Passive Side-Channel Analysis

Counter-measures and Advanced Attacks

Lorenzo CASALINO

CentraleSupélec, Inria, IRISA

04 Feb. 2025

Agenda

1. On the Cost of Side-Channel Analysis
2. Counter-measures
   - Masking
   - Hiding
3. Break the Masking Counter-measure
   - Second-order Analysis
   - Bivariate Analysis
   - Conclusion

On the Cost of Side-Channel Analysis

Side-Channel Analysis Efficiency

Goals of a Side-Channel Attacker

An attacker wants to mount an attack that:

- recover with high probability the sensitive value
- is *efficient* (i.e., takes reasonable time)

The *number of traces* for a successful attack impact on the efficiency of the attack itself:

- Acquisition;
- Storing;
- Preprocessing;
- Analysis.

Thus, we can measure the efficiency (dually, the *complexity*) of a side-channel attack by the number of traces required.

Counter-measures

Side-Channel Counter-measure

Any (hardware or software) approach to increase the complexity of a side-channel attack.

In the scientific literature, we categorise the counter-measures in:

Masking Counter-measure

Randomly encode data **to remove** the (statistical) influence of data processing on the side-channel signal.

Hiding Counter-measure

Inject noise in the side-channel signal **to hide** (statistical) influence of the data processing on the side-channel signal.

Masking

A First Example: a Side-Channel Secure Bitwise AND

We want to compute, in a side-channel secure manner, the bitwise XOR between a:

- **public** variable **b**
- **private** (or *sensitive*) variable **a**.

Table: AND's Truth Table

a	**b**	**c**
0	0	0
0	1	0
1	0	0
1	1	1

An attacker observing **c** gets information on **a**:

- $\mathbf{c} = 1$ then $\mathbf{a} = 1$ for sure,
- $\mathbf{c} = 0$ then $\mathbf{a} = 0$ with high probability.

The AND algorithm **is not** side-channel secure.

A First Example: a Side-Channel Secure Bitwise AND

Now, let us consider $\mathbf{c'} = \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.

For $\mathbf{r} = 0$:

- $\mathbf{c'} = 1 = \mathbf{c} \oplus 0$ then $\mathbf{a} = 1$ for sure
- $\mathbf{c'} = 0 = \mathbf{c} \oplus 0$ then $\mathbf{a} = 0$ with high probability.

For $\mathbf{r} = 1$:

- $\mathbf{c'} = 0 = \mathbf{c} \oplus 1$ then $\mathbf{a} = 1$ for sure
- $\mathbf{c'} = 1 = \mathbf{c} \oplus 1$ then $\mathbf{a} = 0$ with high probability.

Table: AND's Truth Table

a	**b**	**c**	$\mathbf{c'}_{r=0}$	$\mathbf{c'}_{r=1}$
0	0	0	0	1
0	1	0	0	1
1	0	0	0	1
1	1	1	1	0

If the attacker knows that **r** is fixed, they get knowledge on **a**.

A First Example: a Side-Channel Secure Bitwise AND

Let us consider $\mathbf{c'} = \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}$$

Without knowing (or *observing*) **r**, can you guess the **a**'s value?

A First Example: a Side-Channel Secure Bitwise AND

Let us consider $\mathbf{c'} = \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}$$

Without knowing (or *observing*) **r**, can you guess the **a**'s value?

A First Example: a Side-Channel Secure Bitwise AND

Let us consider **c'** $= \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}$$

Without knowing (or *observing*) **r**, can you guess the **a**'s value?

A First Example: a Side-Channel Secure Bitwise AND

Let us consider **c'** $= \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}$$

Without knowing (or *observing*) **r**, can you guess the **a**'s value?

A First Example: a Side-Channel Secure Bitwise AND

Let us consider **c'** $= \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}$$

Without knowing (or *observing*) **r**, can you guess the **a**'s value?

A First Example: a Side-Channel Secure Bitwise AND

Let us consider **c'** $= \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{1}_{\mathbf{a}=1}, \underbrace{0}_{\mathbf{a}=0}, \underbrace{0}_{\mathbf{a}=0}$$

Without knowing (or *observing*) **r**, can you guess **a**'s value?

The Risk of Using Fixed Variables

A fixed **r** does not protect against statistical analysis.

A First Example: a Side-Channel Secure Bitwise AND

Let us consider $\mathbf{c'} = \mathbf{c} \oplus \mathbf{r}$, where $\mathbf{r} \in \{0, 1\}$ and **fixed**.
Let us consider the following sequence for $\mathbf{c}' = \mathbf{c} \oplus \mathbf{r}$:

$$\underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{1}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}, \underbrace{0}_{\mathbf{a}=?}$$

If you can get the value of **a** (and you give a mathematical proof), you win 1 billion!

Without observing **r**, you **can not** derive any information on **a**.

This is the principle behind *masking*.

Masking

Goal

Randomly encode data **to remove** the (statistical) influence of processing data and on the side-channel signal.

Principle

Replace all the sensitive variables with their *randomised encoding*.

Randomised Encoding

Given a variable $\mathbf{v}$ and a random variable $\mathbf{r}$, we call *randomised encoding of* $\mathbf{v}$ the pair:

$$\overline{\mathbf{v}} = < \underbrace{\mathbf{v} \oplus \mathbf{r}}_{\overline{\mathbf{v}}_0}, \underbrace{\mathbf{r}}_{\overline{\mathbf{v}}_1} > .$$

We call $\overline{\mathbf{v}}_0$ and $\overline{\mathbf{v}}_1$ *shares*.

Masking: Fundamental Properties

Correctness

Given an encoding $\overline{\mathbf{v}}$ of a variable $\mathbf{v}$, we say that:

$$\overline{\mathbf{v}} \text{ is } \textit{correct} \iff \mathbf{v} = \overline{\mathbf{v}}_0 \oplus \overline{\mathbf{v}}_1.$$

First-order (1^{st}-order) Security

Given an encoding $\overline{\mathbf{v}}$ of a variable $\mathbf{v}$, we say that:

$$\overline{\mathbf{v}} \text{ is } 1^{\text{st}}\text{-order secure} \Leftrightarrow 1^{\text{st}}\text{-order analysis does not get } \mathbf{v} \text{ from } \overline{\mathbf{v}}.$$

The DPA and CPA you studied **are** first-order analysis.
We will see the concept of first-order and higher-order analysis later.

Masking an Algorithm

Masking sensitive data $\Rightarrow$ adapt the algorithm to use the encodings

Example: bitwise XOR

a and **b** are sensitive variables in input to the following function:

```
def bitwiseXOR(a: bit,b: bit) -> c: bit:
  return xor(a, b)
```

How do we adapt (i.e., mask) `bitwiseXOR` to process $\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$?

Example: Secure XOR

```
def secXOR(???) -> ??? :
  ???
  ???
  ???
```

Masking an Algorithm

Masking sensitive data $\Rightarrow$ adapt the algorithm to use the encodings

Example: bitwise XOR

a and **b** are sensitive variables in input to the following function:

```
def bitwiseXOR(a: bit,b: bit) -> c: bit:
  return xor(a, b)
```

How do we adapt (i.e., mask) `bitwiseXOR` to process $\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$?

Example: Secure XOR

```
def secXOR(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  ???
  ???
  ???
```

Masking an Algorithm

Masking sensitive data $\Rightarrow$ adapt the algorithm to use the encodings

Example: bitwise XOR

a and **b** are sensitive variables in input to the following function:

```
def bitwiseXOR(a: bit,b: bit) -> c: bit:
  return xor(a, b)
```

How do we adapt (i.e., mask) `bitwiseXOR` to process $\overline{\mathbf{a}}$ and $\overline{\mathbf{b}}$?

Example: Secure XOR

```
def secXOR(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = xor(a0, b0)
  c1 = xor(a1, b1)
  return c0, c1
```

Masking an Algorithm

We modified `bitwiseXOR` to work on encodings. Now, we have to:

- verify *correctness*
- verify *first-order security* (we won't do that).

Example: secure XOR

```
def secXOR(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = xor(a0, b0)
  c1 = xor(a1, b1)
  return c0, c1
```

Example: secXOR's Correctness

We demonstrate just the *if* proposition:

$$\begin{aligned}\overline{\mathbf{c}}_0 \oplus \overline{\mathbf{c}}_1 &= (\overline{\mathbf{a}}_0 \oplus \overline{\mathbf{b}}_0) \oplus (\overline{\mathbf{a}}_1 \oplus \overline{\mathbf{b}}_1)\\ &(\overline{\mathbf{a}}_0 \oplus \overline{\mathbf{a}}_1) \oplus (\overline{\mathbf{b}}_0 \oplus \overline{\mathbf{b}}_1)\\ &\mathbf{a} \oplus \mathbf{b} = \mathbf{c}\ \blacksquare.\end{aligned}$$

Masking another Algorithm

Example: bitwise AND

```
def bitwiseAND(a: bit,b: bit) -> c: bit:
  return and(a, b)
```

Example: Secure AND

```
def secAND(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = and(a0, b0)
  c1 = and(a1, b1)
  return c0, c1
```

Example: secAND's Correctness

We demonstrate just the *only if* proposition:
By assumption, $\forall\ \overline{\mathbf{a}}, \overline{\mathbf{b}}$: secAND is *correct*.

$$\begin{aligned}\mathbf{c} = \mathbf{a} \cdot \mathbf{b} =& (\overline{\mathbf{a0}} \oplus \overline{\mathbf{a1}}) \cdot (\overline{\mathbf{b0}} \oplus \overline{\mathbf{b1}}) \\ & (\overline{\mathbf{a0}} \cdot \overline{\mathbf{b0}}) \oplus (\overline{\mathbf{a0}} \cdot \overline{\mathbf{b1}}) \oplus (\overline{\mathbf{a1}} \cdot \overline{\mathbf{b0}}) \oplus (\overline{\mathbf{a1}} \cdot \overline{\mathbf{b1}}) \\ & \neq (\overline{\mathbf{a0}} \cdot \overline{\mathbf{b0}}) \oplus (\overline{\mathbf{a1}} \cdot \overline{\mathbf{b1}}) = \mathbf{c}.\end{aligned}$$

Masking another Algorithm

Example: secAND's Correctness

`secAND` is not correct! It misses the *cross-products*:

$$(\bar{\mathbf{a}}_0 \cdot \bar{\mathbf{b}}_1) \text{ and } (\bar{\mathbf{a}}_1 \cdot \bar{\mathbf{b}}_0) :$$

```
def secAND(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = and(a0, b0)
  c1 = and(a1, b1)

  c2 = and(a0, b1) # cross-product
  c3 = and(a1, b0) # cross-product

  z = genRandomVar() # generate random variable
  tmp = xor(xor(c2, z), c3) # z avoids information leakage
  c0 = xor(c0, z)
  c1 = xor(c1, tmp)

  return c0, c1
```

`bitwiseAND` is an example of *non-linear* algorithm.

Masking: a Recap of Its Application

We distinguish between:

Linear Computations : for instance, the `bitwiseXOR`

Masking (Unary) Linear Computations

Given a linear algorithm $\mathbf{c} = \mathtt{A}(\mathbf{a})$:

```
def secA(a0: bit, a1: bit) -> c0: bit, c1: bit:
  c0 = A(a0)
  c1 = A(a1)
  return c0, c1
```

Masking (Binary) Linear Computations

Given a linear algorithm $\mathbf{c} = \mathtt{A}(\mathbf{a}, \mathbf{b})$:

```
def secA(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = A(a0, b0)
  c1 = A(a1, b1)
  return c0, c1
```

Masking: a Recap of Its Application

We distinguish between:

Linear Computations : for instance, the `bitwiseXOR`

Non-Linear Computations : for instance, the `bitwiseAND`

Generic Approach to Mask Non-Linear Computations

1. Decompose the algorithm in a set of basic linear and non-linear operations (e.g., `NOT` and `AND`)
2. Mask the basic operations
3. Compose back the masked basic operations.

Nota Bene: scientific literature contains the masked version of certain basic non-linear operations (e.g., `bitwiseAND`)

On the Cost of Masked Linear Computations

Masking is not free, neither a cheap countermeasure.

```
def bitwiseXOR(a: bit,b: bit) -> c: bit:
  return xor(a, b)
```

```
def secXOR(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = xor(a0, b0)
  c1 = xor(a1, b1)
  return c0, c1
```

- Number of computations: 2 `xors`, 0 random variables
- The unprotected `bitwiseXOR` requires only 1 `xor`

On the Cost of Masked Linear Computations

Generally, masked linear computations require $\times 2$ more operations.

On the Cost of Masked Non-Linear Computations

Masking is not free, neither a cheap countermeasure.

```
def secAND(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  c0 = and(a0, b0)
  c1 = and(a1, b1)

  c2 = and(a0, b1) # cross-product
  c3 = and(a1, b0) # cross-product

  z = genRandomVar() # generate random variable
  tmp = xor(xor(c2, z), c3) # z avoids information leakage
  c0 = xor(c0, z)
  c1 = xor(c1, tmp)

  return c0, c1
```

- Number of computations: 4 `and`s, 4 `xor`s, 1 random variable
- The unprotected `bitwiseAND` requires only 1 `and`

On the Cost of Masked Non-Linear Computations

Generally, masked non-linear computations require $\times 4$ computations.

Masking More Complex Algorithms

Until now, we have:

- investigated the application of masking to two simple (i.e., one basic computation) algorithms,
- seen how to apply masking to linear and non-linear computations.

In the lab activities, we will go through the masking of a more complex algorithm: the `AES-128` cryptosystem.

Example: AES-128's Round Function

```
def round(state: 4x4 Matrix, roundKey: 4x4 Matrix) -> state: 4x4 Matrix:
  state = shiftRows(state)
  state = subByte(state)
  state = mixColumns(state)
  state = addRoundKey(state, RoundKey)

  return state
```

Hiding

Hiding

Goal

Inject noise in the side-channel signal **to hide** (statistical) influence of the data processing on the side-channel signal.

Side-channel Signal L

Composition of an *informative signal* I and a *non-informative* signal N (also called *noise*):

$$L = I + N$$

Hiding's Principle

Increase N, such that $L \approx N$. The attacker requires a huge amount of traces to detect I.

Overview of Hiding techniques

Random Delays Injection

Randomly insert delays in the execution of one or more computations.

Shuffling

Randomly change the order of executed computations.

Code Polymorphism

Randomly replace computations with semantically equivalent computations.

Overview of Hiding techniques

Random Delays Injection

Randomly insert delays in the execution of the program.

```
def secXOR(a0: bit, a1: bit, b0: bit, b1: bit) -> c0: bit, c1: bit:
  z = genRandomVar() # Generate a random variable
  c0 = xor(a0, b0)
  for i from 0 to r: # Insert a random delay
    doNothing()
  c1 = xor(a1, b1)
  return c0, c1
```

Overview of Hiding techniques

Shuffling

Randomly change the order of executed computations.

We apply shuffling at the level of instructions, loops, functions, ...

Instruction Shuffling

Randomly change the order of executed instructions.

```
def secAND(...):
  c0 = and(a0, b0)
  c1 = and(a1, b1)

  c2 = and(a0, b1)
  c3 = and(a1, b0)

  ...

  return c0, c1
```

```
def secAND(...):
  c2 = and(a0, b1)
  c3 = and(a1, b0)

  c0 = and(a0, b0)
  c1 = and(a1, b1)

  ...

  return c0, c1
```

```
def secAND(...):
  c0 = and(a0, b0)
  c2 = and(a0, b1)

  c1 = and(a1, b1)
  c3 = and(a1, b0)

  ...

  return c0, c1
```

Overview of Hiding techniques

Code Polymorphism

Randomly replace computations with semantically equivalent computations.

We apply code polymorphism at the level of instructions, loops, functions, ...

```
def secXOR(...):
  c0 = xor(a0, b0)
  c1 = xor(a1, b1)
  return c0, c1
```

```
def secXOR(...):
  tmp00 = or(a0, b0)
  tmp01 = not(tmp00)
  c0 = not(or(tmp00, tmp01))

  tmp10 = or(a1, b1)
  tmp11 = not(tmp10)
  c1 = not(or(tmp10, tmp11))
  return c0, c1
```

Break the Masking Counter-measure

Second-order Analysis

First-order Attacks and Security

We have seen that a correct application of masking is first-order secure. That is, DPA and CPA, in the forms we have seen them, cannot recover any the sensitive information.
But what does mean *first-order secure*? And is it possible to be *second-order secure*?

Distinguisher

A statistical tool that, given a set a leakage hypothesis and a set of traces, provides a score quantifying the likelihood ...

First-order Side-Channel Analysis

Any side-channel analysis where the distinguisher relies on the first-order statistical moment (i.e., the mean) of the encoding's distribution.

DPA based on the *Difference of Means* and CPA **are** first-order side-channel analysis.

First-order Attacks and Security

Let us imagine that:

- we mask a 2-bit variable **v**,
- we can model the leakage with the Hamming weight function

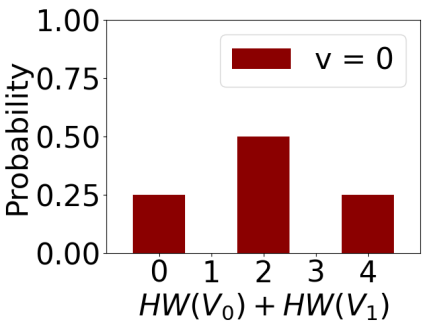

Figure: mean = 2

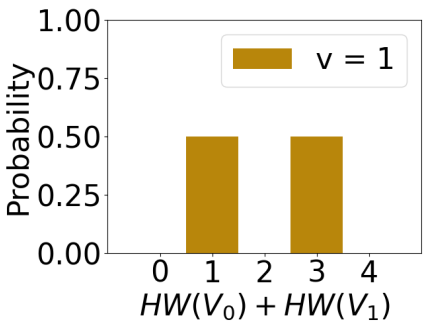

Figure: mean = 2

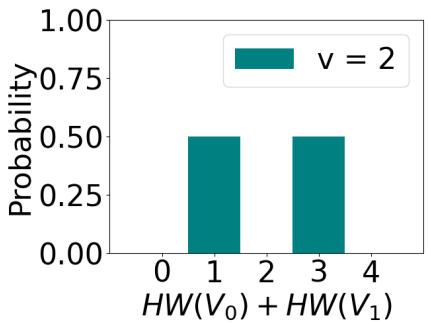

Figure: mean = 2

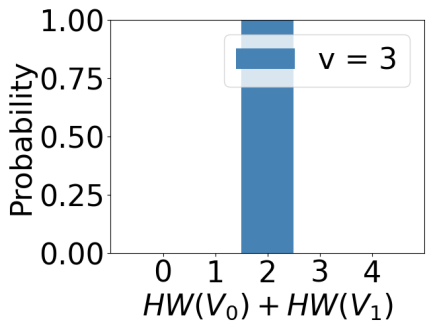

Figure: mean = 2

Whatever the value of **v**, the mean is always the same!.

A first-order analysis (i.e., exploiting the leakage distribution's mean) can not break (first-order) masking.

But wait, yet the 4 distributions are different!

Second-Order Side-Channel Analysis

Second-Order Side-Channel Analysis

Perform the attack exploiting the *second-order* statistical moment (i.e., the variance) of the leakage distribution.

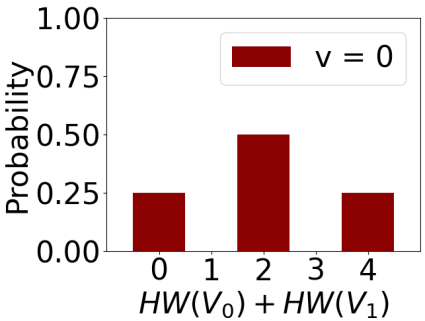

Figure: Variance = 2

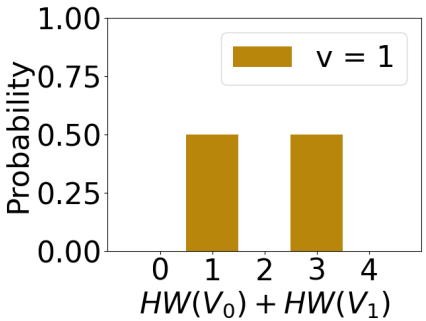

Figure: Variance = 1

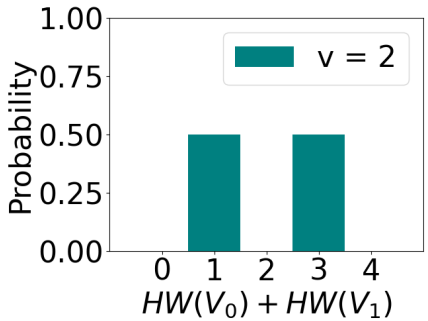

Figure: Variance = 1

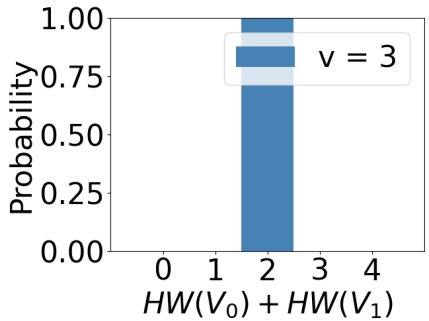

Figure: Variance = 0

The variance of the leakage distributions differ!

A second-order analysis (i.e., exploiting the leakage distribution's variance) *can* break (first-order) masking.

During the lab activities, we will implement a second-order CPA.

Bivariate Analysis

Bivariate Side-Channel Analysis

The two shares $\bar{\mathbf{v}}_0, \bar{\mathbf{v}}_1$ of an encoding $\bar{\mathbf{v}}$ may be employed in two different time instants $t = t_0$ and $t = t_1$.
Even a second-order analysis on $t = t_0$ or on $t = t_1$ will never recover any sensitive information. Why?
Because we would *miss* information.

Bivariate Analysis

Side-channel analysis considering *two* samples at the same time.

In a bivariate analysis, we *combine* the two samples via the *Centred-Product* function.

Centred Product Recombination

Let T be a set of n side-channel traces, each made of m samples, and T_0, T_1 the samples where the share $\bar{\mathbf{v}}_0$ and $\bar{\mathbf{v}}_1$ are used.
We have:

$$C(T_i, T_j) = (T_i - \mu_i) \cdot (T_j - \mu_j)$$

the centred product combination of the two samples.

Conclusion

Recap

- We overviewed the masking counter-measure
- We overviewed its principles (correctness and first-order security)
- We overviewed how to apply it to simple and complex algorithms through a compositional approach
- We overviewed higher-order and multivariated side-channel analysis and how we can use them to break first-order implementations.

In the next lab activities, we will:

- go through the masking of simple operations
- go throught the first-order masking of the AES-128 cryptosystem
- attack your first-order masked AES-128 implementation by means of second-order and (maybe) bivariated analysis.