# Block Ciphers Implementations Provably Secure Against Second Order Side Channel Analysis

Matthieu Rivain[1,2], Emmanuelle Dottax[1] & Emmanuel Prouff[1]

Oberthur Card Systems

University of Luxembourg

February 11, 2008

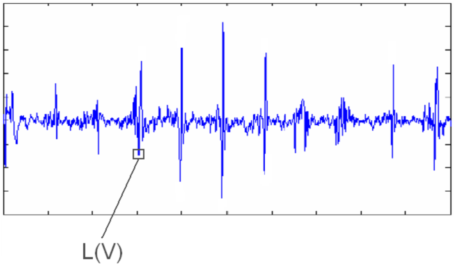
# Outline

# Side Channel Analysis

- Side Channel Analysis (SCA) is a strong cryptanalytic technique targeting physical implementations
- The physical leakage of the execution of any algorithm depends on the intermediate variables
- SCA exploits leakage on sensitive variables that depend on the secret key

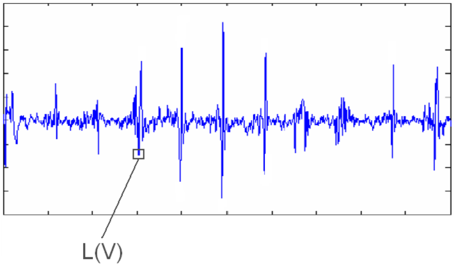
# Side Channel Analysis

- $V$ depends on a few key bits
  - $\Rightarrow$ possible key recovery attack exploiting $L(V)$



# Side Channel Analysis

- $V$ depends on a few key bits
  - $\Rightarrow$ possible key recovery attack exploiting $L(V)$




- Classical statistical distinguishers:
  - correlation techniques – generic
  - maximum likelihood – strong adversary model

## Masking & Higher Order SCA

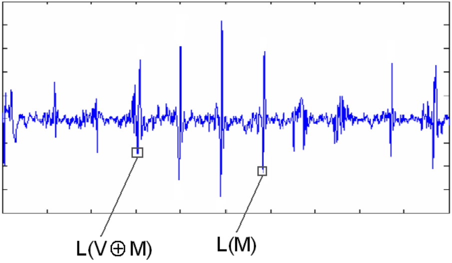
- One or several random values – the masks – are added to every sensitive variable

# Masking & Higher Order SCA

- One or several random values – the masks – are added to every sensitive variable
- First order masking: one single mask

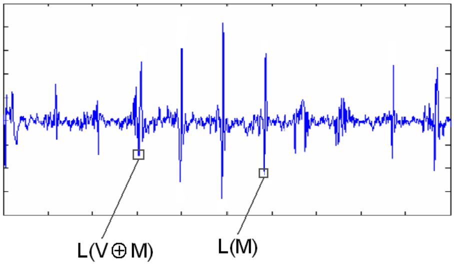
# Masking & Higher Order SCA

- One or several random values – the masks – are added to every sensitive variable
- First order masking: one single mask
- Second Order Side Channel Analysis
  - $M$ : random mask
  - $V \oplus M$ : masked variable



# Masking & Higher Order SCA

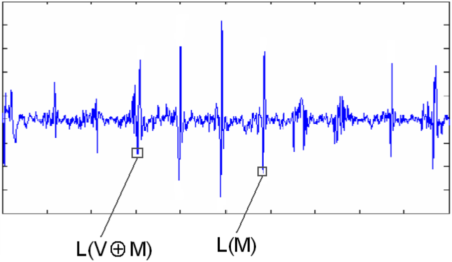
- One or several random values – the masks – are added to every sensitive variable
- First order masking: one single mask
- Second Order Side Channel Analysis
  - $M$ : random mask
  - $V \oplus M$ : masked variable




- To thwart 2O-SCA: use second order masking

# Masking & Higher Order SCA

- One or several random values – the masks – are added to every sensitive variable
- First order masking: one single mask
- Second Order Side Channel Analysis
  - $M$ : random mask
  - $V \oplus M$ : masked variable




- To thwart 2O-SCA: use second order masking
- $d^{\text{th}}$ order masking is broken by $(d+1)^{\text{th}}$ order SCA

# Why Using Masking ?

- [Chari+ CRYPTO'99] SCA complexity increases
  - exponentially with the masking order
  - polynomially with hiding-like countermeasures (noise addition, operation order randomization, ...)
- Incrementing the masking order is of great interest for SCA resistance

# Why Using Masking ?

- [Chari+ CRYPTO'99] SCA complexity increases
  - exponentially with the masking order
  - polynomially with hiding-like countermeasures (noise addition, operation order randomization, ...)
- Incrementing the masking order is of great interest for SCA resistance

- Many papers focus on improving 2O-SCA
- A few papers deal with resistant implementations

# Why Using Masking ?

- [Chari+ CRYPTO'99] SCA complexity increases
  - exponentially with the masking order
  - polynomially with hiding-like countermeasures (noise addition, operation order randomization, ...)
- Incrementing the masking order is of great interest for SCA resistance

- Many papers focus on improving 2O-SCA
- A few papers deal with resistant implementations

- First step: provable security against 2O-SCA

# Security Against 2O-SCA

**Definition (2O-SCA Security)**

A cryptographic algorithm is said to be **secure against 2O-SCA** if every pair of its intermediate variables is independent of any sensitive variable.

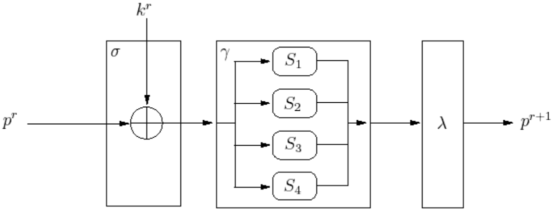
- An algorithm security can be formally proved
  - listing all intermediate variables
  - checking every pair independency

# Block Cipher Description

- Iterated block cipher

# Block Cipher Description

- Iterated block cipher
- Round transformation: $\rho[k](\cdot) = \lambda \circ \gamma \circ \sigma[k](\cdot)$



- Second order masking:
  - $p = p_0 \oplus p_1 \oplus p_2$
  - $k = k_0 \oplus k_1 \oplus k_2$

- $(p_1, p_2)$ and $(k_1, k_2)$ randomly generated

# Securing Block Ciphers Implementations

- Second order masking:
  - $p = p_0 \oplus p_1 \oplus p_2$
  - $k = k_0 \oplus k_1 \oplus k_2$
- $(p_1, p_2)$ and $(k_1, k_2)$ randomly generated
- Goal: perform a round transformation from the 3 shares
  - The shares must be process separately
  - The completeness relation must be preserved

# Securing the Round Transformation

- Linear layer: simple

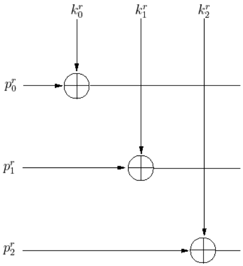
- Linear layer: $\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2)$

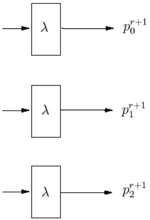
# Securing the Round Transformation

- Linear layer: $\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2)$
- Key addition layer: simple

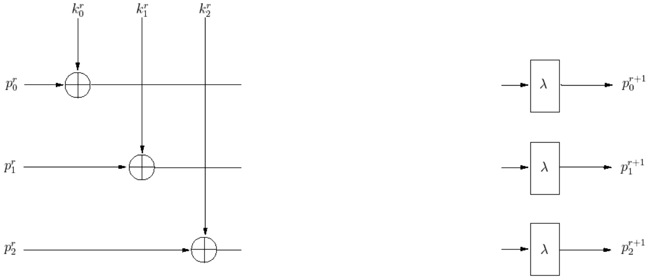
# Securing the Round Transformation

- Linear layer: $\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2)$
- Key addition layer: $\sigma[k](p) = \sigma[k_0](p_0) \oplus \sigma[k_1](p_1) \oplus \sigma[k_2](p_2)$



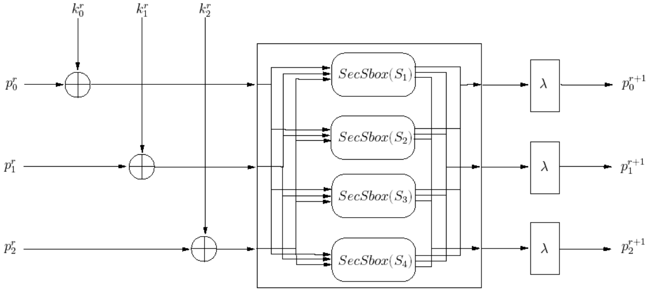



- Linear layer: $\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2)$
- Key addition layer: $\sigma[k](p) = \sigma[k_0](p_0) \oplus \sigma[k_1](p_1) \oplus \sigma[k_2](p_2)$
- Non-linear layer: issue



# Securing the Round Transformation

- Linear layer: $\lambda(p) = \lambda(p_0) \oplus \lambda(p_1) \oplus \lambda(p_2)$
- Key addition layer: $\sigma[k](p) = \sigma[k_0](p_0) \oplus \sigma[k_1](p_1) \oplus \sigma[k_2](p_2)$
- Non-linear layer: issue
  - Problem: secure an S-box implementation



Oberthur Card Systems

# Secure S-box Implementation – Problem

- $S$ : $n \times m$ S-box

# Secure S-box Implementation – Problem

- $S$ : $n \times m$ S-box
- $\tilde{x} = x \oplus r_1 \oplus r_2$ : $n$-bit masked input, $(r_1, r_2)$ : $n$-bit input masks

# Secure S-box Implementation – Problem

- $S$ : $n \times m$ S-box
- $\tilde{x} = x \oplus r_1 \oplus r_2$ : $n$-bit masked input, $(r_1, r_2)$ : $n$-bit input masks
- $(s_1, s_2)$ : $m$-bit output masks

# Secure S-box Implementation – Problem

- $S$ : $n \times m$ S-box
- $\tilde{x} = x \oplus r_1 \oplus r_2$ : $n$-bit masked input, $(r_1, r_2)$ : $n$-bit input masks
- $(s_1, s_2)$ : $m$-bit output masks
- Goal : process $S(x) \oplus s_1 \oplus s_2$
- Requirement : every pair of inter. var. must be indep. of $x$

# Our Proposition

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
4. $\quad a' \leftarrow a \oplus r'$
5. $\quad T[a'] \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
6. **return** $T[r_3]$

# Our Proposition

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
4. $\quad a' \leftarrow a \oplus r'$
5. $\quad T[a'] \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
6. **return** $T[r_3]$

- When $a = r_1 \oplus r_2$ :
  - $\tilde{x} \oplus a = x$ – desired masked output

# Our Proposition

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
4. $\quad a' \leftarrow a \oplus r'$
5. $\quad T[a'] \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
6. **return** $T[r_3]$

- When $a = r_1 \oplus r_2$ :
  - $\tilde{x} \oplus a = x$ – desired masked output
  - $a' = r_3$ – stored in $T[r_3]$

# Our Proposition

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $r_3 \leftarrow rand(n)$
2. $r' \leftarrow (r_1 \oplus r_3) \oplus r_2$
3. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
4. $\quad a' \leftarrow a \oplus r'$
5. $\quad T[a'] \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
6. **return** $T[r_3]$

- When $a = r_1 \oplus r_2$ :
  - $\tilde{x} \oplus a = x$ – desired masked output
  - $a' = r_3$ – stored in $T[r_3]$
- Every pair of inter. var. is indep. of $x$

Oberthur Card Systems

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
2. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
3. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
4. **return** $R_0$

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
2. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
3. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
4. **return** $R_0$

- When $a = r_1 \oplus r_2$:
  - $\tilde{x} \oplus a = x$ – desired masked output

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
2. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
3. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
4. **return** $R_0$

- When $a = r_1 \oplus r_2$:
  - $\tilde{x} \oplus a = x$ – desired masked output
  - $cmp = 0$ – stored in $R_0$

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
2. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
3. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
4. **return** $R_0$

- When $a = r_1 \oplus r_2$:
  - $\tilde{x} \oplus a = x$ – desired masked output
  - $cmp = 0$ – stored in $R_0$
- However there is a flaw: $(cmp, \tilde{x} \oplus a)$ depends on $x$!

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
2. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
3. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
4. **return** $R_0$

# Another Proposition

- $compare(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
2. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare(a \oplus r_1, r_2)$
4. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
5. **return** $R_0$

# Another Proposition

- $compare_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
2. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare_b(a \oplus r_1, r_2)$
4. $\quad R_{cmp} \leftarrow \left(S(\tilde{x} \oplus a) \oplus s_1\right) \oplus s_2$
5. **return** $R_0$

# Another Proposition

- $compare_b(x,y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
2. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare_b(a \oplus r_1, r_2)$
4. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
5. **return** $R_b$

# Another Proposition

- $compare_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
2. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare_b(a \oplus r_1, r_2)$
4. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
5. **return** $R_b$

- The security relies on the $compare_b$ implementation

# Another Proposition

- $compare_b(x, y) = \begin{cases} b & \text{if } x = y \\ \bar{b} & \text{if } x \neq y \end{cases}$

**Input:** $\tilde{x} = x \oplus r_1 \oplus r_2$, $(r_1, r_2)$, $(s_1, s_2)$
**Output:** $S(x) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
2. **for** $a$ **from** $0$ **to** $2^n - 1$ **do**
3. $\quad cmp \leftarrow compare_b(a \oplus r_1, r_2)$
4. $\quad R_{cmp} \leftarrow \big(S(\tilde{x} \oplus a) \oplus s_1\big) \oplus s_2$
5. **return** $R_b$

- The security relies on the $compare_b$ implementation
- Less efficient than the previous solution but less memory consuming

# Improvement

- Both methods process a loop on every possible S-box output
- Improvement: process several S-box outputs at the same time

Oberthur Card Systems

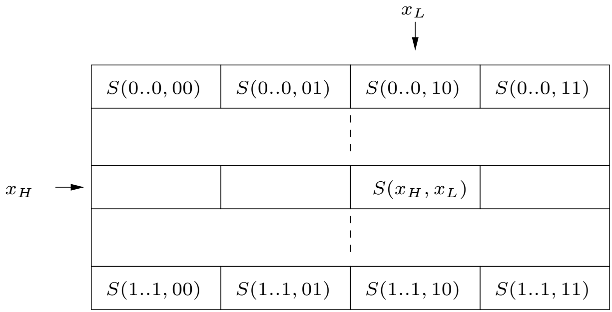
# Improvement

- Both methods process a loop on every possible S-box output
- Improvement: process several S-box outputs at the same time
  - *e.g.* 4 S-box outputs can be stored in one $\mu$P word

| | | | |
|---|---|---|---|
| $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| ⋮ | | | |
| $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

# Improvement

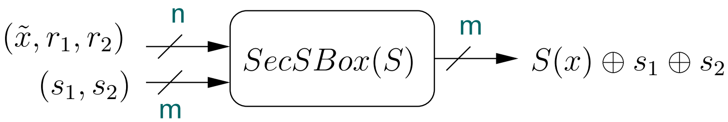
- Both methods process a loop on every possible S-box output
- Improvement: process several S-box outputs at the same time
  - *e.g.* 4 S-box outputs can be stored in one $\mu$P word




- $S'(x_H) = \big(S(x_H, 00), S(x_H, 01), S(x_H, 10), S(x_H, 11)\big)$

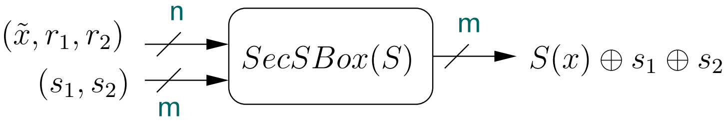
# Improvement

- Without improvement – $S$ : $n \times m$ S-box

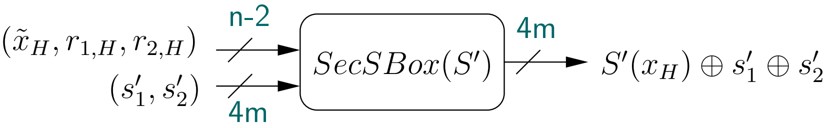


# Improvement

- Without improvement – $S$ : $n \times m$ S-box

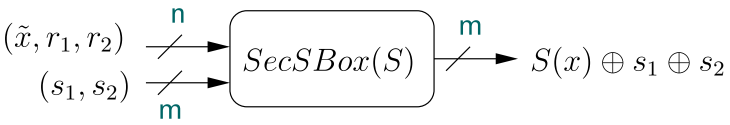



- With improvement – $S'$ : $(n-2) \times 4m$ S-box

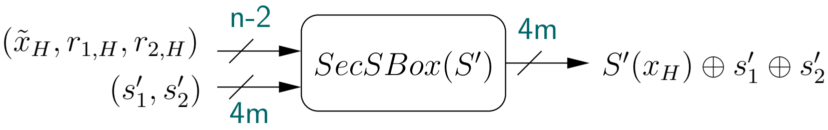


# Improvement

- Without improvement – $S$ : $n \times m$ S-box



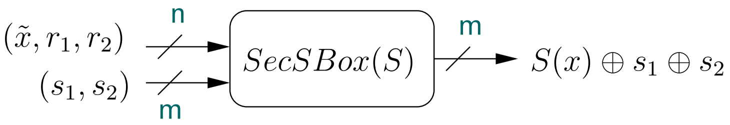

- With improvement – $S'$ : $(n-2) \times 4m$ S-box



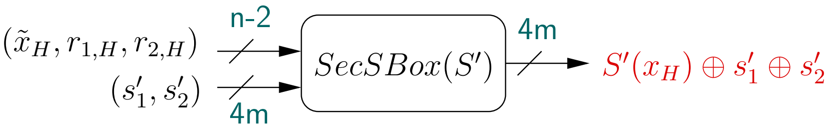

  - 4 times faster !

# Improvement

- Without improvement – $S : n \times m$ S-box




- With improvement – $S' : (n-2) \times 4m$ S-box




  - 4 times faster !
  - Returns the whole line of the matrix containing the masked output

# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | | | | |
| $x_H$ → | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | | | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

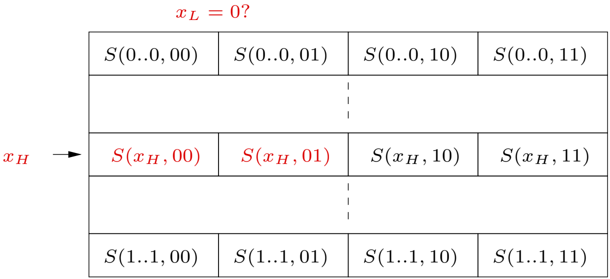
# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | | ⋮ | | |
| $x_H$ → | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | | ⋮ | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$



# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$

$x_L = 1?$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | | ⋮ | | |
| $x_H \rightarrow$ | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | | ⋮ | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$

$x_L = 00$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | ⋮ | | | |
| $x_H$ → | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | ⋮ | | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$

$x_L = 01$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | | | | |
| $x_H$ → | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | | | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

Oberthur Card Systems

# Improvement

- Returned value: $S'(x_H) \oplus s'_1 \oplus s'_2$
- Second step: extract masked $S(x) \oplus s_1 \oplus s_2$
  - Requires a *Select* algorithm which from a masked bit securely selects the corresponding half

$x_L = 01$

| | | | | |
|---|---|---|---|---|
| | $S(0..0, 00)$ | $S(0..0, 01)$ | $S(0..0, 10)$ | $S(0..0, 11)$ |
| | | ⋮ | | |
| $x_H$ → | $S(x_H, 00)$ | $S(x_H, 01)$ | $S(x_H, 10)$ | $S(x_H, 11)$ |
| | | ⋮ | | |
| | $S(1..1, 00)$ | $S(1..1, 01)$ | $S(1..1, 10)$ | $S(1..1, 11)$ |

# Schramm & Paar Solution (CT-RSA 2006)

- Computation of a masked S-box :

$$S^{\star}(y) = S(y \oplus r_1 \oplus r_2) \oplus s_1 \oplus s_2$$

# Schramm & Paar Solution (CT-RSA 2006)

- Computation of a masked S-box :

$$S^{\star}(y) = S(y \oplus r_1 \oplus r_2) \oplus s_1 \oplus s_2$$

- Schramm & Paar 1:
  - Two table re-computations

# Schramm & Paar Solution (CT-RSA 2006)

- Computation of a masked S-box :

$$S^\star(y) = S(y \oplus r_1 \oplus r_2) \oplus s_1 \oplus s_2$$

- Schramm & Paar 1:
  - Two table re-computations
- Schramm & Paar 2:
  - Involves the last masked S-box
  - One single table re-computation
  - Potential flaws for straightforward implementation

# Schramm & Paar Solution (CT-RSA 2006)

- Computation of a masked S-box :

$$S^\star(y) = S(y \oplus r_1 \oplus r_2) \oplus s_1 \oplus s_2$$

- Schramm & Paar 1:
  - Two table re-computations
- Schramm & Paar 2:
  - Involves the last masked S-box
  - One single table re-computation
  - Potential flaws for straightforward implementation
- Compared to our solutions:
  - Fewer operations
  - More memory

# AES implementations

| Solution | cycles | RAM (bytes) | ROM (bytes) |
|---|---|---|---|
| Schramm & Paar 1 | $1083 \times 10^3$ | $512 + 86$ | 2247 |
| Schramm & Paar 2 | $594 \times 10^3$ | $512 + 90$ | 2336 |
| Our solution | $672 \times 10^3$ | $256 + 86$ | 2215 |

AES implementations secure against 2O-DSCA on an 8-bit microcontroller

# $8 \times 8$ S-box Implementations

| Solution | Cycles | RAM (bytes) | ROM (bytes) |
|---|---|---|---|
| 8-bit architecture | | | |
| Schramm & Paar 1 | 6703 | $512 + 3$ | $119 + 256$ |
| Schramm & Paar 2 | 3638 | $512 + 7$ | $89 + 256$ |
| Our solution | 4142 | $256 + 3$ | $88 + 256$ |
| 16-bit architecture | | | |
| Schramm & Paar 1 | 6418 | 512 | $96 + 512$ |
| Schramm & Paar 2 | 3090 | 512 | $56 + 256$ |
| Our solution | 4125 | 256 | $98 + 512$ |
| 32-bit architecture | | | |
| Schramm & Paar 2 | 3359 | 512 | na. |
| Our solution | 4143 | 256 | na. |

Comparison of $8 \times 8$ S-box implementations secure against 2O-SCA on 8-bit, 16-bit and 32-bit architectures.

Oberthur Card Systems

# $8 \times 8$ S-box Implementations

| Solution | Cycles | RAM (bytes) | ROM (bytes) |
|---|---|---|---|
| 8-bit architecture | | | |
| Schramm & Paar 1 | 6703 | $512 + 3$ | $119 + 256$ |
| Schramm & Paar 2 | 3638 | $512 + 7$ | $89 + 256$ |
| Our solution | 4142 | $256 + 3$ | $88 + 256$ |
| 16-bit architecture | | | |
| Schramm & Paar 1 | 6418 | 512 | $96 + 512$ |
| Schramm & Paar 2 | 3090 | 512 | $56 + 256$ |
| Our solution | 4125 | 256 | $98 + 512$ |
| Our improved solution | 2099 | 256 | $260 + 256$ |
| 32-bit architecture | | | |
| Schramm & Paar 2 | 3359 | 512 | na. |
| Our solution | 4143 | 256 | na. |
| Our improved solution | 1415 | 256 | na. |

Comparison of $8 \times 8$ S-box implementations secure against 2O-SCA on 8-bit, 16-bit and 32-bit architectures.

# Conclusion

- Block ciphers implementations provably secure against 2O-SCA
- Two new methods to secure S-box implementations against 2O-SCA
- Our solutions allow different efficiency/memory trade-offs
- Improvement when several S-box outputs can be stored on one microprocessor word
- The security of all our propositions is formally demonstrated