does not involve the key. The key is introduced through the `addRoundKey` operation, which **XORs** the key with the state before any step begins. This process is illustrated in the diagram below.

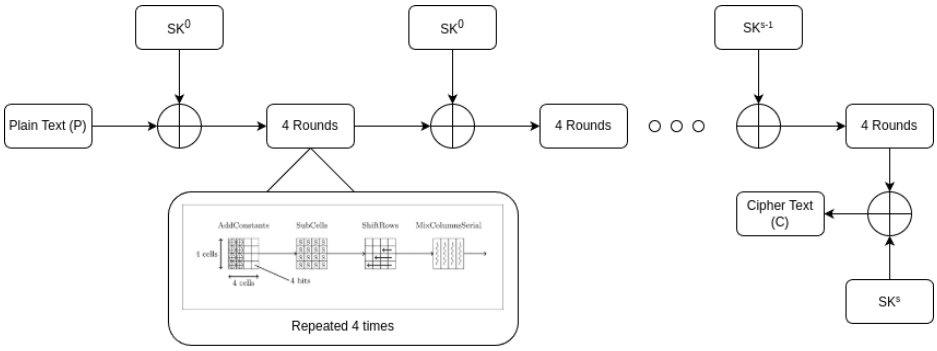

Figure 1: The figure shows the pipeline for the encryption of plaintext using LED Block Cipher

2.4 Pipeline of LED Decryption

We were initially taken aback, when we found that there was no existing implementation for the **decryption function** of LED. Following the lessons from class, we successfully developed the decryption function by incorporating the `InverseMixColumns` operation. The details of the decryption process are provided below.

2.5 Derivation of InverseMixColumn Matrix

In the LED cipher, the `MixColumns` operation is performed by multiplying the state matrix with a pre-defined matrix, known as the MDS matrix. The matrix used in LED for this operation is defined over the finite field $\mathbb{GF}(2^4)$, with the irreducible polynomial $x^4 + x + 1$.

The MDS matrix, M, for the `MixColumns` operation is given by:

$$M = \begin{bmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{bmatrix}$$

In order to reverse the `MixColumns` operation during decryption, we need to compute the inverse of this matrix. The inverse of a matrix M over a finite field is obtained by performing Gaussian elimination or by using a direct formula for matrix inversion in the field $\mathbb{GF}(2^4)$.

Using the MDS matrix M and checking its invertibility, we can compute the inverse matrix, M_{inv}, which is used in the decryption step. If the matrix is invertible, we obtain the inverse matrix M_{inv} as:

$$M_{\text{inv}} = \begin{bmatrix} C & C & D & 4 \\ 3 & 8 & 4 & 5 \\ 7 & 6 & 2 & E \\ D & 9 & 9 & D \end{bmatrix}$$

This inverse matrix M_{inv} is used in the decryption process of LED to reverse the effects of the `MixColumns` operation applied during encryption.

The matrix inversion can be computed using the following Python code, which checks for invertibility and calculates the inverse of the MDS matrix:

```
# SAGEMATH CODE TO DERIVE THE INVERSE MIX COLUMN #
# Define GF(2^4) with the proper irreducible polynomial
F.<a> = GF(2^4, modulus=x^4 + x + 1)

# Define the MDS matrix using field elements directly
M = Matrix(F, [[F.fetch_int(0x4), F.fetch_int(0x1), F.fetch_int(0x2), F.fetch_int(0x2)],
               [F.fetch_int(0x8), F.fetch_int(0x6), F.fetch_int(0x5), F.fetch_int(0x6)],
               [F.fetch_int(0xB), F.fetch_int(0xE), F.fetch_int(0xA), F.fetch_int(0x9)],
               [F.fetch_int(0x2), F.fetch_int(0x2), F.fetch_int(0xF), F.fetch_int(0xB)]])

# Check if the matrix is invertible
if M.is_invertible():
    print("The matrix is invertible.")
    M_inv = M.inverse()
    print("Inverse of the MDS Matrix:")
    print(M_inv)
else:
    print("The matrix is not invertible.")
```

Other functions, **`AddKey`**, **`ShiftRows`** and **`SubCells`** are inverted as done in AES decryption.

3 SBox Analysis

3.1 DDT

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
1	-	-	-	4	-	-	-	4	-	4	-	-	-	4	-	-
2	-	-	-	2	-	4	2	-	-	-	2	-	2	2	2	-
3	-	2	-	2	2	-	4	2	-	-	2	2	-	-	-	-
4	-	-	-	-	-	4	2	2	-	2	2	-	2	-	2	-
5	-	2	-	-	2	-	-	-	-	2	2	2	4	2	-	-
6	-	-	2	-	-	-	2	-	2	-	-	4	2	-	-	4
7	-	4	2	-	-	-	2	-	2	-	-	-	2	-	-	4
8	-	-	-	2	-	-	-	2	-	2	-	4	-	2	-	4
9	-	-	2	-	4	-	2	-	2	-	-	-	2	-	4	-
a	-	-	2	2	-	4	-	-	2	-	2	-	-	2	2	-
b	-	2	-	-	2	-	-	-	4	2	2	2	-	2	-	-
c	-	-	2	-	-	4	-	2	2	2	2	-	-	-	2	-
d	-	2	4	2	2	-	-	2	-	-	2	2	-	-	-	-
e	-	-	2	2	-	-	2	2	2	2	-	-	2	2	-	-
f	-	4	-	-	4	-	-	-	-	-	-	-	-	-	4	4

Maximum Differential Probability for the Sbox $=\frac{1}{4}$.
The Differential Uniformity of the Sbox is 4.
The Differential Branch Number is

3.2 LAT

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	8	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.
1	.	.	.	.	.	-4	.	-4	.	.	.	.	-4	.	4	
2	.	.	2	2	-2	-2	.	.	2	-2	.	4	.	4	-2	2
3	.	.	2	2	2	-2	-4	.	-2	2	-4	.	.	.	-2	-2
4	.	.	-2	2	-2	-2	.	4	-2	-2	.	-4	.	.	-2	2
5	.	.	-2	2	-2	2	.	.	2	2	-4	.	4	.	2	2
6	.	.	.	-4	.	.	-4	.	.	-4	.	.	4	.	.	.
7	.	.	.	4	4	.	.	.	.	-4	.	.	.	.	4	.
8	.	.	2	-2	.	.	-2	2	-2	2	.	.	-2	2	4	4
9	.	4	-2	-2	.	.	2	-2	-2	-2	-4	.	-2	2	.	.
a	.	.	4	.	2	2	2	-2	.	.	.	-4	2	2	-2	2
b	.	-4	.	.	-2	-2	2	-2	-4	.	.	.	2	2	2	-2
c	.	.	.	.	-2	-2	-2	-2	4	.	.	-4	-2	2	2	-2
d	.	4	4	.	-2	-2	2	2	.	.	.	.	2	-2	2	-2
e	.	.	2	2	-4	4	-2	-2	-2	-2	.	.	-2	-2	.	.
f	.	4	-2	2	.	.	-2	-2	-2	2	4	.	2	2	.	.

Maximum Bias for the Sbox is 4

4 Differential Cryptanalysis

To perform differential cryptanalysis, we began by selecting a **maximal differential characteristic** for the S-box used in the LED cipher. The S-box, a 4-bit substitution table, plays a crucial role in ensuring non-linearity in the block cipher. Through analysis, we determined that the **maximal differential probability** of the S-box is 2^{-2}.

4 Round Differential Characteristic of LED Cipher

To maximize the differential characteristic for 4 rounds of the LED cipher, the input difference must be carefully selected Specifically for Round 1, the input difference should be such that, after the MixColumns operation of Round1(inorder to achieve the maximum differential probability), the difference remains confined to a single nibble. This is achieved by the following sequence of operations:

1. Choosing the Desired Difference: We first choose the differential we want to propagate through the cipher after the Round 1 Mixcolumns operation.

2. Applying the Inverse MixColumns Transformation: The chosen difference is transformed back using the inverse MixColumns operation to determine the corresponding state before the MixColumns step. When this IMDS matrix is multiplied with the input difference vector
$$\Delta = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},$$
it yields the output difference vector
$$\Delta' = \begin{bmatrix} C \\ 4 \\ 2 \\ 9 \end{bmatrix}.$$
The operation is performed as follows:
$$\text{IMDS} \cdot \Delta \xrightarrow{\text{Matrix Multiplication}} \Delta',$$
where:
$$\begin{bmatrix} C & C & D & 4 \\ 3 & 8 & 4 & 5 \\ 7 & 6 & 2 & E \\ D & 9 & 9 & D \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$
$$= \begin{bmatrix} C \\ 4 \\ 2 \\ 9 \end{bmatrix}.$$

3. Applying the Inverse ShiftRows Transformation: The resulting state is further transformed using the inverse ShiftRows operation to undo row permutations introduced by the cipher.

4. Applying the Inverse SubBytes Transformation: Finally, the inverse S-box is applied to determine the exact input difference required to prepare the plaintexts.

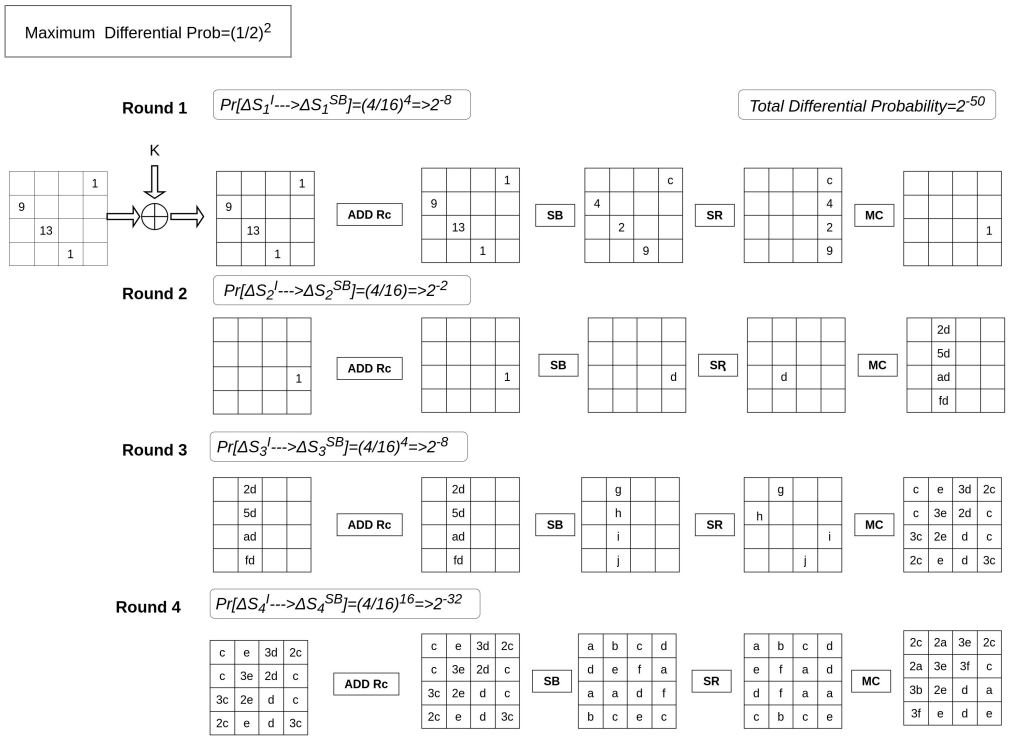

Figure 2: 4 Round differential characteristics for LED Cipher

This sequence ensures that the plaintext differences are optimized to achieve the **maximal differential probability** of for 4 rounds of the LED cipher.

In context of differential cryptanalysis the round function operation can be divided into two categories:

1. Deterministic: ShiftRows, MixColumns, and AddRoundConstant propagate differences deterministically.
2. Probabilistic Operation: SubCells (SubBytes) is a non-linear operation using an S-box, where the propagation of input differences depends on the properties of the S-box. Therefore, the propagation is probabilistic

thus the total differential probability can be calculated as :

- Round 1: $P_1 = 2^{-8}$
- Round 2: $P_2 = 2^{-4}$
- Round 3: $P_3 = 2^{-8}$
- Round 4: $P_4 = 2^{-32}$

Therefore the total Maximal Differential probability=$P_1 * P_2 * P_3 * P_4 = 2^{-50}$

5 Integral Cryptanalysis

The design of the LED cipher is similar to that of the AES, and more specifically the rounds include operations such as AddRoundConstant, ShiftRows, Substitution using the

S-box, and MixColumns. These operations are repeated 4 times for a single step. In summary, the LED cipher consists of 4 rounds, each comprising the following operations:

- AddRoundConstant
- ShiftRows
- Substitution using the S-box
- MixColumns

The claim is that the LED cipher is similar to the AES, and therefore, the properties studied for AES, such as the "All Property," "Constant Property," and "Balanced Property," are expected to exhibit similar behavior in the LED cipher. To observe this closely, let us examine the details.

Consider a set of 16 plaintexts. In the LED cipher, we have 16 nibbles, each representing a 4-bit value. We design a set containing 16 plaintexts such that the first nibble (4 bits) for each plaintext spans the range $[0000, 1111]$, corresponding to $[0, 15]$ or $[0, f]$ in hexadecimal.

Formally, each plaintext p_i in the set can be represented as:

$$\{i\}\_c_1\_c_2\_c_3\_c_4\_c_5\_c_6\_c_7\_c_8\_c_9\_c_{10}\_c_{11}\_c_{12}\_c_{13}\_c_{14}\_c_{15}$$

where $i \in \{0, 1, \ldots, 15\}$ (or $i \in [0, f]$) represents the first nibble, and $c_1, c_2, \ldots, c_{15}$ are constants or fixed values for the remaining nibbles.
Let us formally define the "All Property", "Constant Property" and "Balanced Property":

- **All Property:**
  In the integral cryptanalysis, the byte in which all values appear exactly once among all the texts in the set is called the **All Property**.
- **Constant Property:**
  The byte in which all texts in the set have an identical value is called the **Constant Property**.
- **Balanced Property:**
  The XOR sum of all the texts in the set is 0, that is, the XOR of all 16 plaintexts is equivalent to 0. This property is called the **Balanced Property** and is denoted by B.

Figure 3 represents a sample from the plaintext set that we have taken to complete the integral attack. For the operations like addround constant, shift rows and substitution

0	C_0	C_1	C_2
C_3	C_4	C_5	C_6
C_7	C_8	C_9	C_{10}
C_{11}	C_{12}	C_{13}	C_{14}

(a) Sample plain text 1

F	C_0	C_1	C_2
C_3	C_4	C_5	C_6
C_7	C_8	C_9	C_{10}
C_{11}	C_{12}	C_{13}	C_{14}

(b) Sample plain text 2

Figure 3: TA general representation for plaintext elements

via the s box the property "All property" and "Constant Property" remains the same.

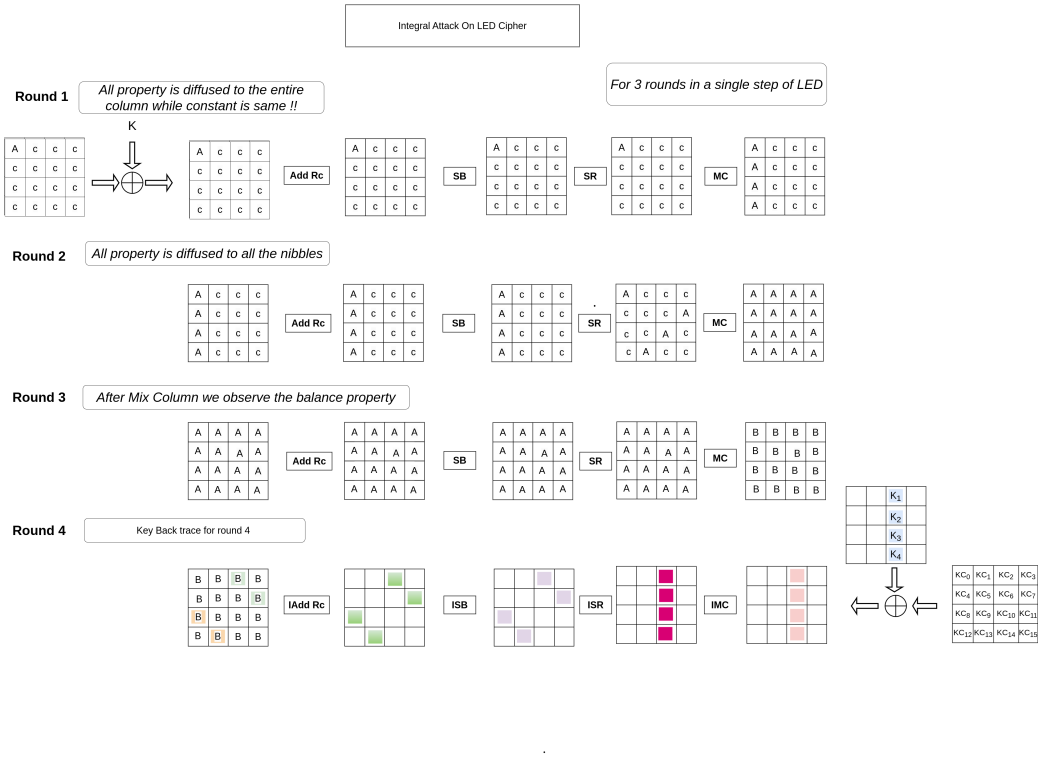

Figure 4: 4 Round key recovery for LED Cipher

However for the operation like mix column the "All property" tends to spread out. The attack incorporates these properties itself and then runs a 3 round integral attack with one round of a key recovery in the last round. This could be better understood as when we guess 4 nibbles in a column or in total of 16 bits then for plaintext set and correspondingly ciphertext set for 1 step could be used to find the right guessed value.
For three rounds we will achieve the balance property and then our task is reduced to check for the four nibbles that we have gussed then XOR with the ciphertext followed by the inverse mix column operation using the inverse mix column opoeration then applying the inverse shift operation and finally the inverse add round constant operation and get a round reduced state, for all the 16 ciphertexts corresponding to our originally taken plaintext set, the XOR sum of all the round reduced state will eventually be 0 as showed by the balance property.
Mathematically, it can be represented as: $\forall i, j, k \in [0, 15], \quad \bigoplus_{i,k=0}^{15} S_{k,i}[j] == 0$

6 Software Implementation

6.1 CLI Discord with RSA Key Exchange and LED Encryption/Decryption

1. **Server and Client Connection**:

    - The **server** is always running, listening for incoming client connections on a specified port.

- The **client** initiates the connection by making a request to the server.

2. **Public Key Exchange**:
   - Once the client and server are connected, they exchange their **public keys**, derived using **RSA** encryption.
3. **Channel List**:
   - The **server** sends a list of available channels to the client.
   - Channels represent different groups for communication, similar to Discord.
4. **Channel Selection**:
   - The **client** selects a channel by sending the **channel name** to the server.
   - The **server** checks if the channel name exists in the database:
     - If not, the server terminates the connection.
     - If the channel exists, the server requests the **password** for the channel.
5. **Password Encryption**:
   - The **client** encrypts the password using the **server's public key** $E_{\text{server}}(\text{password})$ and sends it to the server.
6. **Password Decryption**:
   - The **server** decrypts the received password using its **private key** D_{server} (encrypted_password).
   - If the password is incorrect, the server terminates the connection.
   - If the password is correct, the **server** grants the client access to the channel.
7. **Channel Key Exchange**:
   - The **server** generates a unique **channel key** and encrypts it using the **client's public key** $E_{\text{client}}(\text{channel\_key})$.
   - The **client** receives the encrypted channel key and decrypts it using its **private key** $D_{\text{client}}(\text{encrypted\_key})$, obtaining the key used to encrypt and decrypt messages on that channel.
8. **Message Exchange**:
   - The **client** encrypts the message using the **channel key** $E_{\text{channel\_key}}$ (message) before sending it to the server.
   - The **server** receives the encrypted message and forwards it to all clients in the channel.
   - Each client decrypts the message using the same **channel key** $D_{\text{channel\_key}}$ (encrypted_message) to read it.
9. **Encryption Method**:
   - We use the **ECB mode** of encryption with the **LED cipher** to encrypt the text messages.

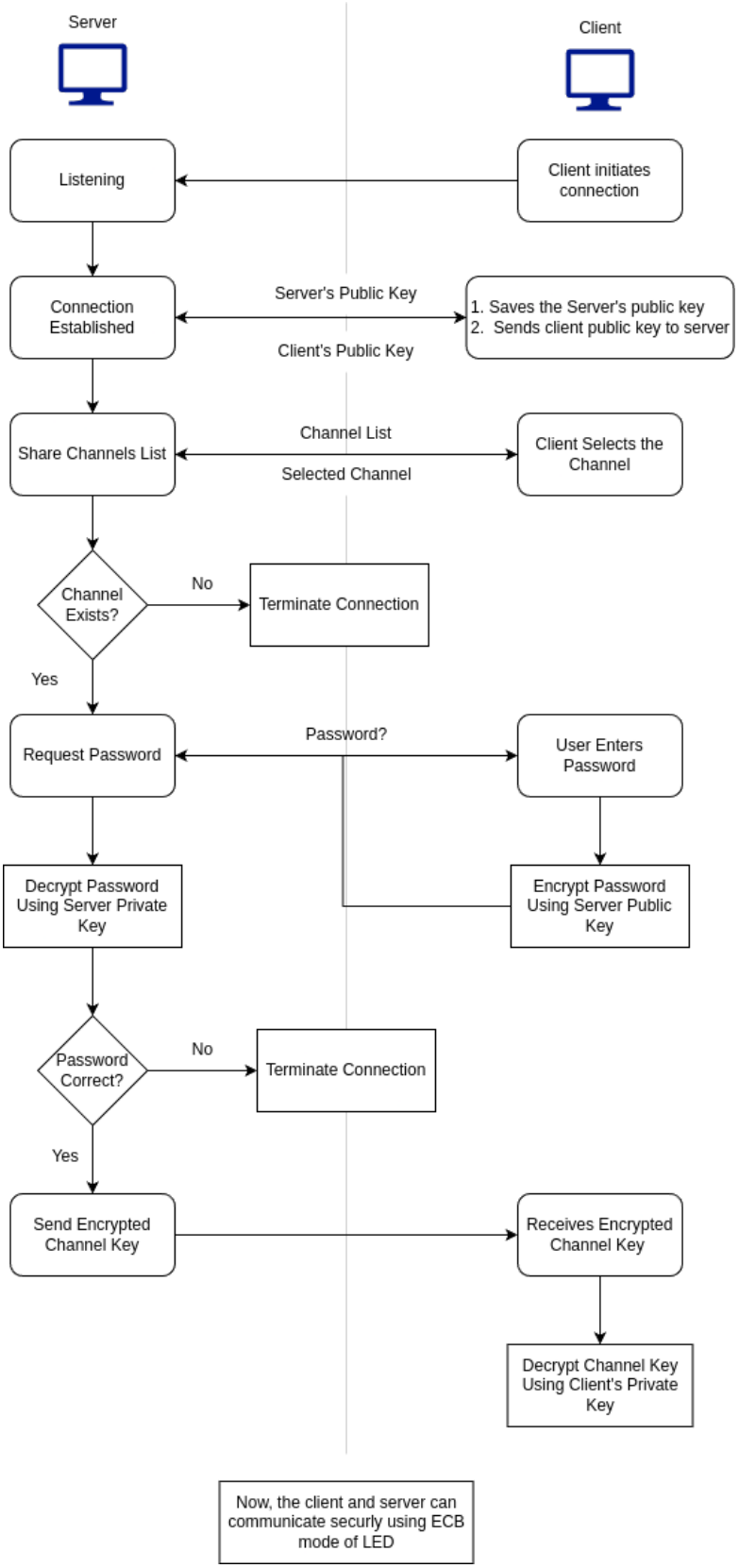

Figure 5: Workflow of Software Implementation

6.2 Hash Construction

The paper discusses the application of LED in a hash function setting. We implemented a hash function based on the LED encryption function using the **`Davies-Mayer`** construction. Additionally, we attempted to run **`Pre-image Attack`**, **`Second Pre-image Attack`**, and **`Collision Detection Attack`**. However, these attacks were unsuccessful due to the high-order complexity involved. The code for the hash construction and attacks can be found in the github repository.