Masking - Defeating Power Analysis Attacks

Side-Channel Security

Rishub Nagpal

June 6, 2023

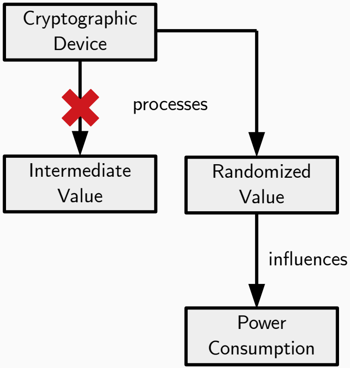
IAIK – Graz University of Technology

Overview

Recap

We want to...

- Operate on randomized intermediate values
- But still require correct algorithm output



Secret Sharing Intuition

- Compute f on input x and secret s ...
  - But avoid using s directly

$$f(x, s) = y$$

Secret Sharing Intuition

- Compute f on input x and secret $s \ldots$
  - But avoid using s directly
- Idea: Split s into e.g. 3 shares s_1, s_2, s_3 such that:
  - $s = s_1 \circ s_2 \circ s_3$
  - Individual shares do not reveal s
  - Each 2-combination of shares does not reveal s
  - The computed y_1, y_2, y_3 can be combined to y

$$f(x, s) = y$$

$$f(x_1, s_1) = y_1$$

$$f(x_2, s_2) = y_2$$

$$f(x_3, s_3) = y_3$$

$$y = y_1 \circ y_2 \circ y_3$$

Secret Sharing Intuition

- Compute f on input x and secret $s\dots$
  - But avoid using s directly
- Idea: Split s into e.g. 3 shares s_1, s_2, s_3 such that:
  - $s = s_1 \circ s_2 \circ s_3$
  - Individual shares do not reveal s
  - Each 2-combination of shares does not reveal s
  - The computed y_1, y_2, y_3 can be combined to y
- For technical reasons:
  - Split x into 3 shares x_1, x_2, x_3 as well

$$f(x, s) = y$$

$$f(x_1, s_1) = y_1$$

$$f(x_2, s_2) = y_2$$

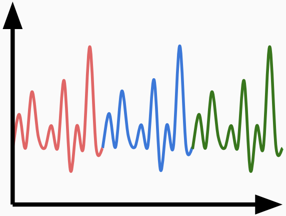
$$f(x_3, s_3) = y_3$$

$$y = y_1 \circ y_2 \circ y_3$$

- Application to crypto operations:
  - Split key k into k_1, k_2, k_3
  - Split plaintext x into x_1, x_2, x_3
  - Compute ciphertext $y = y_1 \circ y_2 \circ y_3$
  - (Use new shares for each encryption!)

$$\mathrm{enc}(x_1, k_1) = y_1$$
$$\mathrm{enc}(x_2, k_2) = y_2$$
$$\mathrm{enc}(x_3, k_3) = y_3$$
$$y = y_1 \circ y_2 \circ y_3$$

- We do secret sharing on one device and multiple shares of the key k: $k_1, k_2, k_3 \rightarrow$



Masking an algorithm

How to mask ?

1. Write your computation an algebraic circuit.
2. Share the inputs.
3. Implement the circuit, replacing gates with masked gadgets.
4. Unmask the result.

Masking an algorithm

Inputs sharing

Sharing a bit

Input: x.

1: $x_0 \leftarrow x$
2: **for** $i = 1$ to $d-1$ **do**
3: $\quad x_i \xleftarrow{\$} \mathbb{F}_2$
4: $x_0 \leftarrow x \oplus \bigoplus_{i=1}^{d-1} x_i$

Output: $(x_0, \ldots, x_{d-1})$.

If x is sensitive, run ahead of time.

To not reuse a sharing: refresh

Input: $(x_0, \ldots, x_{d-1})$.

1: **for** $i = 1$ to $d - 1$ **do**
2: $\quad r_i \xleftarrow{\$} \mathbb{F}_2$
3: $r_0 \leftarrow \bigoplus_{i=1}^{d-1} r_i$
4: **for** $i = 0$ to $d - 1$ **do**
5: $\quad y_i \leftarrow x_i \oplus r_i$

Output: $(y_0, \ldots, y_{d-1})$.

Masking an algorithm

Masking a circuit

Masked XOR Gate

Input: $(x_0, \ldots, x_{d-1}, (y_0, \ldots, y_{d-1})$.

for $i = 0$ to $d - 1$ **do**

$\quad z_i \leftarrow x_i \oplus y_i$

Output: $(z_0, \ldots, z_{d-1})$.

Masked NOT Gate

Input: $(x_0, \ldots, x_{d-1})$

1: $y_0 \leftarrow \neg x_0$
2: **for** $i = 1$ to $d - 1$ **do**
3: $\quad y_i \leftarrow x_i$

Output: $(y_0, \ldots, y_{d-1})$.

Masked AND Gate: ISW multiplication

Input: $(x_0, \ldots, x_{d-1}, (y_0, \ldots, y_{d-1})$.

for $i = 0$ to $d-1$ **do**
    for $j = i+1$ to $d-1$ **do**
        $r_{ij} \xleftarrow{\$} \mathbb{F}_2,\ r_{ji} \leftarrow r_{ij}$
for $i = 0$ to $d-1$ **do**
    for $j = 0$ to $d-1$ **do**
        $p_{ij} \leftarrow x_i \odot y_j$
        if $i \neq j$ **then**
            $t_{ij} \leftarrow p_{ij} \oplus r_{ij}$
        else
            $t_{ij} \leftarrow p_{ij}$
for $i = 0$ to $d-1$ **do**
    $z_i = \bigoplus_{j=0}^{d-1} t_{ij}$

Output: $(z_0, \ldots, z_{d-1})$.

Other masked gates?

AND+NOT would be enough, but not efficient.

Other masked gates?

AND+NOT would be enough, but not efficient.

Other 2-input gates: NAND, OR, NOR?

Other masked gates?

AND+NOT would be enough, but not efficient.

Other 2-input gates: NAND, OR, NOR? De Morgan laws

Other masked gates?

AND+NOT would be enough, but not efficient.

Other 2-input gates: NAND, OR, NOR? De Morgan laws

More than 2 inputs?

Other masked gates?

AND+NOT would be enough, but not efficient.

Other 2-input gates: NAND, OR, NOR? De Morgan laws

More than 2 inputs? Why not? Challenging to make efficient.

Composition problem

What happens when we connect gadgets together in a larger circuit?

Are we still “secure” ?

Composition problem

What happens when we connect gadgets together in a larger circuit?

Are we still “secure” ?

Masking security: t-probing model:

A circuit is t-probing secure if any observation of t wires in the circuit is independent of the secret (unmasked) inputs.

Masking an algorithm

Unmask the result

XOR shares together :)

Beyond Boolean masking

Other masking approaches

- Arithmetic masking in $\mathbb{F}_{2^n}$, $\mathbb{Z}_n$
- Table-based masking
- Threshold implementations
- Code-based masking

Other security models:

- region probing model
- random probing model
- noise leakage model

Masking in practice: Hardware implementations

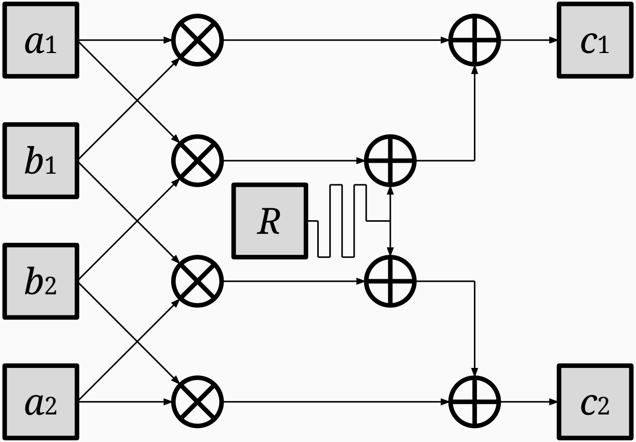
Hardware leakage

- Value overwriting: load x_0 then x_1.
- Transition leakage $\sim x_0 \oplus x_1$.
- To be avoided! $\rightarrow$ Hardwired "domains".

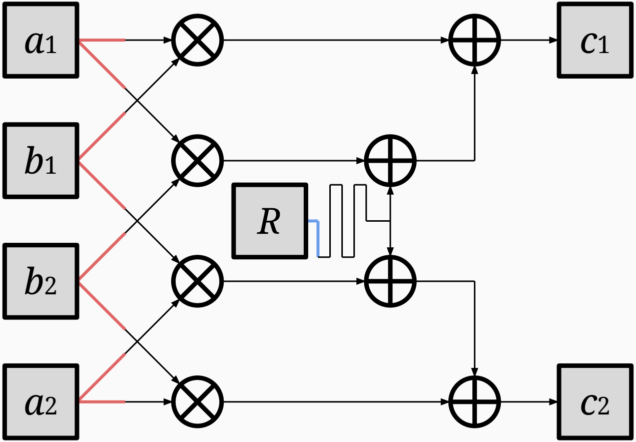
- In HW multiple operations are performed in a single clock cycle
- Logic gates cause a certain delay of the signal
- Propagation of signals in a combinatorial logic can lead to "glitches"
  - Ephemeral incorrect computations
  - Leakage

Modelled in the *robust t-probing model*.

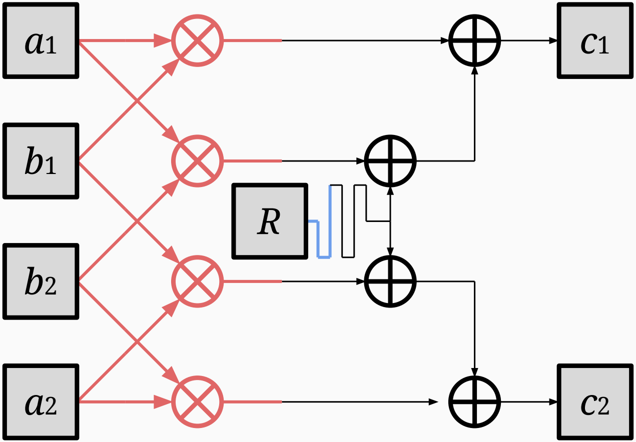
Example: Glitches in first-order ISW AND gadget



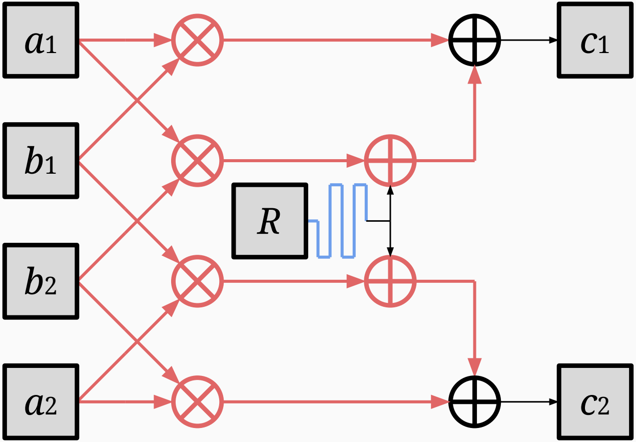
Example: Glitches in first-order ISW AND gadget



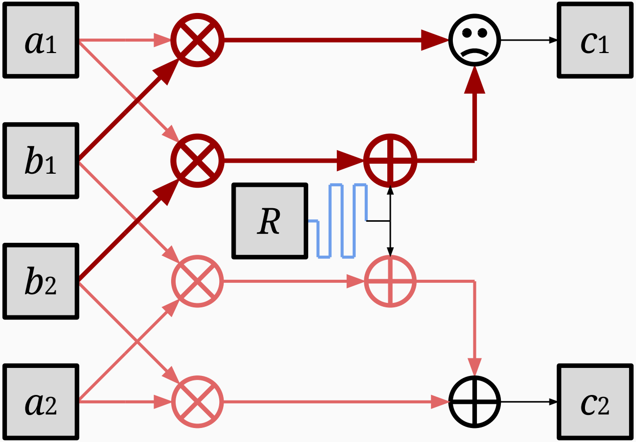
Example: Glitches in first-order ISW AND gadget



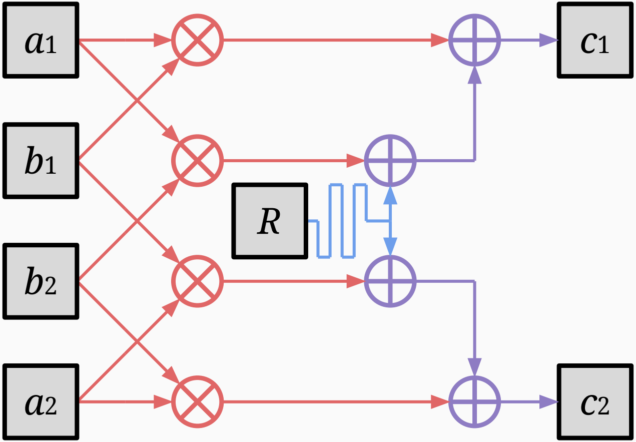
Example: Glitches in first-order ISW AND gadget



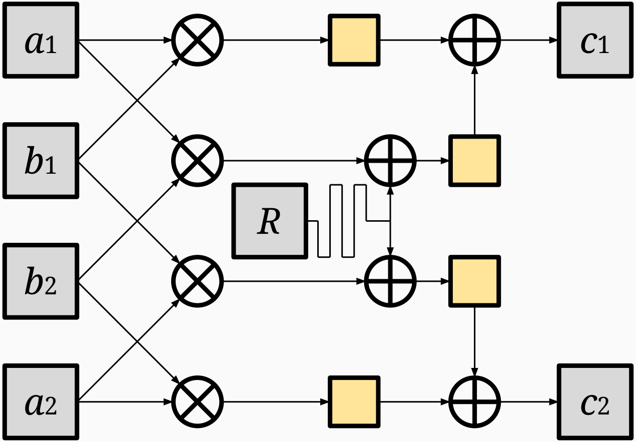
Example: Glitches in first-order ISW AND gadget



Example: Glitches in first-order ISW AND gadget



Example: Glitches in first-order ISW AND gadget → DOM gadget



Masking in practice: Software implementations

- If you write C-code:
  - Compilers can reorder instructions as long as logic is the same.
  - Compilers can change logic as long as result is the same.
  - Write assembly instead.

Masking in software

- If you write C-code:
  - Compilers can reorder instructions as long as logic is the same.
  - Compilers can change logic as long as result is the same.
  - Write assembly instead.
- A processor is still hardware...
  - Transitions:

```
STORE R1 x0
STORE R1 x1
```

- If you write C-code:
  - Compilers can reorder instructions as long as logic is the same.
  - Compilers can change logic as long as result is the same.
  - Write assembly instead.
- A processor is still hardware...
  - Transitions:

    ```
    STORE R1 x0
    STORE R1 x1
    ```
  - Glitches: less of a problem.
  - Countermeasures
    - "Lazy engineering": double number of shares.
    - + "Only one share in the processor"
    - ...

- At every point in time during encryption the processed data is random

Recap - Masking

- At every point in time during encryption the processed data is random
- That data still leaks but has no correlation the original (unmasked) data

Recap - Masking

- At every point in time during encryption the processed data is random
- That data still leaks but has no correlation the original (unmasked) data
- $\Rightarrow$ First-order power analysis does not work anymore

Recap - Masking

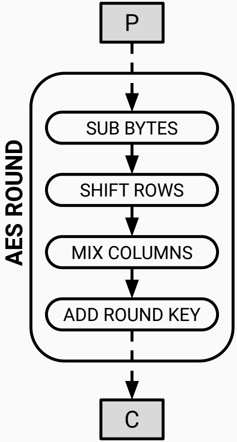
- At every point in time during encryption the processed data is random
- That data still leaks but has no correlation the original (unmasked) data
- $\Rightarrow$ First-order power analysis does not work anymore
- Attacker could now consider combinations of points in the power trace

Recap - Masking

- At every point in time during encryption the processed data is random
- That data still leaks but has no correlation the original (unmasked) data
- $\Rightarrow$ First-order power analysis does not work anymore
- Attacker could now consider combinations of points in the power trace
- How to choose the masking order? Depends on noise, etc.
  - Security: $1/\mathrm{SNR}^d$ (provable – but tricky).
  - Cost: $\mathcal{O}(\mathrm{d}^2)$ (for non-linear gadgets).
  - Deployed: 1st and 2nd order masking (?).
  - Practically-relevant order increase (stronger attacks, PQ Crypto).

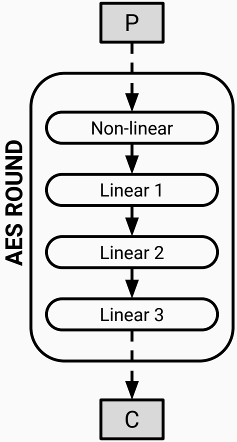
Masking AES

- AES consists of iterative application of 4 functions
  - In case of AES-128 we have 10 (+1 initial) rounds
  - Initial/final rounds are smaller



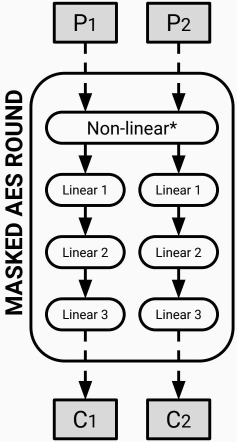
Masking - Application to AES

- AES consists of iterative application of 4 functions
  - In case of AES-128 we have 10 (+1 initial) rounds
  - Initial/final rounds are smaller
- Identify linear/non-linear functions

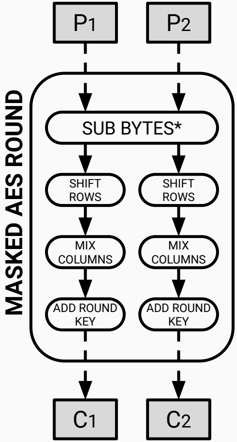


Masking - Application to AES

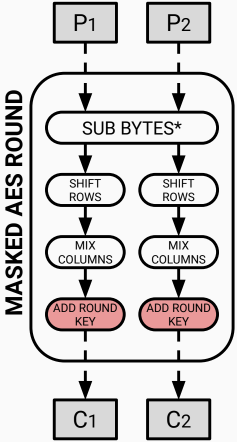
- AES consists of iterative application of 4 functions
  - In case of AES-128 we have 10 (+1 initial) rounds
  - Initial/final rounds are smaller
- Identify linear/non-linear functions
- Split computation into shares accordingly
  - XOR each byte of P with randomness → P1, P2
  - Calculate functions on shares
  - Pairwise XOR each byte of C1, C2 → C



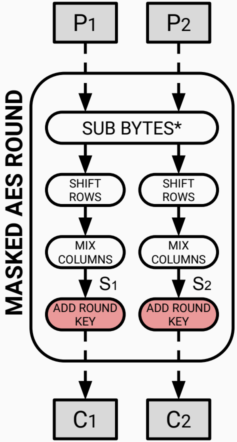
- AES consists of iterative application of 4 functions
    - In case of AES-128 we have 10 (+1 initial) rounds
    - Initial/final rounds are smaller
- Identify linear/non-linear functions
- Split computation into shares accordingly
    - XOR each byte of P with randomness → P1, P2
    - Calculate functions on shares
    - Pairwise XOR each byte of C1, C2 → C
- Done?



- AES consists of iterative application of 4 functions
  - In case of AES-128 we have 10 (+1 initial) rounds
  - Initial/final rounds are smaller
- Identify linear/non-linear functions
- Split computation into shares accordingly
  - XOR each byte of P with randomness → P_1, P_2
  - Calculate functions on shares
  - Pairwise XOR each byte of C_1, C_2 → C
- Done?

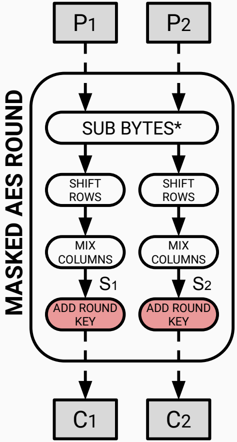


- Takes two inputs: State & Key
  - State (S1,S2) is already shared, key is not
  - XOR-ing key to both shares cancels out!
  - $C_1 = S_1 \oplus K$
    $C_2 = S_2 \oplus K$
    $C = C_1 \oplus C_2 = (S_1 \oplus K) \oplus (S_2 \oplus K) = S_1 \oplus S_2$



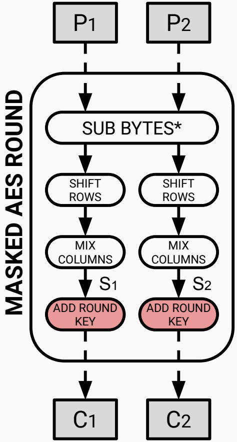
Be careful with ADD ROUNDKEY!

- Takes two inputs: State & Key
  - State (S1,S2) is already shared, key is not
  - XOR-ing key to both shares cancels out!
  - $C_1 = S_1 \oplus K$
    $C_2 = S_2 \oplus K$
    $C = C_1 \oplus C_2 = (S_1 \oplus K) \oplus (S_2 \oplus K) = S_1 \oplus S_2$
- Solution 1: XOR key only to one share
  - Works... but defies the purpose of masking



Be careful with ADD ROUNDKEY!

- Takes two inputs: State & Key
  - State (S1,S2) is already shared, key is not
  - XOR-ing key to both shares cancels out!
  - $C_1 = S_1 \oplus K$
    $C_2 = S_2 \oplus K$
    $C = C_1 \oplus C_2 = (S_1 \oplus K) \oplus (S_2 \oplus K) = S_1 \oplus S_2$
- Solution 1: XOR key only to one share
  - Works... but defies the purpose of masking
- Solution 2: XOR shared key to both shares
  - Actually works
  - Where do we get a shared key?



- Easier way:
  - Precompute all round keys
  - Split them into shares, store them
  - Requires lots of memory (problematic for μC, ASIC)

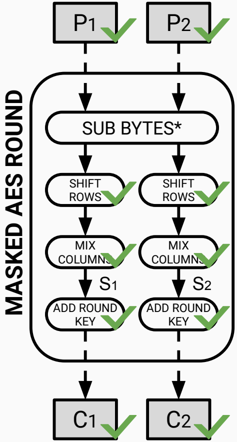
Masking - How to Handle Keys?

- Easier way:
    - Precompute all round keys
    - Split them into shares, store them
    - Requires lots of memory (problematic for μC, ASIC)
- Harder way:
    - Calculate rounds keys on the fly...

Masking - AES Key Schedule

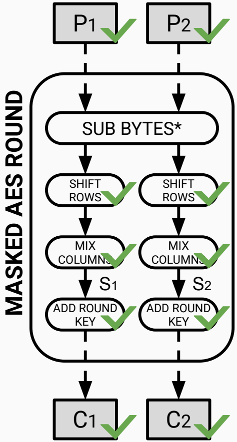
- Operates on the 128-bit key state (4x4 bytes)
- Consists of:
  - ROT WORD (one-byte left circular shift in one 4-byte array)
  - SUB WORD (SUB BYTES applied to one 4-byte array)
  - RCON (XOR of 4-byte round constant)

- We know how to:
  - Split inputs into shares
  - Calculate linear functions
  - Handle keys
  - Recover output



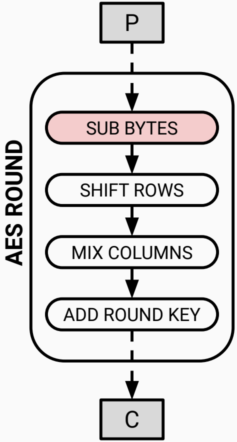
Masking AES - Recap so far

- We know how to:
  - Split inputs into shares
  - Calculate linear functions
  - Handle keys
  - Recover output
- Remaining:
  - Calculate non-linear functions

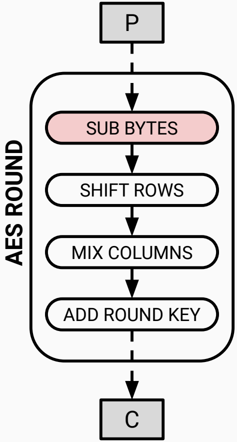


SUB BYTES*

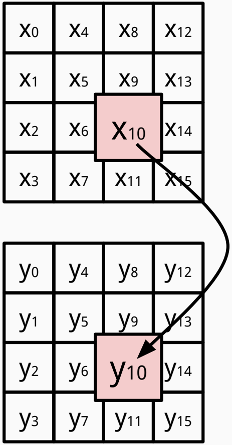
- AES would be a linear function without SUB BYTES



- AES would be a linear function without SUB BYTES
- Attack:
  - Setup equation system that relates key bits to P and C
  - Collect pairs of P and C
  - Solve system using Gaussian elimination



- Implementation: SUB BYTES $(x) = y$
  - Table-lookup
  - One byte (8-bits) input/output
  - Performed for each byte of the state



SUB BYTES

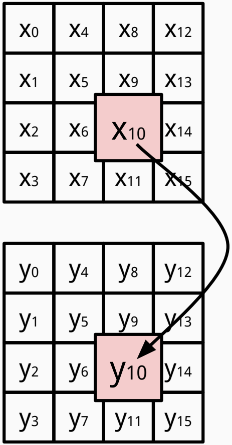
	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.a	.b	.c	.d	.e	.f
0.	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1.	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2.	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3.	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4.	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5.	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6.	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7.	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8.	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9.	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a.	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b.	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c.	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d.	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e.	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f.	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Masking SUB BYTES Table Lookup

- Desired behavior:

$$x = x_1 \oplus x_2$$
$$\text{SUB BYTES}(x_1, x_2) = (y_1, y_2)$$
$$y = y_1 \oplus y_2$$

  - Fix one share, precompute lookup table for the other share
- This approach is not so popular anymore...
  - Requires pre-calculation of 16 tables each round
  - Memory demanding



Masking SUB BYTES the Other Way...

- Desired behavior:

$$x = x_1 \oplus x_2$$
$$\text{SUB BYTES}(x_1, x_2) = (y_1, y_2)$$
$$y = y_1 \oplus y_2$$

- Find out algebraic description of SUB BYTES
- Implement it using ordinary mathematical operations
- Mask those...

1. Interpret 8-bit input as polynomial over GF(2)

Algebraic Description of SUB BYTES

1. Interpret 8-bit input as polynomial over GF(2)
2. Calculate its multiplicative inverse in $GF(2^8)$

1. Interpret 8-bit input as polynomial over GF(2)
2. Calculate its multiplicative inverse in $GF(2^8)$
3. Transform the inverse using an affine transformation

Algebraic Description of SUB BYTES

1. Interpret 8-bit input as polynomial over GF(2)
2. Calculate its multiplicative inverse in GF(2^8)
3. Transform the inverse using an affine transformation
4. Interpret resulting polynomial as 8-bit output

Algebraic Description of SUB BYTES

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

1. Interpret 8-bit input as polynomial over GF(2)
   - GF(2) = Galois Field(2) = Finite Field with two elements (0,1)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

1. Interpret 8-bit input as polynomial over GF(2)
   - GF(2) = Galois Field(2) = Finite Field with two elements (0,1)
   - Input (hex): 0xee
   - Input (bin): 0b11101110

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

1. Interpret 8-bit input as polynomial over GF(2)
   - GF(2) = Galois Field(2) = Finite Field with two elements (0,1)
   - Input (hex): 0xee
   - Input (bin): 0b11101110
   - $1x^7 + 1x^6 + 1x^5 + 0x^4 + 1x^3 + 1x^2 + 1x + 0$

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

1. Interpret 8-bit input as polynomial over GF(2)
    - GF(2) = Galois Field(2) = Finite Field with two elements (0,1)
    - Input (hex): 0xee
    - Input (bin): 0b11101110
    - $1x^7 + 1x^6 + 1x^5 + 0x^4 + 1x^3 + 1x^2 + 1x + 0$
    - $\mathbf{x} = x^7 + x^6 + x^5 + x^3 + x^2 + x$

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in $\mathrm{GF}(2^8)$
    - $\mathrm{GF}(2^8)$ = Finite Field with 256 elements (degree 7 polynomials, binary coefficients)

Algebraic Description of SUB BYTES (1)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in $GF(2^8)$
   - $GF(2^8)$ = Finite Field with 256 elements (degree 7 polynomials, binary coefficients)
   - Multiplicative inverse $\mathbf{x}^{-1}$ satisfies: $\mathbf{x} \times \mathbf{x}^{-1} = 1$

Algebraic Description of SUB BYTES (1)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in $GF(2^8)$
   - $GF(2^8)$ = Finite Field with 256 elements (degree 7 polynomials, binary coefficients)
   - Multiplicative inverse $\mathbf{x}^{-1}$ satisfies: $\mathbf{x} \times \mathbf{x}^{-1} = 1$
   - One small problem: 0 has no inverse, hence we simply map 0 to 0

Algebraic Description of SUB BYTES (2)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in GF(2^8)
   - $\mathbf{x}^{-1}$ is calculated, e.g., via $\mathbf{x}^{254}$ since $\mathbf{x} \times \mathbf{x}^{254} = \mathbf{x}^{255} = 1$ in GF(2^8) (Fermat's little theorem)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in $GF(2^8)$
   - $\mathbf{x}^{-1}$ is calculated, e.g., via $\mathbf{x}^{254}$ since $\mathbf{x} \times \mathbf{x}^{254} = \mathbf{x}^{255} = 1$ in $GF(2^8)$ (Fermat's little theorem)
   - $\mathbf{x}^{254}$ could be calculated via square & multiply in $GF(2^8)$...

Algebraic Description of SUB BYTES (2)

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

2. Calculate multiplicative inverse of $\mathbf{x}$ in GF(2^8)
   - $\mathbf{x}^{-1}$ is calculated, e.g., via $\mathbf{x}^{254}$ since $\mathbf{x} \times \mathbf{x}^{254} = \mathbf{x}^{255} = 1$ in GF(2^8) (Fermat's little theorem)
   - $\mathbf{x}^{254}$ could be calculated via square & multiply in GF(2^8)...
   - Alternative more efficient methods were extensively studied...

3. Transform the inverse using an affine transformation

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

1. Interpret input
2. Calculate inverse
3. Transform inverse
4. Interpret result

4. Interpret resulting polynomial as 8-bit output
   - $\mathbf{x} = x^5 + x^3$
   - Output (hex): 0x28
   - Output (bin): 0b00101000

Boyar-Peralta Sbox

- Primarily used for software implementations (and in Task 3)

- Primarily used for software implementations (and in Task 3)
- Bitwise description

Boyar-Peralta Sbox

- Primarily used for software implementations (and in Task 3)
- Bitwise description
- Consists of three layers:
    - Top Linear Layer
    - Middle Non-Linear Layer
    - Bottom Linear Layer

Boyar-Peralta Sbox - Top Linear Layer

```
u0=input[0]
u1=input[1]
u2=input[2]
u3=input[3]
u4=input[4]
u5=input[5]
u6=input[6]
u7=input[7]
```

```
t1=u0⊕u3
t2=u0⊕u5
t3=u0⊕u6
t4=u3⊕u5
t5=u4⊕u6
t6=t1⊕t5
t7=u1⊕u2
t8=u7⊕t6
t9=u7⊕t7
```

```
t10=t6⊕t7
t11=u1⊕u5
t12=u2⊕u5
t13=t3⊕t4
t14=t6⊕t11
t15=t5⊕t11
t16=t5⊕t12
t17=t9⊕t16
t18=u3⊕u7
```

```
t19=t7⊕t18
t20=t1⊕t19
t21=u6⊕u7
t22=t7⊕t21
t23=t2⊕t22
t24=t2⊕t10
t25=t20⊕t17
t26=t3⊕t16
t27=t1⊕t12
```

Boyar-Peralta Sbox - Middle Non-Linear Layer

$m1=t13\times t6$

$m2=t23\times t8$

$m3=t14\oplus m1$

$m4=t19\times u7$

$m5=m4\oplus m1$

$m6=t3\times t16$

$m7=t22\times t9$

$m8=t26\oplus m6$

$m9=t20\times t17$

$m10=m9\oplus m6$

$m11=t1\times t15$

$m12=t4\times t27$

$m13=m12\oplus m11$

$m14=t2\times t10$

$m15=m14\oplus m11$

$m16=m3\oplus m2$

$m17=m5\oplus t24$

$m18=m8\oplus m7m$

$19=m10\oplus m15$

$m20=m16\oplus m13$

$m21=m17\oplus m15$

$m22=m18\oplus m13$

$m23=m19\oplus t25$

$m24=m22\oplus m23$

$m25=m22\times m20$

$m26=m21\oplus m25$

$m27=m20\oplus m21$

$m28=m23\oplus m25$

$m29=m28\times m27$

$m30=m26\times m24$

$m31=m20\times m23$

$m32=m27\times m31$

$m33=m27\oplus m25$

$m34=m21\times m22$

$m35=m24\times m34$

$m36=m24\oplus m25$

$m37=m21\oplus m29$

$m38=m32\oplus m33$

$m39=m23\oplus m30$

$m40=m35\oplus m36$

$m41=m38\oplus m40$

$m42=m37\oplus m39$

$m43=m37\oplus m38$

$m44=m39\oplus m40$

$m45=m42\oplus m41$

$m46=m44\times t6$

$m47=m40\times t8$

$m48=m39\times u7$

$m49=m43\times t16$

$m50=m38\times t9$

$m51=m37\times t17$

$m52=m42\times t15$

$m53=m45\times t27$

$m54=m41\times t10$

$m55=m44\times t13$

$m56=m40\times t23$

$m57=m39\times t19$

$m58=m43\times t3$

$m59=m38\times t22$

$m60=m37\times t20$

$m61=m42\times t1$

$m62=m45\times t4$

$m63=m41\times t2$

Boyar-Peralta Sbox - Bottom Linear Layer


```
l0=m61⊕m62
l1=m50⊕m56
l2=m46⊕m48
l3=m47⊕m55
l4=m54⊕m58
l5=m49⊕m61
l6=m62⊕l5
l7=m46⊕l3
l8=m51⊕m59
l9=m52⊕m53
l10=m53⊕l4
l11=m60⊕l2
l12=m48⊕m51

l13=m50⊕l0
l14=m52⊕m61
l15=m55⊕l1
l16=m56⊕l0
l17=m57⊕l1
l18=m58⊕l8
l19=m63⊕l4
l20=l0⊕l1
l21=l1⊕l7
l22=l3⊕l12
l23=l18⊕l2
l24=l15⊕l9
l25=l6⊕l10

l26=l7⊕l9
l27=l8⊕l10
l28=l11⊕l14
l29=l11⊕l17

output[0]=l6⊕l24
output[1]=¬l16⊕l26
output[2]=¬l19⊕l28
output[3]=l6⊕l21
output[4]=l20⊕l22
output[5]=l25⊕l29
output[6]=¬l13⊕l27
output[7]=¬l6⊕l23
```

Boyar-Peralta Sbox

- In total 129 instructions

Boyar-Peralta Sbox

- In total 129 instructions
- $\approx 129 \times$ slower than one table lookup

Boyar-Peralta Sbox

- In total 129 instructions
- $\approx 129 \times$ slower than one table lookup
- Performance can be improved via bitslicing (Task 3)
    - Convert 4×4 byte state into 8×16-bit state
    - First 16-bit reg holds the LSBs of all 16 bytes, etc...
    - Calculate Sbox bitwise but with 16-bit registers

- In total 129 instructions
- $\approx$ 129 $\times$ slower than one table lookup
- Performance can be improved via bitslicing (Task 3)
  - Convert 4$\times$4 byte state into 8$\times$16-bit state
  - First 16-bit reg holds the LSBs of all 16 bytes, etc...
  - Calculate Sbox bitwise but with 16-bit registers
- Still $\approx$ 8 $\times$ slower than lookup tables

Boyar-Peralta Sbox

- In total 129 instructions
- $\approx$ 129 $\times$ slower than one table lookup
- Performance can be improved via bitslicing (Task 3)
  - Convert 4$\times$4 byte state into 8$\times$16-bit state
  - First 16-bit reg holds the LSBs of all 16 bytes, etc...
  - Calculate Sbox bitwise but with 16-bit registers
- Still $\approx$ 8 $\times$ slower than lookup tables
- AES was never meant to be used that way, but we have to...

Masking Boyar-Peralta Sbox

- Sbox only consists of: $\oplus$, $\times$, $\neg$ in GF(2), hence XOR, AND, NOT

Masking Boyar-Peralta Sbox

- Sbox only consists of: $\oplus$, $\times$, $\neg$ in GF(2), hence XOR, AND, NOT
- Masking $\oplus$ and $\neg$ is easy:
  - Duplicate $\oplus$ and perform them on both shares
  - $\neg$ is equal to $\oplus$ 1 thus only performed on one share

Masking Boyar-Peralta Sbox

- Sbox only consists of: $\oplus$, $\times$, $\neg$ in GF(2), hence XOR, AND, NOT
- Masking $\oplus$ and $\neg$ is easy:
  - Duplicate $\oplus$ and perform them on both shares
  - $\neg$ is equal to $\oplus$ 1 thus only performed on one share
- Remaining problem: Masking $\times$ (AND-gate)

Notes regarding task 3

What to do?

- Your task: implement a masked AES.
  - Send key shares (always fresh).
  - Do the masked computation (ISW).
- Keep it simple, no premature optimization!
- Ensure that you get the correct values at the end.

A few tricks:

- Masking PRNG
  - No need for cryptographically-secure PRNG.
  - Can be fairly simple, e.g. Linear congruential generator (LCG). *Use fresh seeds.*
- `__attribute__ ((noinline))` "hides" the content of a function to the optimizer (write bitwise AND, XOR, NOT functions. . .).
- If you want to avoid transitions (optional):
  - Gadgets as functions with an inline assembly blocks.
  - Gadget functions takes pointers shares arrays.
  - C code calls gadgets, does not touch the shares
- To disable masking: set input sharings as $(x, 0, \dots, 0)$ and set PRNG output to 0.

Thank you!

Questions:

rishub.nagpal@iaik.tugraz.at

Masking - Defeating Power Analysis Attacks

Side-Channel Security

Rishub Nagpal

June 6, 2023

IAIK – Graz University of Technology