



This work is distributed  
under the CC BY-SA license,  
unless stated otherwise.

# HardwareSecurity

## **12. Physical side-channel attack defences 2 (active)**

Ján Jančár

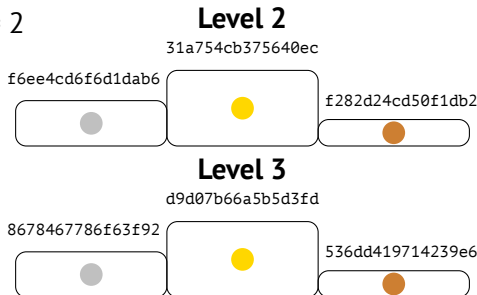
Summer 2026

# Today

- Countermeasure principles
- Attacks & countermeasures

# Assignment 2

- Q1: Operation leakage in level 1, Data leakage in levels 2 and 3, Hamming weight leakage
- Q2: Normal distribution, zero mean, sdev = 2
- Level 1: Simple SPA was enough
  - Single trace
- Level 2: Generalized DPA required
  - Generalized DPA: ~ 100 traces
  - CPA: ~ 30 traces
  - Profiled attack: 1 trace?!?
- Level 3: RPA/ZVP required
  - ~ 2 traces per bit, maybe 1 possible
  - Some horizontal collision attack may also work
- Top 3 (level 2 and 3): Share your solutions with your classmates in the discussion forum if you want



# Fault attack

## Kill chain

### Cyber kill chain

- 1 Fault injection
  - Attacker manipulates the environment and creates a fault
  - EM pulse, voltage glitch, ...
- 2 Fault manifestation
  - Fault manifests as a logical effect in target execution
  - Bit flip, byte reset, ...
- 3 Fault propagation
  - Fault propagates and impacts execution further
  - Wrong branch taken, ...
- 4 Fault observation
  - Attacker is able to observe the result of the fault somehow
  - Wrong ciphertext output, wrong signature output, ...
- 5 Fault exploitation
  - Attacker exploits the fault to break some security property
  - Key recovery, verification bypass, ...

Stopping the attack at **any** step is sufficient.

# Fault tolerant design

- Focus on naturally occurring faults
- No attacker control
- Space probes, hostile environments, radiation, solar flares
- Not enough against fault attacks
  - Attacker has a goal, natural faults do not
  - Some techniques will help

# Countermeasure taxonomy

- **Level:** Where is it?
  - Algorithm
  - Software
  - Hardware / Logical
  - Physical / Electrical
- **Objective:** What does it do?
  - Prevention
  - Detection
  - Correction
  - Infection
- **Killchain step:** What does it eliminate?
  - 1 Fault injection
  - 2 Fault manifestation
  - 3 Fault propagation
  - 4 Fault observation
  - 5 Fault exploitation

## Embedded Cryptography 1

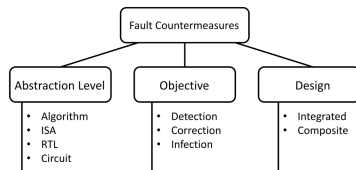


Figure 12.1. Taxonomy of fault countermeasures along three dimensions

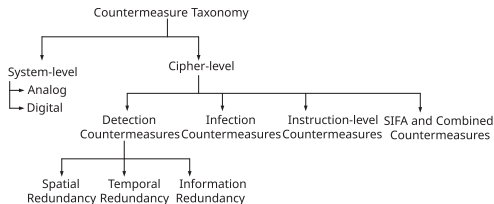


Figure 10.5. Taxonomy of notable FA countermeasures

# Attack complexity

## Attacker's goals ✖

- Get the secret
- Bypass verification
- ...

## Defender's goals 🛡

- There is no absolute security in hardware
- Make the attack *as hard as possible* (as costly as possible)
  - More faults required → harder to do  $n$  faults than one
  - More precise faults required → harder to fault a precise time/location
  - More repeatable faults required → harder to repeat the same fault
  - More specific faults required → harder to cause a specific value than a random one
- Random bit/byte fault in a large state processed over a long time
  - Easy
- Repeated same byte set fault several times at precise times
  - Hard

# Countermeasure principles

- Prevention
- Detection + Response
- Redundancy
- Randomness
- Infection

# Prevention

- EMFI: Shielding
  - Metallic package/coating
  - Can include a detector
- Clock glitching: Clock filtering/internal clock
  - Phase-locked loops (PLLs)
  - Glitch detectors ( $T_{clk}$  too short)
- Voltage glitching: Power filtering
  - Low-dropout regulators (LDOs)
  - Glitch detectors (voltage sensors)
- ...
- 👍 Prevents faults altogether
- 👍 Also prevents natural faults
- 🚫 Can be bypassed using different injection technique

- |   |                              |
|---|------------------------------|
| 1 | Algorithm                    |
| 2 | Software                     |
| 3 | Hardware / Logical           |
| 4 | <b>Physical / Electrical</b> |

- |   |                   |
|---|-------------------|
| 1 | <b>Prevention</b> |
| 2 | Detection         |
| 3 | Correction        |
| 4 | Infection         |

- |   |                        |
|---|------------------------|
| 1 | <b>Fault injection</b> |
| 2 | Fault manifestation    |
| 3 | Fault propagation      |
| 4 | Fault observation      |
| 5 | Fault exploitation     |

# Detection

- Sensors and detectors (physical level)
  - Can also be done at higher levels (through redundancy)
- Detect fault injection before manifestation
- 👍 Independent of target functionality
- 👎 Can be bypassed using different injection technique

- 1 Algorithm
- 2 Software
- 3 Hardware / Logical
- 4 **Physical / Electrical**

- 1 Prevention
- 2 **Detection**
- 3 Correction
- 4 Infection

- 1 **Fault injection**
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 Fault exploitation

# Fault response

- Detection needs a response
- You detected a fault, what do you do?
  - Return error
  - Return random (plausible) data
  - Correct it (if you can)
  - Go silent (temporarily)
  - Go silent and delete all secrets (brick)
- Faults may occasionally happen naturally
  - Response should not be too aggressive
  - False positive ratio of detector
    - ▶ If low, response can be aggressive
    - ▶ If high, response needs to be softer
- Fault counters: Keep count of detected faults, take action after a few faults
  - Fault counter could also be faulted/teared

# Redundancy

- Redundancy: Have redundant element in the computation
- Why does redundancy help defend against fault attack?
  - How do you know you have the correct result if you only have one result?
  - Harder to inject more faults (into redundant copies)
- How do you add redundancy?
  - **Spatial**: Multiple parallel executions
  - **Temporal**: Repeat execution several times
  - **Information**: Encode values redundantly
- Requires checking and merging
  - Can also be faulted!

- 1 Algorithm
- 2 Software
- 3 Hardware / Logical
- 4 Physical / Electrical

- 1 Prevention
- 2 Detection
- 3 Correction
- 4 Infection

- 1 Fault injection
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 Fault exploitation

# Redundancy

## Temporal

- $y_1 = f(x); y_2 = f(x); \dots y_n = f(x)$
- $y_1 \stackrel{?}{=} y_2 \stackrel{?}{=} \dots \stackrel{?}{=} y_n$
- ✔ Detects up-to  $n - 1$  faults
- ✔ Can correct using majority vote
- ✘ Slow
- ✘ Check can be faulted
- ✘ May not detect semi-permanent fault

# Redundancy

## Spatial

- $y_1 = f(x) \mid y_2 = f(x) \mid \dots \mid y_n = f(x)$
- $y_1 \stackrel{?}{=} y_2 \stackrel{?}{=} \dots \stackrel{?}{=} y_n$
- 👍 Detects up-to  $n - 1$  faults
- 👍 Can correct using majority vote
- 👎 Lots of space
- 👎 Check can be faulted

# Redundancy

## Information

```
# bit in (0, 1)
if bit:
    # something
else:
    # something else
```

- ❓ What could be improved in this case?
  - Any non-zero value is truthy, zero is falsy
  - Gives attacker too much freedom to fault
  - Use values with redundancy

```
VAL_TRUE = 0xAA; VAL_FALSE = 0x55
if bit == VAL_TRUE:
    # something
elif bit == VAL_FALSE:
    # something else
else:
    # Fault!
```

# Randomness

- Redundancy not enough against many precise faults
- Unpredictable/unknown to the attacker
- ? How does randomness help?
  - Attacker needs to know what faults to do and where/when

- 1 **Algorithm**
- 2 **Software**
- 3 **Hardware / Logical**
- 4 Physical / Electrical

- 1 **Prevention**
- 2 Detection
- 3 Correction
- 4 **Infection**

- 1 **Fault injection**
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 **Fault exploitation**

# Randomness

## Misalignment

- Targets the SCA assistance to FA
- Attacker needs to time fault precisely
- 👍 Makes precise fault injection harder
- 👎 Attacker can try a lot and get lucky
- 👎 Real-time pattern matching

# Randomness

## Redundancy

```
VAL_TRUE = rand()
VAL_FALSE = not(VAL_TRUE)
if bit == VAL_TRUE:
    # something
elif bit == VAL_FALSE:
    # something else
else:
    # Fault!
```

- Randomizes redundant representations
- $y_1 = f(\text{rerand}(x)); y_2 = f(\text{rerand}(x)); \dots y_n = f(\text{rerand}(x))$   
 $\text{derand}(y_1) \stackrel{?}{=} \text{derand}(y_2) \stackrel{?}{=} \dots \stackrel{?}{=} \text{derand}(y_n)$ 
  - Only possible sometimes
- 👍 Makes specific fault injection harder
- 👎 Non-specific faults largely unaffected

# Infection

- Detection: Target detects fault and response mitigates it
- Correction: Target detects fault and is able to correct it
- **Infection:** Detection and response *embedded* in the computation
  - Response makes the result useless by randomizing it (**infecting**)
  - Response always happens, is no-op if no fault is present
- Infected result has no relationship to the secret key or plaintext

- 1 **Algorithm**
- 2 **Software**
- 3 Hardware / Logical
- 4 Physical / Electrical

- 1 Prevention
- 2 Detection
- 3 Correction
- 4 **Infection**

- 1 Fault injection
- 2 Fault manifestation
- 3 **Fault propagation**
- 4 Fault observation
- 5 **Fault exploitation**

## Block cipher

**Inputs:**  $P, K = k^j$  for  $j \in \{1, \dots, n\}$ ,  $(\beta, k^0)$  dummy plaintext and key s.t.;  $RoundFunction(\beta, k^0) = \beta$

**Output:**  $C = \text{BlockCipher}(P, K)$

```

1: State  $R_0 \leftarrow P$ , Redundant state  $R_1 \leftarrow P$ , Dummy state  $R_2 \leftarrow \beta$ 
2:  $i \leftarrow 1, q \leftarrow 1$ 
3:  $rstr \leftarrow \{0, 1\}^t$ 
4: while  $q \leq t$  do
5:    $\lambda \leftarrow rstr[q]$ 
6:    $\kappa \leftarrow (i \wedge \lambda) \oplus 2(\neg \lambda)$ 
7:    $\zeta \leftarrow \lambda \cdot \lceil i/2 \rceil$ 
8:    $R_\kappa \leftarrow \text{RoundFunction}(R_\kappa, k^\zeta)$ 
9:    $\gamma \leftarrow \lambda(\neg(i \wedge 1)) \cdot \text{IsNonZero}(R_0 \oplus R_1)$ 
10:   $\delta \leftarrow (\neg \lambda) \cdot \text{IsNonZero}(R_2 \oplus \beta)$ 
11:   $R_0 \leftarrow (\neg(\gamma \vee \delta) \cdot R_0) \oplus ((\gamma \vee \delta) \cdot R_2)$ 
12:   $i \leftarrow i + \lambda$ 
13:   $q \leftarrow q + 1$ 
14: return  $(R_0)$ 

```

$\triangleright \#1(rstr) = 2n, \#0(rstr) = t - 2n$   
 $\triangleright \lambda = 1$  real round,  $\lambda = 0$  dummy round  
 $\triangleright \kappa$  selects the state  
 $\triangleright \zeta$  is actual round counter, 0 for dummy  
 $\triangleright$  Do the round  
 $\triangleright$  Was there a fault in  $R_0$  or  $R_1$ ?  
 $\triangleright$  Was there a fault in  $R_2$ ?  
 $\triangleright$  Infect  $R_0$  if faulty

# **Attacks & Countermeasures**

# Attacks & Countermeasures

## DFA

- Attacker needs  $\geq$  one correct encryption,  $\geq$  one faulty encryption with an unknown (single byte) fault in a certain round, on a fixed plaintext

### 1 Fault injection

- Prevent / Detect + abort
- Dummy rounds
- Misalignment

### 2 Fault manifestation

- Detect + abort / Correct

### 3 Fault propagation

- Infect / Correct

### 4 Fault observation

- Detect + abort / correct

### 5 Fault exploitation

- Do not allow repeated plaintexts (CBC with random IV)

# Attacks & Countermeasures

## SFA

- 1 Fault injection
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 Fault exploitation

- Attacker needs many faulty ciphertexts, with an unknown biased fault in a certain round, the plaintext no longer needs to be fixed
- Same as DFA, disallowing repeated plaintexts no longer works
- Limiting key reuse prevents exploitation (only up to  $n$  encryptions per key)
- Detection can be more relaxed compared to DFA

# Attacks & Countermeasures

## SIFA

- 1 Fault injection
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 Fault exploitation

- Attacker needs many correct ciphertexts, which had an ineffective (biased) fault in a certain round, the plaintext no longer needs to be fixed
- Prevention + detection on physical level still works
- Detection on a logical level no longer works
  - SIFA uses ineffective faults which do not manifest
  - Yet, running a SIFA attack will also cause effective faults
  - These can be detected before too much leaks
  - But, response must be permanent
- Correction still works

# Attacks & Countermeasures

## RSA-CRT Bellcore attack

- |   |                     |
|---|---------------------|
| 1 | Fault injection     |
| 2 | Fault manifestation |
| 3 | Fault propagation   |
| 4 | Fault observation   |
| 5 | Fault exploitation  |

- Attacker needs any (unknown) fault in computation of:

$$s_1 \equiv m^{d_p} \bmod p$$

$$s_2 \equiv m^{d_q} \bmod q$$

- Resulting signature is invalid
- What can we do?
  - All of the generic prevention and detection
  - Sign-then-verify
- Shamir's countermeasure: Redundancy on algorithm level
  - 1  $r \leftarrow$  random prime (small)
  - 2  $s_1 \equiv m^{d \bmod \varphi(rp)} \bmod r \cdot p$
  - 3  $s_2 \equiv m^{d \bmod \varphi(rq)} \bmod r \cdot q$
  - 4 Check  $s_1 \equiv s_2 \bmod r$

# Attacks & Countermeasures

## ECDSA (repeated nonces)

- |   |                     |
|---|---------------------|
| 1 | Fault injection     |
| 2 | Fault manifestation |
| 3 | Fault propagation   |
| 4 | Fault observation   |
| 5 | Fault exploitation  |

- Attacker needs repeated use of nonce  $k$  for two different messages
- Hard to fix
  - There is no state that remembers past nonces
  - Nonces are (supposed to be) random
  - Many ways a repeat can happen: RNG stuck at 0, ...
- Generic prevention and detection works
- Hedged nonce derivation: Deterministic + random
  - $k = H(m, d, rand())$
  - Deterministic part provides difference for  $m_1 \neq m_2$

# Attacks & Countermeasures

## ECDSA (related nonces)

- 1 Fault injection
- 2 Fault manifestation
- 3 Fault propagation
- 4 Fault observation
- 5 Fault exploitation

- Attacker needs to know the relation between two nonces for the same message
- Generic prevention and detection works
- Hedged nonce derivation: Deterministic + random
  - $k = H(m, d, rand())$
  - Random part provides unguessable randomness

# Attacks & Countermeasures

## ECDH safe-error

- Attacker faults operation and detects whether it is a dummy or not
- Remove dummy operations
  - There should not be values that are used/unused based on key bits
- Problem: M safe-error attacks

# Takeaways

- How to stop a fault attack by targeting a step in its kill chain
- How to push attacks into less practical territory
- There is a large variety of fault attack countermeasures

- Fault injection with ChipWhisperer-Nano

# Next week

- Guest lecture by **Thomas Roche**
  - On Zoom at the *usual time*
  - Everyone please come if you can
  - You will have a chance to ask questions
- No tutorial

# Recommended reading

- For **this** lecture
  - Read Embedded Cryptography 1 – Chapter 12 – Fault countermeasures
- For the **next** lecture (13th week)
  - Nothing
  - Thomas will talk about: [A Side Journey to Titan](#) and [EUCLEAK](#)
- For the **next next** lecture (14th week)
  - Nothing
  - Łukasz will talk about: [SoK: SCA-secure ECC in software – mission impossible?](#)
- For the **last** lecture (15th week)
  - Read Security Engineering – Chapter 28 – Assurance and Sustainability

# References

- Secure Application Programming in the presence of Side Channel Attacks
- Infective Computation and Dummy Rounds: Fault Protection for Block Ciphers without Check-before-Output
- Destroying Fault Invariant with Randomization
- SIFA: Exploiting Ineffective Fault Inductions on Symmetric Cryptography
- A synthesis of side-channel attacks on elliptic curve cryptography in smart-cards
- A Systematic M Safe-error Detection in Hardware Implementations of Cryptographic Algorithms