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

Hardware Security

9. Physical side-channel attack defences 1 (passive)

Ján Jančár

Summer 2026

Today

- Attacks and what they need
- Hiding
- Masking
- Blinding

Assignments

Assignment 2

- There is a leaderboard
  - Please, use side-channel attacks to solve the assignment
  - The curve is small enough such that ECDLP solving is possible

Attacks

SPA

```
def scalarmult(P, k):
  R = P
  for i in range(k.bit_length()-1,-1,-1):
    R = dbl(R)
    if k & (1 << i):
       R = add(R, P)
  return R
```

```
def scalarmult(P, k):
  R = infty
  for i in range(255,-1,-1):
    R = dbl(R)
    if k & (1 << i):
      R = add(R, P)
    else:
      dummy = add(R, P)
  return R
```

- Exploits **operation** leakage
- Detects **patterns** that correspond to operations
- **One** or a few traces
- How do we prevent it?
  - Regular algorithms, essentially like constant-time
  - Most of the other countermeasures also work, but regular algorithms are the basis
  - Not so easy in some cases: Minerva leakage

Attacks

Minerva leakage

```
def scalarmult(P, k):
  R = P
  for i in range(k.bit_length()-1,-1,-1):
    R = dbl(R)
    if k & (1 << i):
      R = add(R, P)
    else:
      dummy = add(R, P)
  return R
```

- Incomplete formulas
- → Can not compute with the point-at-infinity
- → Can not start from fixed bit-length
- → Has to use a different countermeasure (Brumley-Tuveri)

$$\hat{k} = \begin{cases} k + 2n & \text{if } \lceil \log_2(k+n) \rceil = \lceil \log_2 n \rceil, \\ k + n & \text{otherwise.} \end{cases}$$

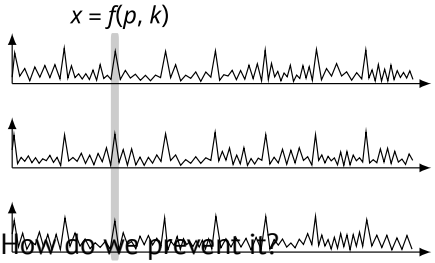
- Fixes $\log_2(\hat{k})$ to a constant value, $[\hat{k}]G = [k]G$
- Nuances of the cryptographic algorithm hinder/help countermeasures

Attacks

DPA

```
def scalarmult(P, k):
  R = infty
  for i in range(255,-1,-1):
    R = dbl(R)
    if k & (1 << i):
      R = add(R, P)
    else:
      dummy = add(R, P)
  return R
```

- Exploits **data** leakage
- Many traces with **fixed secret**
- Processes traces **sample-wise**
- Simulation of **intermediate values**



- How do we prevent it?
  - Introduce misalignment
  - Leak less
  - Introduce noise
  - Randomize or split intermediate values
  - Never reuse a secret

Countermeasures

Level

- **Algorithmic/Software** level
  - Change the algorithm or software
  - 👍 Easy, cheap, reconfigurable
  - 👎 Least control
- **Hardware** level
  - Change the hardware, but not the logic cells
  - 👍 High-level of control
  - 👎 Not that expensive
- **Cell** level
  - Change the logic cells themselves
  - 👍 Most control
  - 👎 Most custom and expensive

In practice, secure hardware uses a combination of all three.

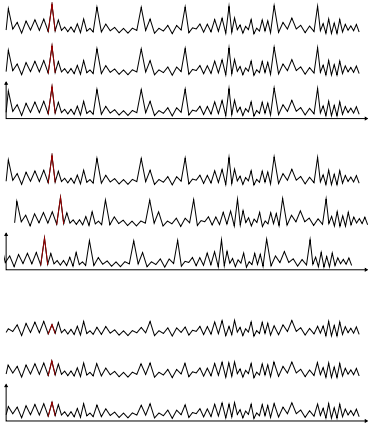
Countermeasures

Type

- **Hiding**
  - Make the power consumption **independent** of operations and intermediate values
  - *"Hide the leakage*"
- **Masking**
  - Make the power consumption **useless** by splitting the intermediate values
  - *"Randomize the leakage*"
- **Blinding**
  - Make the power consumption **useless** by randomizing the intermediate values
  - *"Randomize the leakage*" (without splitting)

Hiding

- **Goal**: Make the power consumption **independent** of operations and intermediate values
- Either
  - Make the consumption random → More noise
  - Make the consumption constant → Less signal
- Two ways
  - Time domain
    - Hide the leakage in some timespan
  - Amplitude domain
    - Hide the leakage signal itself



Hiding

Time domain (Software + Hardware)

- **Random initial delay**
  - Misaligns the beginning of an operation
  - Can be quite easily realigned, the rest is aligned

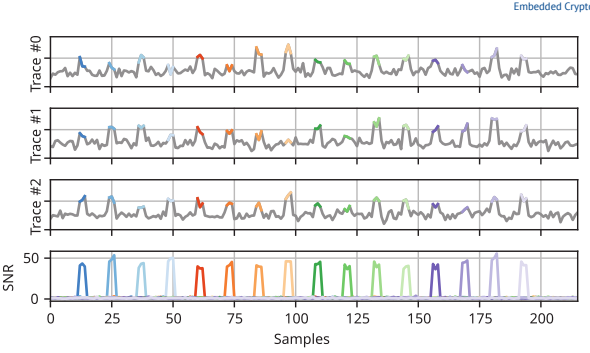


Figure 7.1. *Examples of side-channel traces and SNR for an unprotected SubBytes transformation implementation.*

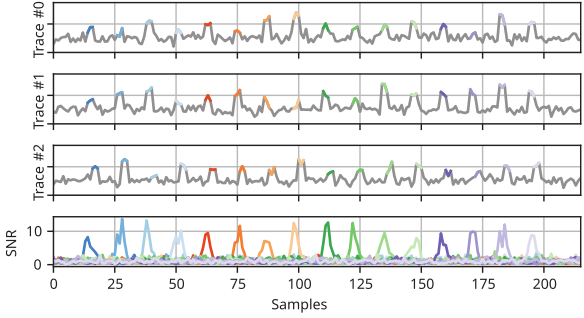


Figure 7.2. *Examples of side-channel traces and SNR for an implementation of the SubBytes transformation protected by the addition of a random initial delay.*

Hiding

Time domain (Software + Hardware)

- **Random delay interrupts**
  - Introduces random misalignment at random points
  - More randomness necessary

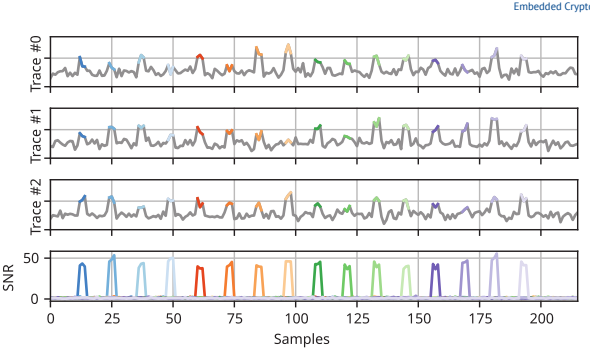


Figure 7.1. *Examples of side-channel traces and SNR for an unprotected SubBytes transformation implementation.*

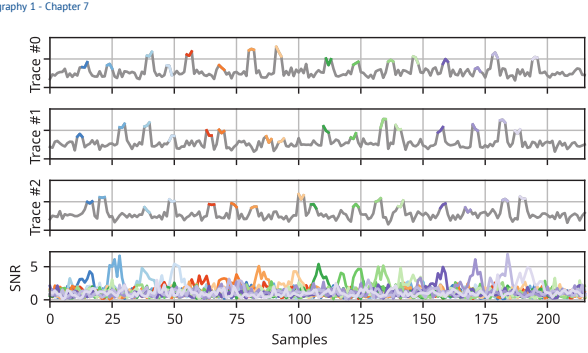


Figure 7.3. *Examples of consumption and SNR traces for an implementation of the SubBytes transformation protected by insertion of random delay interrupts.*

Hiding

Time domain (Software + Hardware)

- **Shuffling independent operations**
  - Re-orders parts of the trace
  - Not always applicable (needs independent operations)

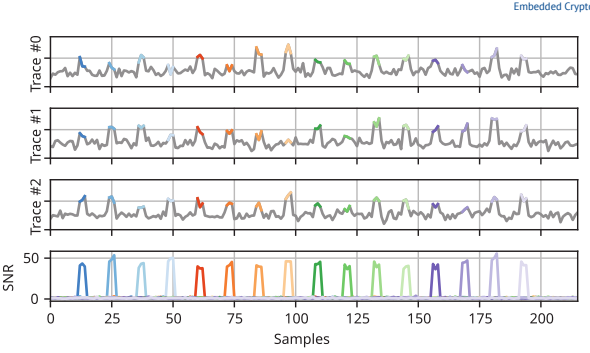


Figure 7.1. *Examples of side-channel traces and SNR for an unprotected SubBytes transformation implementation.*

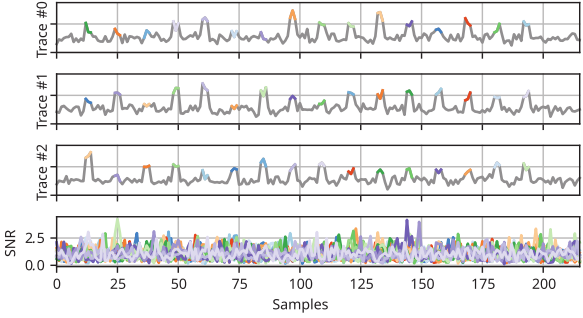


Figure 7.4. *Examples of side-channel traces and SNR for a shuffling-protected implementation of the SubBytes transformation.*

Hiding

Time domain (Software + Hardware)

- **Changing clock frequency**
  - 👍 Warps the time-axis continuously, making standard trace alignment algorithms fail
  - 👎 Can be bypassed using advanced signal processing
- **Skipping clock cycles**
  - 👍 Target does not execute on some clock cycles
  - 👎 These cycles can be recognizable

Hiding

Amplitude domain (Software)

- **Regular and constant-time algorithms**
  - The baseline for all other countermeasures
  - Sometimes, it is costly and does not go well with optimizations
- **Parallel algorithms**
  - Leakage of intermediates the attacker is not targeting acts like noise (**algorithmic noise**)
  - Example:
    - In 1-bit DPA, the other bits are noise
    - The larger the word-size of the target, the harder to attack
    - 8-bit → easy, 32-bit → harder
    - Assignment 2 simulates a 64-bit target

Hiding

Amplitude domain (Hardware)

- **Noise generation**
  - Separate hardware component produces noise in power
  - 👍 Introduces amplitude noise
  - 👎 Needs to be random
  - 👎 EM leakage is localized, it may not cover the leaking part
- **Filtering/current flattening**
  - Filter/flatten the power consumption to decrease signal
  - Voltage regulators and capacitors
  - 👍 Decreases signal in *any* leakage
  - 👎 Requires separate hardware
  - 👎 EM leakage is localized

Hiding

Amplitude domain (Cell)

- **Dual-Rail Precharge (DRP)**
  - DPA-resistant *logic style* (... there are more: SABL, WDDL)
  - Makes the power consumption constant
  - **Dual-Rail**: Logic signal a is represented by a and $\overline{a}$
  - **Precharge**: Two-phase circuit
    - Pre-charge phase: Reset to constant value (1, 1)
    - Evaluation phase: Set to logic value (1, 0) or (0, 1)
  - → Hamming distance stays constant
    $HD((1,1),(1,0)) = HD((1,1),(0,1)) = 1$
  - → Hamming weight also constant
    $HW((1,0)) = HW((0,1)) = 1$
  - 👍 Theoretically removes leakage
  - 👎 High cost, increases silicon area, requires custom cells

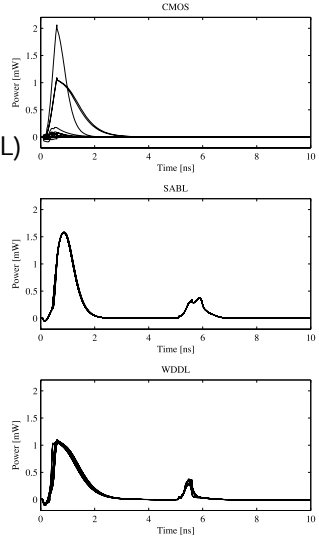


Figure 8.3. Simulated power traces of a CMOS NAND cell, an SABL NAND cell, and a WDDL NAND cell for different input transitions.

Masking

- **Goal**: Make the power consumption **useless** by splitting the intermediate values
- Split each secret x into $d + 1$ variables (**shares**)
  - d of them random (d is the **masking order**)
  - last one such their combination makes up x
- Perform all operations on the shares **separately**
  - The unmasked secret must never appear in the clear
- Software, hardware and cell level
- Several masking schemes
- 👍 Theoretically removes leakage up to d probes
- 👎 High performance cost

All sorts of things can be proven about masking.

Masking

Example

- First-order boolean masking of a single bit x
  - Masking x: Pick random r, set $x_0 = x \oplus r$, $x_1 = r$
  - Recombination: $x_0 \oplus x_1$
  - Representations
    - $x = 1$ -> $(0, 1), (1, 0)$
    - $x = 0$ -> $(0, 0), (1, 1)$
  - Even if attacker learns one of x_0 or x_1 they know nothing about x
    - Attacker learns x_0: $Pr[x = 1 | x_0 = 0] = Pr[x = 0 | x_0 = 0]$
    - Attacker learns x_1: $Pr[x = 1 | x_1 = 0] = Pr[x = 0 | x_1 = 0]$

Masking

Example

- First-order boolean masking of a single bit x
  - Masking x: Pick random r, set $x_0 = x \oplus r$, $x_1 = r$
  - Recombination: $x_0 \oplus x_1$
  - Representations
    - $x = 1$ -> $(0, 1), (1, 0)$
    - $x = 0$ -> $(0, 0), (1, 1)$
  - Even if attacker learns one of x_0 or x_1 they know nothing about x
    - Attacker learns x_0: $Pr[x = 1 | x_0 = 0] = Pr[x = 0 | x_0 = 0]$
    - Attacker learns x_1: $Pr[x = 1 | x_1 = 0] = Pr[x = 0 | x_1 = 0]$
- Why is it important that operations on the shares are done separately?

Masking

Example

- First-order boolean masking of a single bit x
  - Masking x: Pick random r, set $x_0 = x \oplus r$, $x_1 = r$
  - Recombination: $x_0 \oplus x_1$
  - Representations
    - $x = 1$ -> $(0, 1), (1, 0)$
    - $x = 0$ -> $(0, 0), (1, 1)$
  - Even if attacker learns one of x_0 or x_1 they know nothing about x
    - Attacker learns x_0: $Pr[x = 1 | x_0 = 0] = Pr[x = 0 | x_0 = 0]$
    - Attacker learns x_1: $Pr[x = 1 | x_1 = 0] = Pr[x = 0 | x_1 = 0]$
- What would happen in the above example if operations with both shares happened at the same time and their Hamming weight leaked?
  - $x = 1 \rightarrow HW((0, 1)) = HW((1, 0)) = 1$
  - $x = 0 \rightarrow HW((1, 1)) = 2$ or $HW((0, 0)) = 0$
  - Hamming weight leaks information on secret
  - An issue for hardware implementations (mostly)

Masking

Example

- First-order boolean masking of a single bit x
  - Masking x: Pick random r, set $x_0 = x \oplus r$, $x_1 = r$
  - Recombination: $x_0 \oplus x_1$
  - Representations
    - $x = 1$ -> $(0, 1), (1, 0)$
    - $x = 0$ -> $(0, 0), (1, 1)$
  - Even if attacker learns one of x_0 or x_1 they know nothing about x
    - Attacker learns x_0: $Pr[x = 1 | x_0 = 0] = Pr[x = 0 | x_0 = 0]$
    - Attacker learns x_1: $Pr[x = 1 | x_1 = 0] = Pr[x = 0 | x_1 = 0]$
- What about some register holding x_0 first then x_1?
  - Hamming distance leaks information on secret
  - $HD(x_0, x_1) = HW(x_0 \oplus x_1) = HW(x)$
  - An issue for software implementations (mostly)

Masking

Types

- **Boolean**
  - Secret split via XOR: $x = x_0 \oplus x_1 \oplus \ldots \oplus x_d$
  - 👍 Good for symmetric crypto
- **Arithmetic**
  - Secret split via modular addition: $x = x_0 + x_1 + \ldots + x_d \mod n$
  - 👍 Good for asymmetric crypto
- **Multiplicative**
  - Secret split via modular multiplication: $x = x_0 \cdot x_1 \cdot \ldots \cdot x_d \mod n$
  - 👍 Useful for AES S-Box (field inversions)
  - 👎 Can't mask zero value
- Which to use?
  - Depends on the operations
  - You want them to be linear w.r.t. the masking operation
- You can convert between them
  - For a cost (randomness, time)
- There is more
  - Polynomial, Inner-Product, ...

Masking

Boolean

- **Masking** x: Pick random r, set $x_0 = x \oplus r$, $x_1 = r$
- **Recombination**: $x = x_0 \oplus x_1$
- **Linear operations** (e.g., $z = x \oplus y$):
  - Done completely **separately** (no share cross-contamination):
  - $z_0 = x_0 \oplus y_0$ and $z_1 = x_1 \oplus y_1$
- **Non-linear operations** (e.g., $z = x \cdot y$):
  - Cannot be done separately
  - Cross-terms ($x_0 \cdot y_1$) leak if unbuffered
  - Requires **fresh randomness (r')** to blind cross-terms before execution
- 👍 Efficient for bitwise operations
- 👎 Blows up non-linear operations into $O(d^2)$ complexity and uses lots of randomness

Masking

Arithmetic

- **Masking** x: Pick random r, set $x_0 = (x - r) \mod n$, $x_1 = r$
- **Recombination**: $x = (x_0 + x_1) \mod n$
- **Linear operations** (e.g., Adding a constant c):
  - Simply add to one share: $z_0 = (x_0 + c) \mod n$, $z_1 = x_1$
- **Non-linear operations** (e.g., $z = x \cdot y \mod n$):
  - Requires complex cross-term calculation
  - Algebraic multiplications hard unless converted
- 👍 Fits naturally with modular exponentiation and point addition in ECC
- 👎 Bad for logical bitwise operations (AND, XOR, shifts)

Masking

Doing it right

- **Mask refreshing**
  - To not reuse the same masks too long
  - Boolean: $Refresh(x) = x \oplus Mask(0)$
  - Arithmetic: $Refresh(x) = x + Mask(0) \mod n$
- **Mask conversion**
  - Some operations are way faster in some masking schemes
  - Conversion is possible, but also costly

Blinding

- **Goal**: Make the power consumption **useless** by randomizing the intermediate values
- Algorithm-specific randomization
- Randomize intermediate values for each execution
- Example (RSA decryption)
  - Ciphertext $c = m^e$
  - Blind as $c' = c \cdot r^e$ with random r
  - Decrypt c' and obtain $m' = (c \cdot r^e)^d = m \cdot r$
  - Unblind with r^{-1} and obtain $m \cdot r \cdot r^{-1} = m$

Blinding

Scalar randomization

Group scalar randomization

function Mult(P, k)
  $r \xleftarrow{\$} \{0, 1, \dots, 2^b\}$
  return $[k + rn]P$

Additive splitting

function Mult(P, k)
  $r \xleftarrow{\$} \mathbb{Z}_n^*$
  return $[k - r]P + [r]P$

Euclidean splitting

function Mult(P, k)
  $r \xleftarrow{\$} \{0, 1, \dots, 2^{\lfloor \log_2(n)/2 \rfloor}\}$
  $S \leftarrow [r]P$
  $k_1 \leftarrow k \mod r$
  $k_2 \leftarrow \left\lfloor \frac{k}{r} \right\rfloor$
  return $[k_1]P + [k_2]S$

Multiplicative splitting

function Mult(P, k)
  $r \xleftarrow{\$} \{0, 1, \dots, 2^b\}$
  $S \leftarrow [r]P$
  return $[kr^{-1} \mod n]S$

Blinding

Curve and coordinate randomization

Curve randomization

function Mult(P, k, E)

$r \xleftarrow{\$} \{0, 1, \ldots, 2^b\}$

$E_r \leftarrow$ isomorphic curve$_r(E)$

$P_r \leftarrow \text{map}(P, E_r)$

$R_r \leftarrow [k]P_r$

$R \leftarrow \text{map}(R_r, E)$

return R

Coordinate randomization

function Mult($P = (X : Y : Z), k$)

$\lambda \xleftarrow{\$} \mathbb{Z}_p^*$

$Q \leftarrow (\lambda X : \lambda Y : \lambda Z)$

return $[k]Q$

Takeaways

- There is a huge variety of countermeasures
- The countermeasures come at a cost
- Real-world secure hardware combines them

Recommended reading

- For **this** lecture
  - Read Embedded Cryptography 1 - Chapter 7 - Countermeasures and Advanced Attacks
    - Focus on the countermeasures
- For the **next** lecture
  - Skim Power Analysis Attacks - Chapter 8 - Attacks on Hiding
  - Skim Power Analysis Attacks - Chapter 10 - Attacks on Masking
    - Next week we break what we built today

References

- Power Analysis Attacks: Revealing the Secrets of Smart Cards
- Embedded Cryptography 1