SE518 — Lab 2 Report

Security Lab – Introduction to Fault Injection and Differential Fault Analysis

SE518 — Embedded Systems Security
Supervisor: Prof. MIRBAHA Amir-Pasha

Group Members

- HAMZA Muhammed
- IBRAHIM Thomas
- DA ROZA Lukas

Q4.1

Based on the provided information, which round do you think is eliminated as a result of the attack?

> **Round 10 is eliminated by the attack**

Q4.2

Determine the total number of possible hypotheses for a single byte of the K10 key without using the given equation.

> **256 hypotheses (2^8) per byte**, since each byte can take any value from `0x00` to `0xFF`

Q4.3

Develop a Python function to identify the correct hypothesis for each byte of K10.
Your function should rely solely on the final provided equation and must not utilize Plaintexts.

```python
def find_k10_candidates(C1, C2, D1, D2):
    reverse_1 = compute_reverse(D1, D2)
    candidates = []
    for i in range(16):
        candidates.append([])  # empty list

    for pos in range(16):
        candidates[pos] = []
        for k in range(256):
            # C1: correct one
            state_1 = C1.copy()
            state_1[pos] ^= k
            state_1 = ArrayToMatrix(state_1)
            state_1 = InvShiftRows(state_1)
            state_1 = InvSubBytes(state_1)
            out_1 = MatrixToArray(state_1)

            # C2: with fault
            state_2 = C2.copy()
            state_2[pos] ^= k
            state_2 = ArrayToMatrix(state_2)
            state_2 = InvShiftRows(state_2)
            state_2 = InvSubBytes(state_2)
            out_2 = MatrixToArray(state_2)

            shifted_pos = pos_after_shiftrows(pos)
            xor = out_1[shifted_pos] ^ out_2[shifted_pos]
            if xor == reverse_1[shifted_pos]:
                candidates[pos].append(k)
    return candidates
```

Q4.4

Evaluate whether your function accurately identifies the correct hypothesis for each byte of K10.

> **The function is partially successful but not entirely successful.**

Reasoning:

- 15 out of 16 bytes (**93.75%**) are correctly identified with **1 candidate each**
- 1 out of 16 bytes (**byte 10**) still has **2 candidates**, so we cannot determine the correct hypothesis for this byte

If the function is not entirely successful, calculate the number of potential hypotheses using the specified equation.

> **Answer:** The total number of potential hypotheses for the full K10 key is **2**.

Q4.5

Explain how plaintexts could enhance your analysis?

Plaintexts could enhance the analysis by:

- **Verification**: Test candidate keys by encrypting known plaintexts and comparing with correct ciphertexts
- **Additional equations**: Create more DFA equations using different plaintext-ciphertext pairs
- **Elimination**: Quickly eliminate incorrect key candidates through encryption testing

---

Q4.6

Analyzing Corresponding Ciphertexts

Describe the process of incorporating the third set of correct and faulty ciphertexts into your analysis.

The function find_k10_candidates(C1, C2, C3, D1, D2, D3) performs the DFA attack by exploiting **three single-byte fault injections** into the same correct encryption, all occurring **just before the final MixColumns** in the encryption direction (i.e., between **SubBytes and MixColumns in round 10**).

The **third set** of correct and faulty ciphertexts (C1, D3) is incorporated in a **second filtering stage**:

1. After reducing key candidates per byte to **~2 bytes** using the fault pair (D1, D2),
2. The function tests the **top 2 candidates** against the **differential condition derived from (D1, D3)**.
3. Only key bytes that satisfy **both fault-induced differential equations** are retained.

This step **eliminates all but the correct K10 byte** in each position, enabling **full recovery of the last-round key with just three single-byte faults**.

```
def find_k10_candidates(C1, C2, C3, D1, D2, D3):
    reverse_1 = compute_reverse(D1, D2)
    reverse_2 = compute_reverse(D1, D3)

    candidates = []
    key = []
    for i in range(16):
        candidates.append([])  # empty list
        key.append([])

    # First pass: filter using (D1, D2)
    for pos in range(16):
        candidates[pos] = []
        for k in range(256):
            # C1: correct
            state_1 = C1.copy()
            state_1[pos] ^= k
            state_1 = ArrayToMatrix(state_1)
            state_1 = InvShiftRows(state_1)
            state_1 = InvSubBytes(state_1)
            out_1 = MatrixToArray(state_1)

            # C2: faulty
            state_2 = C2.copy()
            state_2[pos] ^= k
            state_2 = ArrayToMatrix(state_2)
            state_2 = InvShiftRows(state_2)
            state_2 = InvSubBytes(state_2)
            out_2 = MatrixToArray(state_2)

            shifted_pos = pos_after_shiftrows(pos)
            xor = out_1[shifted_pos] ^ out_2[shifted_pos]
            if xor == reverse_1[shifted_pos]:
                candidates[pos].append(k)

    # Second pass: filter top candidates using (D1, D3)
    for pos in range(16):
        key[pos] = []
        for k_idx in range(min(2, len(candidates[pos]))):
            k = candidates[pos][k_idx]
```

```
            # C1: correct
            state_1 = C1.copy()
            state_1[pos] ^= k
            state_1 = ArrayToMatrix(state_1)
            state_1 = InvShiftRows(state_1)
            state_1 = InvSubBytes(state_1)
            out_1 = MatrixToArray(state_1)

            # C3: faulty
            state_3 = C3.copy()
            state_3[pos] ^= k
            state_3 = ArrayToMatrix(state_3)
            state_3 = InvShiftRows(state_3)
            state_3 = InvSubBytes(state_3)
            out_3 = MatrixToArray(state_3)

            shifted_pos = pos_after_shiftrows(pos)
            xor = out_1[shifted_pos] ^ out_3[shifted_pos]
            if xor == reverse_2[shifted_pos]:
                key[pos].append(k)

    return key
```

Can you derive a new equation similar to the provided one, using Ca, Cc, Da and Dc?

> **Yes** — a new differential equation can be derived analogously:
>
> InvSubBytes(Ca ⊕ k) ⊕ InvSubBytes(Dc ⊕ k) = compute_reverse(Da, Dc)
>
> This follows the same fault model and inverse transformation path.

Q4.7

What is the total number of hypotheses for the full K10 key after utilizing the third pair of corresponding correct and faulty ciphertexts?

> After intersecting candidates from **both equation pairs**, you should get **exactly 1 candidate per byte**, so:
>
> **Total hypotheses = 1**

Q4.8

What are the possible approaches to identify the correct hypothesis for the full K10 key?

The possible approaches are as follows:

- **Intersection of candidates from multiple fault pairs**
- **Encryption testing with known plaintext-ciphertext pairs**
- **Statistical analysis of candidate keys**

Q4.9

How can this be achieved? What is the resulting outcome?

> The **correct K10 key** should be **uniquely identified**, allowing **recovery of the main AES key** through **inverse key expansion**.

Task 5

Practical Countermeasures Against EMFI Attacks

Temporal Redundancy

- **Execute cryptographic operations twice and compare results**
  → *If results differ, trigger error response*

Spatial Redundancy

- **Use dual-rail logic with complementary signals**
  → *Detects asymmetric faults*
- **Implement triple modular redundancy for critical operations**
  → *Majority voting masks faults*

Detection Circuits

- **Embed EM fault sensors on the chip**
  → *Real-time fault detection*
- **Monitor power supply glitches and clock irregularities**
  → *Triggers alarm on anomaly*

Protocol Level Protections

- **Add integrity checks (MAC) to cryptographic outputs**
  → *Detects corrupted outputs*

- **Implement session keys that change frequently**
  → *Limits impact of key exposure*

Hardening Techniques

- **Randomize execution order of operations**
  → *Increases fault injection difficulty*