Sécurité des Systèmes Embarqués
TP

Lorenzo Casalino
lorenzo.casalino@centralesupelec.fr
lorenzo.casalino@inria.fr

04 Feb. 2025

1 The Masking Countermeasure

In this laboratory activity we will get acquainted with the masking countermeasure.

1.1 Power Consumption, Statistical Dependency and Secret Value Recovery

The principle underlying side-channel analysis is the exploitation of the *statistical dependency* between the measured side channel (e.g., the power consumption) and the processed data.

Let us have a visual representation of such dependency: let us consider a generic binary random variable V (i.e., it can take values 0 or 1) playing the role of a secret value (e.g., a cryptographic key bit) and plot the probability distribution of a variable L representing the measured side channel.

We mathematically model L as:

$$L = f(V, N)$$

where f describes the influence of V and N — the *noise* affecting the side-channel signal — on the measured side channel. In particular instance, we consider the following:

$$f(V, N) = V + \mathcal{N}(\mu = 0, \sigma = 1)$$

[ ]:
```python
import matplotlib.pyplot as plt
import numpy as np

# Some parameters

## The number of masks to use

n_masks = 1000

# The Hamming weight function
def hw(x):
    return bin(x).count('1')
```

```
def genData(v):
    return np.random.normal(v, 1, 50)
```

[ ]:
```
# We work on a binary finite field
field_size = 2

fig, axs = plt.subplots(1, field_size, sharex = False, sharey = True)
for i in range(0, field_size):
    axs[i].hist(genData(i), density = True, bins = 'auto', label = f"v = {i}")
    axs[i].set_xlim(-fieldSize - 4 + 1, fieldSize + 4 - 1)
    axs[i].set_xlabel("Value")
    axs[i].set_ylabel("Probability")
    axs[i].legend()
    axs[i].axvline(x = i, color = 'red')
    axs[i].set_xticks([0, i, fieldSize])

plt.tight_layout()
plt.show()
```

Although the distributions have a similar shape, they differ for their *mean value* (in red). Thus, by exploiting any statistical tool relying on the mean value (for instance, the pearson's correlation coefficient), we can distinguish, from the measured side channel, what value a program processes.

Yet, we use the actual variable V to model the side channel L. In many real-world situations, the influence of V is much more complicated, although we can abstract from several details.

A classical model is the following:

$$L = f(V) = HW(V) + \mathcal{N}(\mu = 0, \sigma = 1)$$

where HW is the Hamming weight function.

Still, by considering V a 1-bit variable (i.e., taking values 0 and 1), we would notice no difference as $V = HW(V)$. For this reason, we exceptionally consider for this example V as a 3-bit variable (i.e., taking values from 0 to 7).

Let us visualise the statistical dependency between L and V under this new mathematical description of L:

[ ]:
```
field_size = 8

fig, axs = plt.subplots(2, 4, sharex = False, sharey = True)

for i in range(0, 2):
    for j in range(0, 4):
        axs[i, j].hist(genData(hw(idx + j)), density = True, bins = 'auto',␣
 ↪label = f"v = {i * 4 + j}")
        axs[i, j].set_xlim(-5, 5)
        if (i == 1):
            axs[i, j].set_xlabel("Value")
        if (j == 0):
            axs[i, j].set_ylabel("Probability")
        axs[i, j].legend()
        axs[i, j].axvline(x = hw(i * 4 + j), color = 'red')
        axs[i, j].set_xticks([0, hw(i * 4 + j), 3])

plt.tight_layout()
plt.show()
```

We observe that many values of V share the same mean value; for instance, $V = 3, 5$ and 6. At the same time, it is still possible to analyse L's mean to distinguish, with a certain probability, the processed value; for instance, when the mean value of L's distribution is 3, we are sure that $V = 7$.

Thus, although the added uncertainty due to the HW function, a side-channel attacker can statistically analyse L's probability distribution (the side channel's probability distribution) to recover a secret datum.

How can we remove such statistical dependency?

1.2 Removing the Statistical Dependency

Let us assume that, instead of processing the random variable V, we process the following:

$$V_0 = V \oplus V_1$$

where V_1 is a uniform random variable.

We call the variable $\overline{V} = \langle V_0, V_1 \rangle$ the *encoding* of V, and V_0, V_1 *shares*.

Before continuing, let us implement the function `genShare`, which generates the share V_0 from a variable V and a random variable V_1:

```
[ ]: def genShare(v, v1):
         assert False, "Missing implementation!"
```

Let us test your implementation:

```
[ ]: testValues = np.random.randint(0, field_size, 10, dtype = np.uint8)
     testMasks = np.random.randint(0, field_size, 10, dtype = np.uint8)

     for i in range(0, 10):
         assert genShare(testValues[i], testMasks[i]) ^ testMasks[i] == testValues[i]
```

Now, we can depict the statistical relationship between V_0 and L:

```
[ ]: field_size = 2

     fig, axs = plt.subplots(1, field_size, sharex = False, sharey = True)
```

```
masks = np.random.randint(0, field_size, nmasks)

for i in range(0, field_size):
    v_zeros = [ genShare(i, r) for r in masks ]
    mean = np.mean([hw(v) for v in v_zeros])
    for v in v_zeros:
        axs[i].hist(genData(hw(v)), density = True, bins = 'auto', label = f"v =␣
 ↪{i}", alpha = 0.2)
    axs[i].set_xlim(-field_size - 4 + 1, field_size + 4 - 1)
    axs[i].set_xlabel("Value")
    axs[i].set_ylabel("Probability")
    axs[i].axvline(x = mean, color = 'red')
    axs[i].set_xticks([0, mean, field_size - 1])
    axs[i].set_xticklabels([0, mean, field_size - 1], rotation = 90)

plt.tight_layout()
plt.show()
```

What we observe is that the mean value of the two distribution is *almost* the same. The reason for the discrepancy is only due to statistical reasons, and we may actually consider them to be the same. As such, we can not learn which V one is processed by looking at the mean (and, actually, at any other statistical moment of L's distribution).

By applying the random variable V_1 on V, we have removed, or *masked*, the statistical relationship between V and L.

Yet, it is really important for V_1 to follow a **random uniform** distribution, otherwise we might still detect statistical dependence between V and L.

Indeed, if for instance we fix $V_1 = 1$ for every value V may take, we have that:

$$V_0 = V \oplus V_1 = V \oplus 1 = -V$$

that is, we observe the same relationship we had before!

Ok, and if V_1 is not fixed but is not a *uniform* random variable? Let us see what happens:

```
[ ]: fig, axs = plt.subplots(1, field_size, sharex = False, sharey = True)

# The share V_{1} can take value 0 with probability 0.7, and value 1 with␣
 ↪probability 0.3
masks = np.random.binomial(1, 0.7, nmasks).astype(int)

for i in range(0, field_size):
    v_zeros = [ genShare(i, r) for r in masks ]
    mean = np.mean([hw(v) for v in v_zeros])
    for v in v_zeros:
        axs[i].hist(genData(hw(v)), density = True, bins = 'auto', label = f"v =␣
↪{i}", alpha = 0.2)
    axs[i].set_xlim(-field_size - 4 + 1, field_size + 4 - 1)
    axs[i].set_xlabel("Value")
    axs[i].set_ylabel("Probability")
    axs[i].axvline(x = mean, color = 'red')
    axs[i].set_xticks([mean])

plt.tight_layout()
plt.show()
```

The mean value of the two distributions is clearly different!

These were simple, operative examples on why V_1 should follow a uniform random distribution. In the next sections, we will see how to adapt computations to process the encodings of its inputs.

1.3 Masking Linear Computations

We say that a computation f is *linear* with respect a certain operation $\oplus$ when it respects the following property:

$$f(x \oplus y) = f(x) \odot f(y)$$

In the context of masking, we can reformulate the property in terms of the encoding $\overline{V} = \langle V_0, V_1 \rangle$ of V:

$$f(V) = f(V_0 \oplus V_1) = f(V_0) \oplus f(V_1)$$

That is, if $\overline{V}$ is a correct encoding for V, we can apply f separately to each share of $\overline{V}$ and still get a correct encoding.

Let us see an example.

1.3.1 Masking the bitwise `Shift` function

Let us consider the bitwise `shift` function.

This function simply shifts a supplied integer `value` on the `left` or on the `right` by a certain `amount` of bits.

[ ]:
```
valid_directions = ['left', 'right']

def shift(value, amount, direction):
    assert direction in valid_directions, f"Invalid direction {direction}."

    if direction == 'left':
        return value << amount
    return value >> amount
```

This function is linear with respect to the $\oplus$ operation we use for masking. As such, we can apply it separately on both shares to get a masked version of `shift`.

Can you implement this `secShift` function?

[ ]:
```
def secShift(v0, v1, amount, direction):
    assert False, "Missing implementation"
```

Now, let us run some tests to check your `secShift` correctly works (i.e., it respect the *correctness* property).

[ ]:
```
testValues = np.random.randint(0, 256, 1000, dtype = np.uint8)
testMasks = np.random.randint(0, 256, 1000, dtype = np.uint8)
testAmounts = np.random.randint(0, 9, 1000, dtype = np.uint8)

for i in testValues:
    v = testValues[i]
    v1 = testMasks[i]
    v0 = genShare(v, v1)
    assert secShift(v0, v1, testAmounts[i], 'left') == shift(v, testAmounts[i],␣
 ↪'left'), f"{i}: test failed."
    assert secShift(v0, v1, testAmounts[i], 'right') == shift(v, testAmounts[i],␣
 ↪'right'), f"{i}: test failed."

print("Test passed!")
```

If the above cell outputs `Test passed!`, congrats: you developed your first masked algorithm! Let us see another example of linear function.

1.3.2 Masking the `XOR` function

The `XOR` function (or operator) forms the basis of several algorithms (e.g., `AES-128`).

[ ]:
```
def bitwise_xor(a, b):
    return a ^ b
```

Can you implement its masked version `secXOR`? This time, it is up to you to specify also the arguments to the masked function.

[ ]:
```
def secXOR():
    assert False, "Missing implementation"
```

Let us test your masked implementation (i.e., it respect the *correctness* property):

[ ]:
```
testValues = np.random.randint(0, 256, 1000, dtype = np.uint8)
testMasks = np.random.randint(0, 256, 1000, dtype = np.uint8)
testAmounts = np.random.randint(0, 9, 1000, dtype = np.uint8)

for i in testValues:
    v = testValues[i]
    v1 = testMasks[i]
    v0 = genShare(v, v1)
    assert secXOR(a0, a1, b0, b1) == xor(a, b), f"{i}: test failed."

print("Test passed!")
```

If you implemented the `secXOR` correctly, you should get `Test passed!`.

Until now, we overviewed the application of masking on linear computation. But what about non-linear ones?

1.4 Masking Non-Linear Computations

A non-linear computations f is, simply, a computation that does not respect the linearity property:

$$f(V) = f(V_0 \oplus V_1) \neq f(V_0) \oplus f(V_1)$$

In other words, we cannot apply the function f on both the shares and pretend to get back a correct encoding. We will see this through a classical example: the bitwise `AND`.

1.4.1 Masking the bitwise`AND` function

Let us try to mask the `AND` as it was a linear computation:

[ ]:
```
def bitwise_and(a, b):
    return a & b

def secAND(a0, a1, b0, b1):
    c0 = bitwise_and(a0, b0)
    c1 = bitwise_and(a1, b1)
    return c0, c1

a, b, a1, b1 = np.random.randint(0, 256, 4)

a0 = genShare(a, a1)
b0 = genShare(b, b1)

c0, c1 = secAND(a0, a1, b0, b1)
c = bitwise_and(a, b)


print(f"c0 ^ c1 = {c0} ^ {c1} = {c0 ^ c1} != {c} = c")
assert (c0 ^ c1) != c
```

As you can see, the recombination of c_0 and c_1 does not equal c. The reason stems from the fact that the current `secAND` does not compute the *non-linear elements* of the `AND` operation.

$$(a_0 \oplus a_1) \cdot (b_0 \oplus b_1) = a_0 b_0 \oplus a_1 b_1 \oplus \underbrace{a_0 b_1 \oplus a_1 b_0}_{\text{non-linear elements}}$$

Clearly, the solution is to add the computation of theses non-linear element to the `secAND` implementation. Yet, we found ourself with four intermediate values, but the `secAND` should return two shares.

The solution is to recombine some of these intermediate values in order to reduce the number of intermediate values to only two.

[ ]:
```
def secAND(a0, a1, b0, b1):
    # Linear elements
    c00 = bitwise_and(a0, b0)
    c11 = bitwise_and(a1, b1)

    # Non-linear elements
    c01 = bitwise_and(a0, b1)
    c10 = bitwise_and(a1, b0)
    tmp = bitwise_xor(c01, c10)

    # Recombine
    c11 = bitwise_xor(c11, tmp)

    return c00, c11
```

[ ]:
```
a, b, a1, b1 = np.random.randint(0, 256, 4)
a0 = genShare(a, a1)
b0 = genShare(b, b1)

c0, c1 = secAND(a0, a1, b0, b1)
c = bitwise_and(a, b)


print(f"c0 ^ c1 = {c0} ^ {c1} = {c0 ^ c1} != {c} = c")
assert (c0 ^ c1) == c

print("Test passed!")
```

Still, we can not blindly recombine the intermediate values; otherwise, we would leak information on the inputs. Indeed, if for instance we look at the intermediate value `tmp` in the `secAND` implementation, we see it can be written as:

$$(a_0 b_1) \oplus (a_1 b_0) = (a_0 b_1) \oplus (a b_0 \oplus a_0 b_0) = a b_0 \oplus a_0 b$$

From this expansion of `tmp`, we see that there is a statistical dependence between `tmp` and `a`, and between `tmp` and `b`.

We can also visualise such statistical dependence:

```
[ ]: # We extrapolate in a dedicated function the computation
     # generating `tmp`.
     def recombNonLinears(nl0, nl1, r):
         return nl0 ^ nl1
```

```
[ ]: fig, axs = plt.subplots(1, field_size, sharex = False, sharey = True)

     for a in range(0, field_size):
         a1s = np.random.randint(0, fieldSize, nmasks)
         a0s = [ genShare(a, a1) for a1 in a1s ]

         b1s = np.random.randint(0, fieldSize, nmasks)
         b0s = [ genShare(0, b1) for b1 in b1s ]

         randoms = np.random.randint(0, fieldSize, nmasks)

         nonLinears = [ (bitwise_and(a0s[i], b1s[i]), bitwise_and(a1s[i], b0s[i]),␣
      ↪randoms[i]) for i in range(0, nmasks) ]

         recombs = [ recombNonLinears(*x) for x in nonLinears ]
         mean = np.mean(recombs)
         for x in recombs:
             axs[a].hist(genData(hw(x)), density = True, bins = 'auto', label = f"v =␣
      ↪{i}", alpha = 0.2)
         axs[a].set_xlim(-field_size - 4 + 1, field_size + 4 - 1)
         axs[a].set_xlabel("Value")
         axs[a].set_ylabel("Probability")
         axs[a].set_xticks([0, mean, field_size])
         axs[a].set_xticklabels([0, mean, field_size], rotation = 90)
         axs[a].axvline(x = mean, color = 'red')
         axs[a].legend(labels = [f"a = {a}"])

     plt.tight_layout()
     plt.show()
```

The distribution for $a = 0$ (on the left) is different from the distribution for $a = 1$ (on the right). What we need is to remove this statistical dependence by combining `tmp` with a random variable `r`:

```
[ ]: def recombNonLinears(nl0, nl1, r):
         return (nl0 ^ r) ^ nl1
```

```
[ ]: fig, axs = plt.subplots(1, field_size, sharex = False, sharey = True)

     for a in range(0, field_size):
         a1s = np.random.randint(0, fieldSize, nmasks)
         a0s = [ genShare(a, a1) for a1 in a1s ]
```

```
    b1s = np.random.randint(0, fieldSize, nmasks)
    b0s = [ genShare(0, b1) for b1 in b1s ]

    randoms = np.random.randint(0, fieldSize, nmasks)

    nonLinears = [ (bitwise_and(a0s[i], b1s[i]), bitwise_and(a1s[i], b0s[i]),␣
 ↪randoms[i]) for i in range(0, nmasks) ]

    recombs = [ recombNonLinears(*x) for x in nonLinears ]
    mean = np.mean(recombs)
    for x in recombs:
        axs[a].hist(genData(hw(x)), density = True, bins = 'auto', label = f"v =␣
 ↪{i}", alpha = 0.2)
    axs[a].set_xlim(-field_size - 4 + 1, field_size + 4 - 1)
    axs[a].set_xlabel("Value")
    axs[a].set_ylabel("Probability")
    axs[a].set_xticks([0, mean, field_size])
    axs[a].set_xticklabels([0, mean, field_size], rotation = 90)
    axs[a].axvline(x = mean, color = 'red')
    axs[a].legend(labels = [f"a = {a}"])

plt.tight_layout()
plt.show()
```

This time, the two distributions are nearly identical, as are their means. You can now modify the `secAND` implementation to introduce the correct recombination of the non-linear elements.

[ ]:
```
def secAND(a0, a1, b0, b1):
    # Linear elements
    c00 = bitwise_and(a0, b0)
    c11 = bitwise_and(a1, b1)

    # Non-linear elements
    c01 = bitwise_and(a0, b1)
    c10 = bitwise_and(a1, b0)

    # Recombine
    r = np.random.randint(0, 256, 1)
    tmp = recombNonLinears(c01, c10, r)

    c00 = bitwise_xor(c00, r)
    c11 = bitwise_xor(c11, tmp)

    return c00, c11

a0 = genShare(a, a1)
b0 = genShare(b, b1)
```

```
c0, c1 = secAND(a0, a1, b0, b1)
c = bitwise_and(a, b)

assert (c0 ^ c1) == c

print("Test passed!")
```

In general, when dealing with non-linear computations, their masking requires to deal with non-linear elements which combine shares with different indeces (e.g., a_0b_1).

The correct handling of such elements is crucial: any mistake might cause the preservation of a statical dependency with the inputs, opening the path to a successful side-channel attack.

A conservative strategy is to mask such non-linear elements with new uniform random values.

1.5 Masking Complex Algorithms

We now know how to mask simple linear and non-linear algorithms, or computations, where *simple* means *made of a single basic operation*. But what about algorithms made by more than one basic operations?

As we overviewed during the class, we can decompose the algorithm in several basic operations, mask such basic operations and compose back the algorithm.

Let us have a practical example with the `NAND` operation.

1.5.1 Masking the `NAND`

The `NAND` operation is simply the application of the `NOT` function to the `AND`'s output:

$$c = \texttt{nand}(a, b) = \texttt{not}(\texttt{and}(a, b))$$

Thus, it is made by the composition of the `AND` and the `NOT` function.

[ ]:
```
def bitwise_not(x):
    return x ^ 1

def bitwise_nand(a, b):
    c = bitwise_and(a, b)
    c = bitwise_not(c)
    return c
```

To provide its masked version, we can mask the two above functions and compose them in the order of the unmasked functions. Can you implement this `secNAND`?

For this, you will need to first provided the masked version `NOT`. Do not worry: `NOT` is a linear computation ;).

```
[ ]: def secNOT():
         assert False, "Missing implementation"
```

```
[ ]: def secNAND():
         assert False, "Missing implementation"
```

Let us test your implementation (i.e., it respect the *correctness* property):

```
[ ]: testValues = np.random.randint(0, 256, 1000, dtype = np.uint8)
     testMasks = np.random.randint(0, 256, 1000, dtype = np.uint8)
     testAmounts = np.random.randint(0, 9, 1000, dtype = np.uint8)

     for i in testValues:
         v = testValues[i]
         v1 = testMasks[i]
         v0 = genShare(v, v1)
         assert secNAND(a0, a1, b0, b1) == bitwise_nand(a, b), f"{i}: test failed."

     print("Test passed!")
```

If you implemented the `secNAND` correctly, you should get `Test passed!`.

1.6 Conclusion

In this laboratory activity, you went through the basics of the masking countermeasure: * You witnessed how an attacker can infer secret information from the probability distribution of the side-channel leakage * You witnessed how masking can counteract an attacker by removing the statistical relationship between the side-channel leakage and the processed sensitive information * You witnessed some generic strategies to mask any linear and non-linear algorithms * You developed masked versions of simple linear and non-linear algorithms

In the next laboratory, we will go through the masking of a full encryption routine, the `AES-128` and the means to break such secured version of the algorithm.