Correlation Power Analysis (CPA) on Romulus-N Encryption
Team Name: Mini Militia

Deepak S (MM22B011) Monish M (CS23M003)

October 13, 2024

1 Introduction

In this analysis, we perform a Correlation Power Analysis (CPA) attack on the Romulus-N encryption algorithm, which is based on the SKINNY block cipher. The CPA attack is conducted using a given set of 500 plaintexts and their respective power traces to deduce the key used in encryption. The attack focuses primarily on the first round of the encryption process, with the aim of recovering the key bytes K_0 to K_7, and then it extends to attack into the next round deriving the next set of key bytes from K_8 to K_{15}.

1.1 Encryption Overview

Romulus-N is a lightweight authenticated encryption scheme that follows the NIST LWC standard and uses the SKINNY block cipher for encryption. The inputs to the encryption process include:

- Message M: 16-byte blocks
- Nonce N: 16-byte blocks
- Associative Data (AD): Fixed as 00
- Key K: 16-byte block

The encryption consists of 80 rounds of SKINNY encryption, divided into two phases:

1. The first 40 rounds operate with AD = 00, Nonce N, and Key K.
2. The subsequent 40 rounds use the message M, Nonce N, and Key K.

The Romulus encryption uses Skinny implementation for its encryption. The Skinny implementation takes 48 bytes of key named as **Tweak Key** for encryption. The Tweak Key can be generated from the actual key using predefined operations on the key and nonce. Each round of the Skinny implemented encryption has the following operations:

- **SubCells** – This operation converts a given byte of data to its substitute value based on a Look-up Table (similar to the S-box in AES).

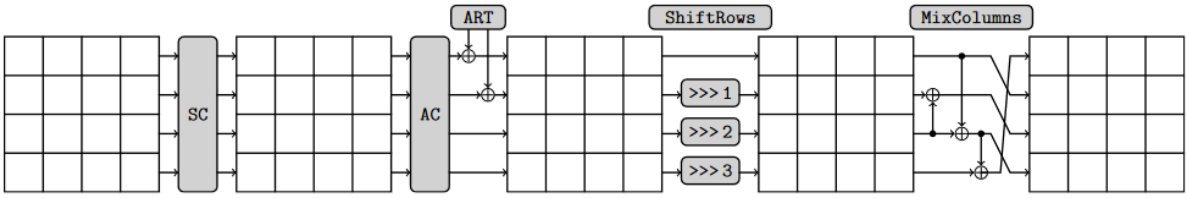

Figure 1: One round encryption of Skinny Implementation in Rombulus

- **AddConstants** – Each byte in the state undergoes bitwise XOR with a predefined matrix called the **constant matrix**. The contents of the constant matrix are fixed for a given round.
- **AddRoundTweakey (ART)** – The Tweak Key for a given round is a 48-byte data with the following structure:
  - The first 16 bytes are **TK1**.
  - The next 16 bytes are the **Nonce** of the plaintext, denoted as **TK2**.
  - The last 16 bytes are the **Derived Key**, denoted as **TK3**.

  For the first round, the Derived Key is the same as the actual key. For subsequent rounds, **TK3** is a combination of a permuted version of the previous round's TK3 and the result of applying an LFSR (Linear Feedback Shift Register) on the previous round's TK3.
- **ShiftRows** – This operation shifts the state matrix in the same manner as AES, but to the right.
- **MixColumns** – The result of the **ShiftRows** operation is further diffused by **MixColumns**, similar to AES but with a different matrix.

The **AddRoundTweakey** operation mentioned above uses only the first two rows of the Tweak Key. This essentially means that the first 8 bytes of **TK1**, **TK2**, and **TK3** will only be used for encryption in the current round. The next 8 bytes (byte 8 to byte 15) will be used in the next round (Round 2).

As a result, the actual key (**K**) bytes K_0, K_1, K_2, K_3, K_4, K_5, K_6, and K_7 will be used in Round 1 operations, while the key bytes K_8, K_9, K_{10}, K_{11}, K_{12}, K_{13}, K_{14}, and K_{15} will be used in Round 2.

We focus on recovering key bytes in the initial rounds, utilizing the CPA attack on the ART (AddRoundTweakey) operation, which reveals correlations between the key and power traces.

2 Methodology

As mentioned above, we intend to do a CPA attack in initial rounds, preferably 1st round of encryption. This would essentially help us in obtaining only key bytes K_0 to K_7 as key bytes K_8 to K_{15} are not being used in Round 1. To obtain key bytes K_8 to K_{15}, we need to proceed to Round 2 implementation as well. Thus, our attack is spread across Round 1 and Round 2.

2.1 Correlation Power Analysis (CPA)

CPA is a powerful side-channel attack that leverages power consumption data during encryption. The basic premise is that specific operations in encryption, such as XOR or S-Box lookups, result in measurable power consumption that correlates with specific key guesses. The CPA attack works by comparing the Hamming weight of intermediate encryption values (derived from key guesses) with the actual power traces.

The attack is divided into two parts:

- **Round 1 Analysis**: We attempt to extract key bytes K_0 to K_7 by correlating the power traces with Hamming weights from the ART output of the first round.

  The attack in Round 1 is carried out to find the key bytes K_0 to K_7. Accordingly, encryption operations on only the first two rows (first 8 bytes) are sufficient. For the attack in Round 1, we need to correlate the output state of the AddRoundTweakey (ART) operation (as this state contains key byte elements for guessing) of Round 1. The following round operations are carried out:

  1. **SubCells** – Each byte input to SubCells would be 00 (as given, $AD = 00$). Perform this operation for the first 8 bytes and store the result as the state.
  2. **AddConstants** – Each byte in the state after SubCells operations is XORed with a predefined constant matrix:

  $$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

  The result is stored as the state and will be input to the next round operation.
  3. **AddRoundTweakey** – The available Python implementation of encryption was used to find the tweak keys (**TK1**, **TK2**, and **TK3**) for Round 1, which are as follows:
     - **TK1**:

     $$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 26 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

     - **TK2**: Nonce extracted from the provided plaintext `.npy` file.
     - **TK3**: 16 bytes of keys to be guessed.

(a) SubCells

Each byte input to SubCells is set to 00 (as given, $AD = 00$). This operation is performed for the first 8 bytes, and the result is stored as the state.

(b) AddConstants

Each byte in the state after SubCells is XORed with a predefined constant matrix:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The result is stored as the state and will be the input to the next round operation.

(c) AddRoundTweakey

The available Python implementation of encryption was used to find the tweak keys (**TK1**, **TK2**, and **TK3**) for Round 1:

- **TK1**:

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 26 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- **TK2**: Nonce extracted from the provided plaintext `.npy` file.
- **TK3**: 16 bytes of keys to be guessed.

(d) Divide and Conquer

Although Romulus encryption happens in blocks of 16 bytes, round operations are carried out byte-wise in hardware. Thus, the key guess can be performed using a divide and conquer approach, by guessing each byte of the key individually. Each key byte guess (K_0 to K_7) has 256 possible values. For each key guess, the hamming weight of each byte from the ART output is calculated.

(e) Correlation

The hamming weight of the ART output with each key guess is correlated with the provided power traces to find the correct key guess. The key guess with the highest correlation value is likely to be the correct one. However, due to possible false positives, the correct key could be any guess with a good correlation value, not necessarily the best.

(f) Discarding False Positives

False positives can be reduced by narrowing the search space for correlation. To further reduce false positives, patterns in the key guesses must be identified. Two consecutive correct key guesses will typically have a consistent pattern. For example, correct guesses for K_0, K_1, K_2, etc., follow a constant positional difference:

$$K_0 \longrightarrow K_1 \longrightarrow K_2 \quad \text{(constant positional difference)}$$

In this case, the constant positional difference for the first 4 key bytes was found to be 67, indicating that the first correct key byte guess occurred around point 4000. The search space was thus reduced, starting at point 4000 for the first byte, 4000 + 67 for the second byte, and so on until the 8th byte, which significantly reduced the false positives.

Round 2 Analysis

This part involves extracting K_8 to K_{15}, using the same approach but applied to the second round of encryption after performing ShiftRows and MixColumns. The output of Round 1 was calculated with RowShift and MixColumns operations. The resulting state would be used as input for the Round 2 operations. To obtain key bytes K_8 to K_{15}, a similar approach to that used for K_0 to K_7 was adopted. However, the following changes with respect to Round 1 were made:

1. **Tweak Key for Round 2** – The Tweak Keys (**TK1**, **TK2**, and **TK3**) were obtained from the Python implementation.
2. **Guessing LFSR Value** – Instead of guessing the key, we need to guess the LFSR value (ranging from 0 to 255) that would yield the maximum/correct correlation. The valid LFSR value is then mapped to the correct key using the inverse LFSR lookup table.

2.2 Implementation Details

The attack was implemented in Python, utilizing numpy and specific power traces loaded from .npy files. The core functionality is provided by the following code snippet, which reads the correlation points from the power traces and identifies the top key guesses for each byte index:

```
for key_guess in range(256):
    for point in range(search_start, search_end):
        correlation_value = correlation_array[key_guess, point]
            results.append((byte_index, key_guess, point, correlation_value))

        if correlation_value > max_correlation:
            max_correlation = correlation_value
            max_point = point
            max_key_guess = key_guess
```

This script collects the correlation points for bytes 0 through 3, and calculates the differences between the key guesses at various stages of the encryption process.

3 Results

3.1 Key Guess Correlations

The CPA attack revealed the following key guess correlations:

- Differences between Byte 0 and Byte 1
- Differences between Byte 1 and Byte 2
- Differences between Byte 2 and Byte 3

The goal was to identify the points where the differences in key guesses were consistent across byte pairs, such that the differences were almost constant. This was done by comparing the differences for consecutive byte indices and searching for common patterns.

4 Conclusion

The CPA attack on Romulus-N encryption successfully identified key byte correlations using power traces from the first round of encryption. By filtering the results and analyzing the differences between consecutive key guesses, we were able to recover the key bytes K_0 to K_7 effectively. The retrieved key is:

key = `"93 4d 67 9d 1f 0a 5c 62 df 0f 00 a7 0b e8 ae 86"`

5 References

- Romulus-N GitHub repository: https://github.com/romulusae/romulus
- Points document on CPA results (attached).