University of
Southern Denmark

Cryptographic Engineering

SCA Countermeasures

Ruben Niederhagen
(based on content by Norman Lahr and Richard Petri)

Department of Mathematics and Computer Science (IMADA)

Introduction

- (Power) leakage depends on the processing of intermediate values.
- The goal of countermeasures is …
  - to avoid or
  - to reduce the dependencies.
- Classes of countermeasures are
  - hiding and
  - masking or blinding.

Countermeasures

Hiding

Introduction

- The goal of hiding is to break the link between the power consumption and the processed data values.
- The execution of the cryptographic algorithm computes the same intermediate value as before.
- Hiding makes it difficult to find (hides) leakage in the power traces.

Ideal Goal

Ideal Properties

The power consumption is independent from the intermediate values if the devices consumes:

- random amounts or
- equal amounts

of power in each clock cycle.

Types of Hiding Techniques

- Prefect randomness or equality cannot be reached in practice.
- However, there are solution which get close to this.
- Hiding can be applied in two dimensions:
    - the time dimension and
    - the amplitude dimension.

Time Dimension

Hiding in the time dimension randomizes the power consumption by the execution of operations at different moments in time. This:

- breaks the alignment property of traces for DPA,
- increases the amount of required traces, and
- decreases the performance.

$\Rightarrow$ A suitable compromise is required.

Time Dimension — Mechanisms

The security of the hiding mechanisms in the time dimension depends on the randomness and the undetectability.

- Software:
  - Random insertion of operations and
  - shuffling of operations
    (requires independence of shuffled operations (e.g. AES S-box lookups).
- Hardware:
  - Random insertion of cycles
    (requires duplication of registers for random data execution),
  - skipping clock pulses, and
  - random variation of clock frequency.

Example AES Shuffling

```
for (i = 0; i < 16; i++)
  state[i] = sbox[state[i] ^ key[i]];
```

$\downarrow$

```
int order[16] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
shuffle(order); /* Shuffles the order */
for (o = 0; o < 16; o++) {
  int i = order[o];
  state[i] = sbox[state[i] ^ key[i]];
}
```

Amplitude Dimension

In the hiding techniques using the amplitude dimension directly change the power-consumption characteristics:

- Equalize or randomize the power consumption per clock cycle.
- Goal: lower the signal-to-noise ratio.

Equalize: Reduce the leakage signal $Var(P_{exp}) \to 0$.

Randomize: Increase the noise $Var(P_{sw.noise} + P_{el.noise}) \to \infty$.

Amplitude Dimension — Mechanisms

- Software:
  - Careful choice of instructions,
  - avoidance of key-dependent (or related) conditional jumps and program flow patterns,
  - avoidance of key-dependent memory addresses;
    usage of, e.g., addresses with equal Hamming weights, and
  - increase of parallel activities.
- Hardware:
  - Filtering or regulation of the power consumption (supply),
  - noise engine, and
  - dual-rail precharge (DRP) logic on the logic gate design level.

Dual-Rail Precharge Logic

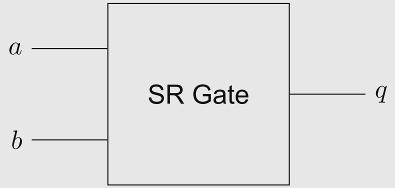
Dual-rail precharge (DRP) logic equalizes the power consumption in each clock cycle:

- Dual-rail logic: Duplicate wires and gates; add and compute the complementary sinal.
- Precharge logic: Split logic computation into precharging and evaluation phases.

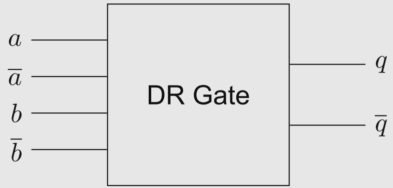
The combination of both reduces leakage from power consumption of buses and logic.

Dual-Rail Logic

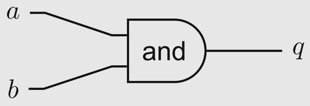
Single-Rail Logic



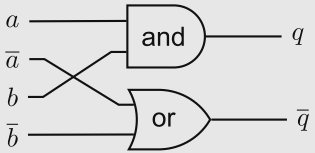
Dual-Rail Logic



Example

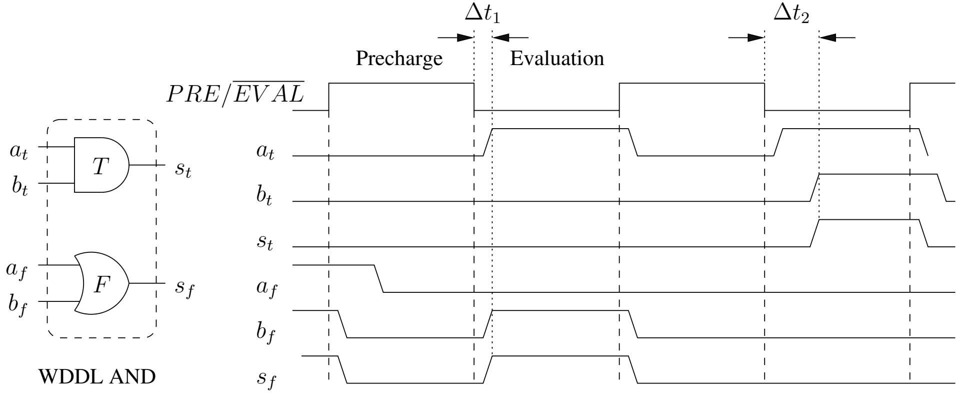


Example



Precharge Logic

- Switching ($0 \rightarrow 1$, $1 \rightarrow 0$) can be differentiated from non-switching ($0 \rightarrow 0$, $1 \rightarrow 1$).
- Solution: Wave Dynamic Differential Logic (WDDL) and other variants.



Countermeasures

Masking (Blinding)

Introduction

Masking makes the power consumption independent of the processed intermediate value by randomizing the intermediate value(s).

- It does not affect the original (data-dependent) power characteristics.
- An intermediate value is concealed by a random value, called mask: $v_m = v \circ m$.
- The mask m is unknown to the attacker.
- A mask on public data is removed after the computation process.
- The application in context of asymmetric schemes is called blinding (cf. timing slides).

Boolean vs. Arithmetic Masking

The masking scheme must fit to the crypto scheme:

Boolean Masking

The mask m is applied using an exclusive-or operation:

$$v_m = v \oplus m$$

Arithmetic Masking

The mask m is applied using an arithmetic operation (addition or multiplication):

$$v_m = v + m \mod n$$
$$v_m = v \times m \mod n$$

Linear vs. Non-linear Functions

- The choice of the masking scheme depends on the cryptographic algorithm.
- Cryptographic algorithms use linear and non-linear functions.
  - Linear: $f(x \circ y) = f(x) \circ f(y)$.
  - Non-linear: $f(x \circ y) \neq f(x) \circ f(y)$.

AES S-box Operation

The S-box operation is based on operation on the multiplicative inverse of finite field element $f(x) = x^{-1}$. Therefore, boolean masking is not applicable:

$$S(x \oplus m) \neq S(x) \oplus S(m).$$

However, multiplicative masking can be applied due to:

$$f(x \times m) = (x \times m)^{-1} = f(x) \times f(m).$$

Secret Sharing

- The intermediate value v is represented by the two shares (v_m, m).
- The knowledge of just one share does not reveal v.
- Masking prevents 1st-order DPA attacks if v_m is pairwise independent to v and m.
- Multiple masks can be used to prevent n-th order DPA.
- Several masks leads to higher computation and memory usage.

Example Masking of a Linear XOR Gate

$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \oplus m_1) \oplus (y_{m_2} \oplus m_2)$$

$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \oplus y_{m_2}) \oplus (m_1 \oplus m_2)$$

$$z_{m_3} \leftarrow x_{m_1} \oplus y_{m_2}, \qquad m_3 \leftarrow m_1 \oplus m_2$$

- Let’s construct an XOR operation with two shares.
- XOR operation can be applied to the shares individually (if the operands are independent).

Example Masking of a Non-linear AND Gate

$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \oplus m_1) \wedge (y_{m_2} \oplus m_2)$$
$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \wedge y_{m_2}) \oplus (x_{m_1} \wedge m_2) \oplus (m_1 \wedge y_{m_2}) \oplus (m_1 \wedge m_2)$$

$$\begin{aligned} s_0 &\leftarrow x_{m_1} \wedge y_{m_2}, & s_1 &\leftarrow x_{m_1} \wedge \ m_2 \\ s_2 &\leftarrow m_1 \wedge y_{m_2}, & s_3 &\leftarrow m_1 \wedge \ m_2 \\ t_0 &\leftarrow s_0 \oplus m', & t_1 &\leftarrow s_1 \oplus m' \\ z_{m_3} &\leftarrow t_0 \oplus s_2, & m_3 &\leftarrow t_1 \oplus s_3 \end{aligned}$$

- Let's construct an AND operation with two shares.
- Direct approach to constructing an AND gate with four output shares, which are registered and recombined.
- Output must be uniform, requiring re-masking with a guard share m'.

SDU

Example Masking of a Non-linear AND Gate

$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \oplus m_1) \wedge (y_{m_2} \oplus m_2)$$

$$(z_{m_3} \oplus m_3) \leftarrow (x_{m_1} \wedge y_{m_2}) \oplus (x_{m_1} \wedge m_2) \oplus (m_1 \wedge y_{m_2}) \oplus (m_1 \wedge m_2)$$

$$s_0 \leftarrow x_{m_1} \wedge y_{m_2}, \qquad s_1 \leftarrow x_{m_1} \vee \neg m_2$$

$$s_2 \leftarrow m_1 \wedge y_{m_2}, \qquad s_3 \leftarrow m_1 \vee \neg m_2$$

$$z_{m_3} \leftarrow s_0 \oplus s_1, \qquad m_3 \leftarrow s_2 \oplus s_3$$

- Let's construct an AND operation with two shares.
- Direct approach to constructing an AND gate with four output shares, which are registered and recombined.
- Output must be uniform, requiring re-masking with a guard share m'.
- If we are careful, we can avoid this guard share. (This does NOT work repeatedly!)

Example Masking on AES

- Analyze the operation of an AES implementation with regard to adding masks.
- Identify linear and non-linear functions.
- Intermediate values must be masked all the time.
- Example from Herbst et al., 2006, for software implementation.

Example Masking on AES

Analysis of the four AES Operations

- AddRoundKey: $s \oplus k \Rightarrow s \oplus (k \oplus m) = (s \oplus k) \oplus m$.
- SubBytes: Non-linear. Use a masked S-box table.
- ShiftRows: No effect if the same mask byte is used for all state bytes.
- MixColumns:
  - Mixes the state bytes.
  - Requires in- and output masks.
  - Care must be taken to not unmask the intermediate values.
  - Using a joint mask for each row performs well.

Example Masking on AES

Preparation

- Generate six random byte masks: m, m' and m_1, m_2, m_3, m_4.
- Compute a masked S-box table S_m for the chosen masks m and m' so that:

$$S_m(x \oplus m) = S(x) \oplus m'.$$

- Compute the output masks of MixColumns so that:

$$(m'_1, m'_2, m'_3, m'_4) = \text{MixColumns}(m_1, m_2, m_3, m_4).$$

Example Masking on AES

AES Round Masking

1. Plaintext/state is masked with (m'_1, m'_2, m'_3, m'_4), m'_i for an entire row.
2. AddRoundKey: The round key is masked so that the masks change from (m'_1, m'_2, m'_3, m'_4) to m.
3. SubBytes: The lookup on S_m changes the mask from m to m'.
4. ShiftRows: All bytes are masked with byte m'. No influence.
5. Remasking before MixColumns: Change masks from m' to (m_1, m_2, m_3, m_4).
6. MixColumns: The masks are changed from (m_1, m_2, m_3, m_4) to (m'_1, m'_2, m'_3, m'_4).

$\Rightarrow$ Mask the final AddRoundKey such that it removes the mask in the last round.

Example Masking on AES

Initialization

State (plaintext)

$$\begin{array}{|c|c|c|c|}\hline p_0 & p_4 & p_8 & p_{12} \\ \hline p_1 & p_5 & p_9 & p_{13} \\ \hline p_2 & p_6 & p_{10} & p_{14} \\ \hline p_3 & p_7 & p_{11} & p_{15} \\ \hline \end{array}$$

$$\oplus$$

$$\begin{array}{|c|c|c|c|}\hline m'_1 & m'_1 & m'_1 & m'_1 \\ \hline m'_2 & m'_2 & m'_2 & m'_2 \\ \hline m'_3 & m'_3 & m'_3 & m'_3 \\ \hline m'_4 & m'_4 & m'_4 & m'_4 \\ \hline \end{array}$$

Round key

$$\begin{array}{|c|c|c|c|}\hline rk_0 & rk_4 & rk_8 & rk_{12} \\ \hline rk_1 & rk_5 & rk_9 & rk_{13} \\ \hline rk_2 & rk_6 & rk_{10} & rk_{14} \\ \hline rk_3 & rk_7 & rk_{11} & rk_{15} \\ \hline \end{array}$$

$$\oplus$$

$$\begin{array}{|c|c|c|c|}\hline m'_1 \oplus m & m'_1 \oplus m & m'_1 \oplus m & m'_1 \oplus m \\ \hline m'_2 \oplus m & m'_2 \oplus m & m'_2 \oplus m & m'_2 \oplus m \\ \hline m'_3 \oplus m & m'_3 \oplus m & m'_3 \oplus m & m'_3 \oplus m \\ \hline m'_4 \oplus m & m'_4 \oplus m & m'_4 \oplus m & m'_4 \oplus m \\ \hline \end{array}$$

Example Masking on AES

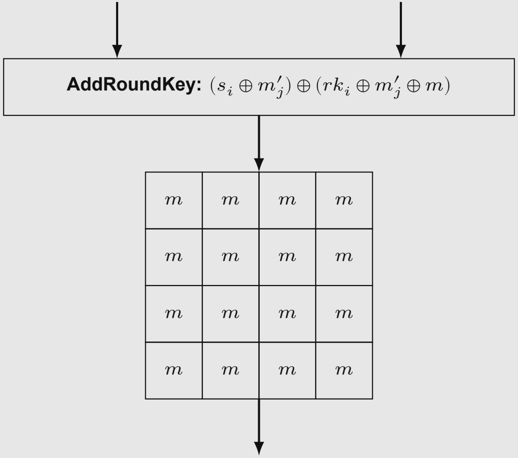
Masks at the Beginning of a Round

m'_1	m'_1	m'_1	m'_1
m'_2	m'_2	m'_2	m'_2
m'_3	m'_3	m'_3	m'_3
m'_4	m'_4	m'_4	m'_4

$m'_1 \oplus m$	$m'_1 \oplus m$	$m'_1 \oplus m$	$m'_1 \oplus m$
$m'_2 \oplus m$	$m'_2 \oplus m$	$m'_2 \oplus m$	$m'_2 \oplus m$
$m'_3 \oplus m$	$m'_3 \oplus m$	$m'_3 \oplus m$	$m'_3 \oplus m$
$m'_4 \oplus m$	$m'_4 \oplus m$	$m'_4 \oplus m$	$m'_4 \oplus m$

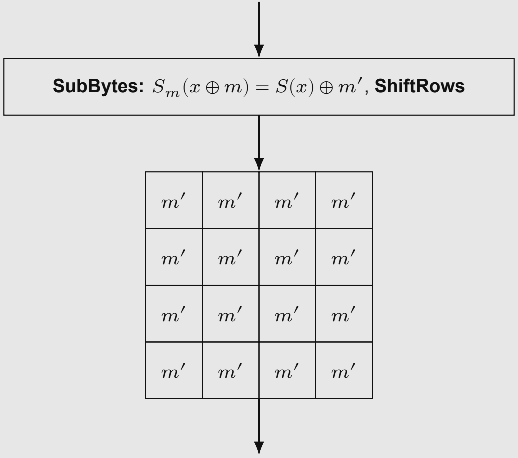
Example Masking on AES

Masks after AddRoundKey


AddRoundKey: $(s_i \oplus m'_j) \oplus (rk_i \oplus m'_j \oplus m)$
m m m m
m m m m
m m m m
m m m m

Example Masking on AES

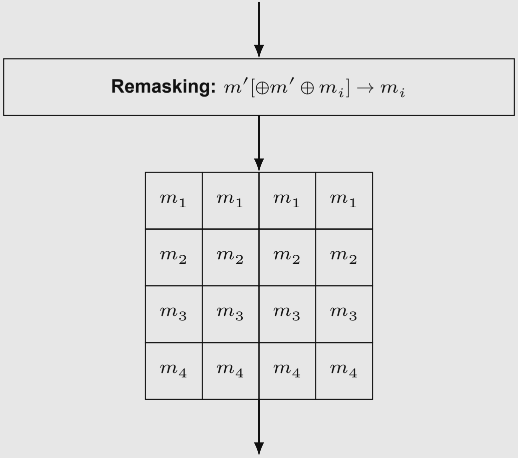
Masks after SubBytes (and ShiftRows)


SubBytes: $S_m(x \oplus m) = S(x) \oplus m'$, **ShiftRows**

m'	m'	m'	m'
m'	m'	m'	m'
m'	m'	m'	m'
m'	m'	m'	m'

Example Masking on AES

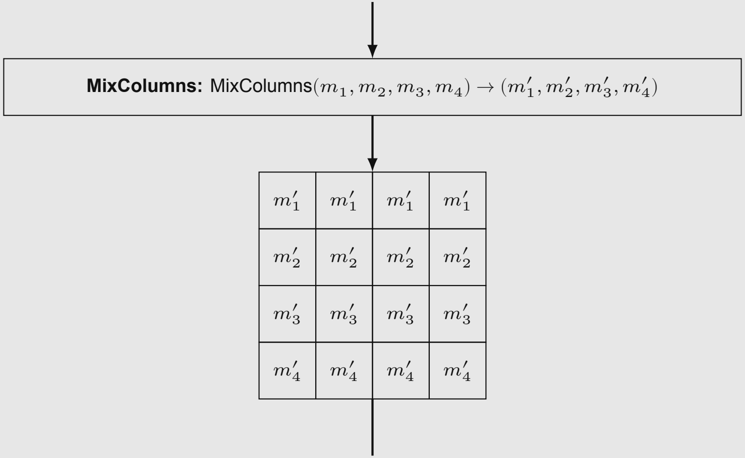
Masks after Remasking



m_1	m_1	m_1	m_1
m_2	m_2	m_2	m_2
m_3	m_3	m_3	m_3
m_4	m_4	m_4	m_4

Example Masking on AES

Masks after MixColumns



MixColumns: MixColumns$(m_1, m_2, m_3, m_4) \rightarrow (m'_1, m'_2, m'_3, m'_4)$

m'_1	m'_1	m'_1	m'_1
m'_2	m'_2	m'_2	m'_2
m'_3	m'_3	m'_3	m'_3
m'_4	m'_4	m'_4	m'_4

Evaluation

- The previous masking resists 1st-order attacks.
- It drastically increases the required traces for an attack.
- The runtime roughly doubles with the countermeasure.
- Most cycles are spent on the precomputations.

$\rightarrow$ Each AES operation requires new masks and thus new randomness.

Masking Pitfalls

- Mask re-usage or biased masks can be exploited.
- (SW) Order of operations is important.
    - Compilers may change the order for better performance.
    - Requires usage of special compilers/flags or of assembly directly.
- (HW) Parallel logic gates can leak information due to signal delays.

Countermeasures

Higher Order Attacks

General Description

Higher-order attacks use key hypothesis which are combinations of multiple points in the power trace (joint leakage).

Second-order: Attack both shares of a secret value or two usages of the same mask.

- Set up a combined hypothesis, e.g., HW$(x_m \oplus m = x)$ or HW$(x_m \oplus y_m = x \oplus y)$.
- Preprocess traces:
  - Identify the exact points of interest (occurrence of x_m and m or x_m and y_m respectively).
  - If the points are unknown, investigate all pairs of possible points (quadratic cost).
  - Combine the measurements on the pairs, e.g., using their product.
- Use standard DPA on preprocessed traces.