# Assignment 3

*CS6630: Secure Processor Microarchitecture*

Arjun Menon V, Akilesh Kannan
EE18B104, EE18B122

## 1 Single-Fault Attack on AES-128

The attack implemented here is adapted from this work [1].

### 1.1 Approach

In order to break the secret key for AES, we only need to get one of the round keys. In this case, we can easily get the last round key through some simple analysis when a fault is introduced in the $8$th, before the $\mathrm{MixColumns}$ step.

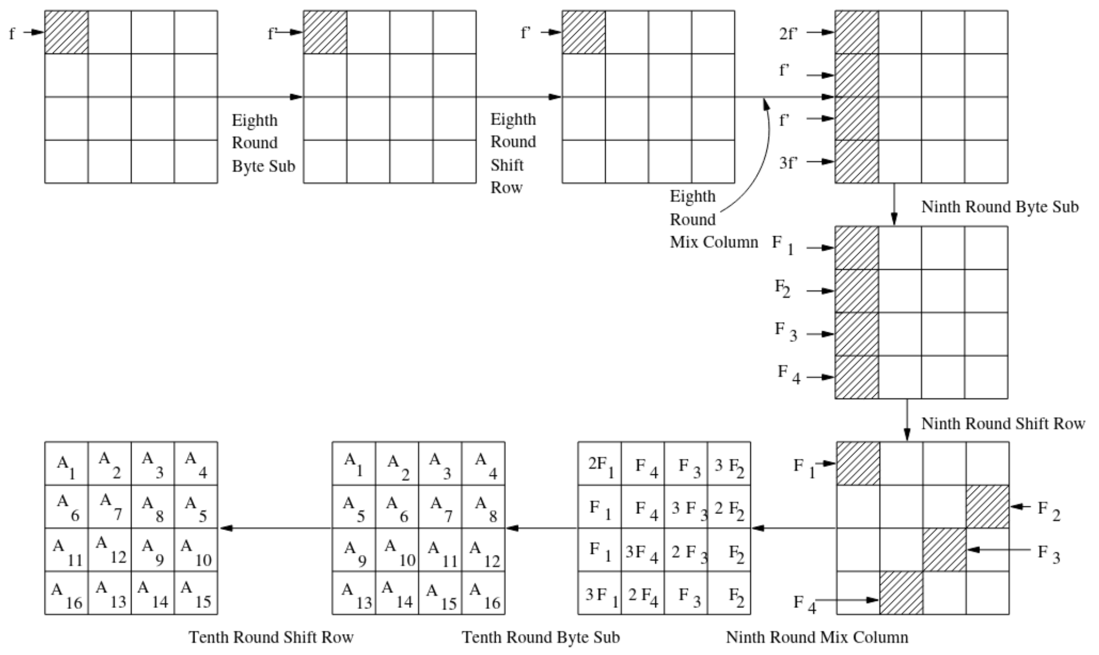

Fig. 1: Propagation of $8$th round fault [1]

Let us assume that the fault in $8$th round is introduced at position 0. We can then trace the propagation of the fault through the rounds as shown in Fig.1.

At the end of $9$th round (after $\mathrm{MixColumns}$), we arrive at the following sets of equations, each involving 4 bytes of the round 10 key $K_{10}$:

$$2F_1 = S^{-1}(x_0 \oplus k_0) \oplus S^{-1}(x'_0 \oplus k_0)$$
$$F_1 = S^{-1}(x_{13} \oplus k_{13}) \oplus S^{-1}(x'_{13} \oplus k_{13})$$
$$F_1 = S^{-1}(x_{10} \oplus k_{10}) \oplus S^{-1}(x'_{10} \oplus k_{10})$$
$$3F_1 = S^{-1}(x_7 \oplus k_7) \oplus S^{-1}(x'_7 \oplus k_7)$$

$$\begin{aligned}2F_4 &= S^{-1}(x_{11} \oplus k_{11}) \oplus S^{-1}(x'_{11} \oplus k_{11})\\ F_4 &= S^{-1}(x_4 \oplus k_4) \oplus S^{-1}(x'_4 \oplus k_4)\\ F_4 &= S^{-1}(x_1 \oplus k_1) \oplus S^{-1}(x'_1 \oplus k_1)\\ 3F_4 &= S^{-1}(x_{14} \oplus k_{14}) \oplus S^{-1}(x'_{14} \oplus k_{14})\end{aligned}$$

$$\begin{aligned}2F_3 &= S^{-1}(x_2 \oplus k_2) \oplus S^{-1}(x'_2 \oplus k_2)\\ F_3 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8)\\ F_3 &= S^{-1}(x_{15} \oplus k_{15}) \oplus S^{-1}(x'_{15} \oplus k_{15})\\ 3F_3 &= S^{-1}(x_8 \oplus k_8) \oplus S^{-1}(x'_8 \oplus k_8)\end{aligned}$$

$$\begin{aligned}2F_2 &= S^{-1}(x_9 \oplus k_9) \oplus S^{-1}(x'_9 \oplus k_9)\\ F_2 &= S^{-1}(x_3 \oplus k_3) \oplus S^{-1}(x'_3 \oplus k_3)\\ F_2 &= S^{-1}(x_6 \oplus k_6) \oplus S^{-1}(x'_6 \oplus k_6)\\ 3F_2 &= S^{-1}(x_{12} \oplus k_{12}) \oplus S^{-1}(x'_{12} \oplus k_{12})\end{aligned}$$

where, $S()$ denotes the SBOX operation, $S^{-1}()$ denotes it's inverse operation, $x_i$ denotes the $i^{\text{th}}$ byte of the correct ciphertext and $x'_i$ denotes the same in the faulty ciphertext. The byte ordering followed is column-major.

As shown in [1], we get $\sim 256$ solutions for each of the 4 sets of equations, thus resulting in $2^{32}$ possible solutions for $K_{10}$. Since, we do not know the position of the induced fault, we repeat this analysis assuming the fault is induced independently at each of the 16 bytes, resulting in $\sim 4096$ solutions for each of the 4-byte subkeys and a total of $2^{36}$ possible $K_{10}$ values.

While [1] proposes a $2^{\text{nd}}$ step in the attack, exploiting the relationship between $9^{\text{th}}$ and $10^{\text{th}}$ round keys to narrow this search space down, we have decided to take a different approach, making use of multiple CT-FCT pairs.

We repeat the above experiment on another CT-FCT pair, and find common solutions for each of the 4-byte subkeys with the previous pair.

*Note that we do not search over the entire $2^{36}$ possibilities for $K_{10}$, but rather only search among the $\sim 4096$ solutions for each of the 4-byte subkeys* - This allows us to find $K_{10}$ rather quickly by piecing together the 4 partial subkeys.

After obtaining $K_{10}$, we can get the secret key and other round keys by backtracking through the $\text{KeySchedule}$ algorithm.

## 1.2 Results

| Parameter | Value | |
|---|---|---|
| Retrieval Time | 2min 3sec | |
| No. of CT-CT' pairs | 2 | |
| Secret Key | [107 142 167 187 226 167 118 199 34 185 173 84 32 102 14 150] | |
| Round Keys | Round 1 | [ 88 37 55 12 136 152 236 57 230 255 99 70 174 112 245 204] |
| | Round 2 | [ 9 195 124 232 137 182 252 162 65 177 211 124 45 184 147 220] |
| | Round 3 | [101 31 250 48 196 118 209 166 93 137 237 88 97 31 198 182] |
| | Round 4 | [165 171 180 223 194 20 92 56 120 115 167 95 221 144 154 121] |
| | Round 5 | [197 19 2 30 100 105 43 74 59 138 86 137 63 238 43 222] |
| | Round 6 | [237 226 31 107 49 241 235 53 252 43 191 31 143 31 35 30] |
| | Round 7 | [ 45 196 109 24 233 237 215 152 226 126 177 89 23 236 235 213] |
| | Round 8 | [227 45 110 232 248 53 72 3 163 232 227 34 29 119 250 70] |
| | Round 9 | [ 22 0 52 76 191 86 80 42 171 89 176 199 127 188 29 128] |
| | Round 10 | [115 164 249 158 48 31 201 33 175 153 109 58 6 82 33 0] |

Tab. 1: Results

## 1.3 Code

**Password cracker script**

```python
import numpy as np

ct = np.load("ciphertexts.npy")
ct_ = np.load("faultytexts.npy")

s = np.load("sbox.npy")
s_ = np.load("sbox_inv.npy")

mult = np.load("multiplies.npy")
rcon = np.load("rcon.npy")

def sbox(byte):
    return s[byte]

def sbox_inv(byte):
    return s_[byte]

# Key reversal from round 10 key
def reverseKey(key10):
    subKeys = np.zeros(176, dtype= np.uint8)
    for i in range(160, 176):
        subKeys[i] = key10[i - 160]
    for i in range(156, -1, -4):
        if i % 16 == 0:
            subKeys[i] = subKeys[i + 16] ^ sbox(subKeys[i+13]) ^ rcon[i >> 4]
            subKeys[i+1] = subKeys[i+17] ^ sbox(subKeys[i+14])
            subKeys[i+2] = subKeys[i+18] ^ sbox(subKeys[i+15])
            subKeys[i+3] = subKeys[i+19] ^ sbox(subKeys[i+12])
        else:
            subKeys[i] = subKeys[i + 16] ^ sbox(subKeys[i+12])
            subKeys[i+1] = subKeys[i+17] ^ sbox(subKeys[i+13])
            subKeys[i+2] = subKeys[i+18] ^ sbox(subKeys[i+14])
            subKeys[i+3] = subKeys[i+19] ^ sbox(subKeys[i+15])
    return subKeys
```

```python
# Get the column affected after ShiftRows
def get_fault_column(position):
    if position == 0 or position == 5 or position == 10 or position == 15:
        return 0
    elif position == 4 or position == 9 or position == 14 or position == 3:
        return 1
    elif position == 8 or position == 13 or position == 2 or position == 7:
        return 2
    elif position == 12 or position == 1 or position == 6 or position == 11:
        return 3
    else:
        return None

# Get the factors for delta in each of the 4 sets of 4 equations
def get_factors(column):
    if column == 0:
        return [[2, 1, 1, 3],
                [1, 1, 3, 2],
                [1, 3, 2, 1],
                [3, 2, 1, 1]]
    elif column == 1:
        return [[3, 2, 1, 1],
                [2, 1, 1, 3],
                [1, 1, 3, 2],
                [1, 3, 2, 1]]
    elif column == 2:
        return [[1, 3, 2, 1],
                [3, 2, 1, 1],
                [2, 1, 1, 3],
                [1, 1, 3, 2]]
    elif column == 3:
        return [[1, 1, 3, 2],
                [1, 3, 2, 1],
                [3, 2, 1, 1],
                [2, 1, 1, 3]]
    else:
        return None

# Solve 1 set of equations (4 key bytes)
def solve(correct, faulty, byte_positions, factors):
    possibilities = []
    # Assume certain delta
    for delta in range(256):
        b_sols = []

        # Solve for each of the 4 bytes
        for i in range(4):
            bi_sols = []
            lhs = mult[factors[i],delta]
            for kb in range(256):
                rhs = sbox_inv(correct[byte_positions[i]] ^ kb) ^
                ↪ sbox_inv(faulty[byte_positions[i]] ^ kb)
                if lhs == rhs:
                    bi_sols.append(kb)
            b_sols.append(bi_sols)

        # If at least one equation not satisfied for given delta, discard
        if (len(b_sols[0]) == 0) or (len(b_sols[1]) == 0) or (len(b_sols[2]) ==
        ↪ 0) or (len(b_sols[3]) == 0):
            continue
        # Else, add solutions to possibilities
        else:
            for b0 in b_sols[0]:
                    for b1 in b_sols[1]:
                        for b2 in b_sols[2]:
                            for b3 in b_sols[3]:
                                keybytes = (b0, b1, b2, b3)
                                possibilities.append(keybytes)
    return possibilities

if __name__ == '__main__':
    # 4 subparts of all inputs and fault positions
    keys_p0 = []
    keys_p1 = []
    keys_p2 = []
    keys_p3 = []
    # solve for 2 pairs of CT-CT'
    for pair_num in range(2):
        # different 4-byte subkeys of K10
        keys_0_7_10_13 = []
        keys_1_4_11_14 = []
        keys_2_5_8_15 = []
        keys_3_6_9_12 = []
        # assume fault is at each position (since we don't know where)
        for position in range(16):
            col_num = get_fault_column(position)
            assert(col_num!=None)
            factors = get_factors(col_num)
            assert(factors!=None)
            keys_0_7_10_13.extend(solve(ct[pair_num], ct_[pair_num], [0, 13,
             ↪  10, 7], factors[0]))
            keys_1_4_11_14.extend(solve(ct[pair_num], ct_[pair_num], [4, 1, 14,
             ↪  11], factors[1]))
            keys_2_5_8_15.extend(solve(ct[pair_num], ct_[pair_num], [8, 5, 2,
             ↪  15], factors[2]))
            keys_3_6_9_12.extend(solve(ct[pair_num], ct_[pair_num], [12, 9, 6,
             ↪  3], factors[3]))
        keys_p0.append(keys_0_7_10_13)
        keys_p1.append(keys_1_4_11_14)
        keys_p2.append(keys_2_5_8_15)
        keys_p3.append(keys_3_6_9_12)
    # Get common subkeys among all
    final_set_0 = list(set(keys_p0[0]) & set(keys_p0[1]))
    final_set_1 = list(set(keys_p1[0]) & set(keys_p1[1]))
    final_set_2 = list(set(keys_p2[0]) & set(keys_p2[1]))
    final_set_3 = list(set(keys_p3[0]) & set(keys_p3[1]))

    # Piece together 4 subkeys to get K10
    key10 = [0] * 16
    indexGroups = [[0, 13, 10, 7], [4, 1, 14, 11], [8, 5, 2, 15], [12, 9, 6, 3]]
    for i in range(0, 4):
        key10[indexGroups[0][i]] = final_set_0[0][i]
        key10[indexGroups[1][i]] = final_set_1[0][i]
        key10[indexGroups[2][i]] = final_set_2[0][i]
        key10[indexGroups[3][i]] = final_set_3[0][i]

    # Key reversal
    np.save("key10", key10)
    allKeys = reverseKey(key10)
    allKey_dict = {}
    print("Secret Key: ", allKeys[0:16])
    for i in range(1, 11, 1):
        print("Round {}:".format(i), allKeys[16*i : 16*(i+1)])
    np.save("allKeys", allKeys)
```

## References

[1] Tunstall, M., Mukhopadhyay, D., and Ali, S. Differential fault analysis of the advanced encryption standard using a single fault. In *Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication* (Berlin, Heidelberg, 2011), C. A. Ardagna and J. Zhou, Eds., Springer Berlin Heidelberg, pp. 224–233.