Countermeasures and Advanced Attacks

Brice Colombier and Vincent Grosso
Université Jean Monnet Saint-Etienne, CNRS
Institut d Optique Graduate School
Laboratoire Hubert Curien UMR 5516
F-42023, SAINT-ETIENNE, France

1 Introduction

In this chapter we present the main algorithmic countermeasures for protecting software implementations of cryptographic algorithms against side-channel attacks. This chapter is based on notions already covered in previous chapters, in particular chapters 3, 4, 5 and 6 and some knowledge of symmetric cryptography. We don't cover lower-level solutions such as adding noise with parallel computations, dual-rail logic aimed at balancing side-channel traces, or the use of asynchronous logic. Indeed, their impact is more difficult to quantify, and the exploitation of traces with these countermeasures generally requires signal processing techniques or the exploitation of different physical properties to foil the countermeasures.

To illustrate the various countermeasures presented in this chapter, we'll take as an example the AES [5] symmetric encryption algorithm and, more specifically, the first non-linear transformation of the encryption: the SubBytes transformation of the first round. This operation can be represented in pseudocode as in Algorithm 1. In this algorithm, `S-Box` is a mapping array storing the 256 possible output values of the AES S-box.

Algorithm 1 SubBytes Transformation

```
Input State (a $4 \times 4$ table, the current state of the AES).
Output State (a $4 \times 4$ table, the AES state after the SubBytes transformation).
1: for $i = 0; i < 16; i++$ do
2:     State[$i/4$][$i\%4$] = S-Box[State[$i/4$][$i\%4$]]
   return State
```

In this implementation, we can see that the state entries are considered sequentially. They are therefore manipulated at a precise point in time, depending solely on the loop index i. As seen in previous chapters, the attacker will link one or more points of the trace obtained by observing side-channel leakages to a sensitive data. In our example, this sensitive data is a byte of the S-box output.

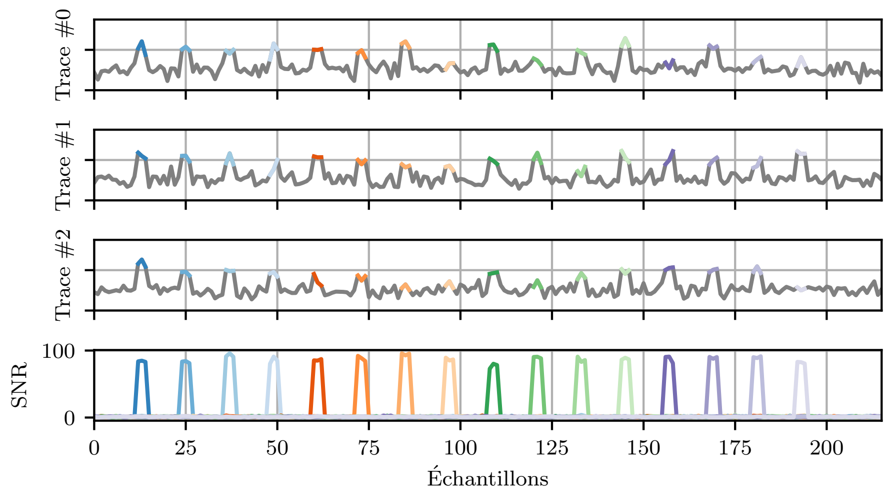

Figure 1: Examples of side-channel traces and SNR for an unprotected SubBytes transformation implementation.

For the sake of simplicity, we won't repeat the inputs/outputs in the following algorithms.

We will use modifications of Algorithm 1 to present the various countermeasures. We simulate the traces as corresponding to the Hammimg weight of the various State elements, to which Gaussian noise is added at several consecutive points corresponding to this information. The various elements are separated by points containing only noise. This simple model provides a good representation of microcontroller leakage. This simulation corresponds to the function `simulate_unprotected` in the attached notebook.

In Figure 1, we can see three examples of traces and the associated signal-to-noise ratio (SNR), at the bottom of the Figure, computed on the basis of 50 000 simulated traces, the SNR has been introduced in chapter 6. We have distinctively colored the different parts of the signal as a function of the byte, i.e. the i index of the loop, manipulated at a given time. The sequential manipulation of sixteen different values is clearly visible. We note that the SNR value associated with each byte is concentrated in a few points in the same identified zones corresponding to byte manipulation. These points therefore have information on the secret value manipulated, i.e. depending on the secret key, hence they are called "point of interest". A time sample corresponds to a point on the trace without distinction as to its "interest". Informative points are used when manipulating the data analyzed during an execution. It is therefore a notion specific to simulations. Points of interest, on the other hand, are detected by statistical analysis, such as SNR.

As a reminder, the SNR is a tool for detecting points of interest in a side-channel trace. These points are those which vary according to the value of the manipulated data, and which can therefore be considered as *informative*. The SNR can also be used to quantify the *intensity* of information leakage. A high SNR value indicates a significant information leak, and therefore an attack that is easier to carry out, i.e. requiring fewer traces. There are a number of criteria for comparing the effectiveness of an side-channel attack, such as the number of traces required, the average rank of the correct key hypothesis, time, a priori knowledge of the system, and so on. In this chapter, we evaluate the effectiveness of an attack by the number of traces needed to achieve a success rate close to 1.

In the rest of the chapter, we present some countermeasures against side-channel attacks. Algorithms 2, 3 and 4 will thus be modifications of Algorithm 1. We detail the impact of these countermeasures on the traces and on the SNR value. We also briefly discuss the additional cost of these countermeasures in terms of execution time and computation overheads required. Finally, we'll look at the attacks an attacker can mount when some of these countermeasures are implemented.

2 Misalignment of traces

One of the first steps in side-channel attacks is the selection of points of interest in the trace, so as to retain only those where there is information that depends on the secret to be recovered, as introduced in chapter 5. One idea frequently employed to reduce the effectiveness of side-channel attacks is therefore to misalign the traces from one run to the next. The scattering of informative points over time affects trace analysis, which is carried out independently for each time sample, i.e. vertically on the figures. As a result, since the information is now dispersed according to the traces considered, the information contained at a given time sample in the trace is reduced. The SNR value obtained at each point is therefore also reduced. It is well known that the success of a first-order attack depends on the SNR (as shown in chapter 6), hence the interest in reducing it to make side-channel attacks more complex.

2.1 Countermeasures

In this section, we'll introduce three countermeasures designed to de-synchronize sensitive operations, i.e. to ensure that, considering several consecutive traces, the points of interest where sensitive data are manipulated are at different time samples.

2.1.1 Random initial delay

The aim of a countermeasure using a random initial delay is to break the alignment between time samples from consecutive traces. This alignment is required for the realization of attacks, which consider the leakage of information at a

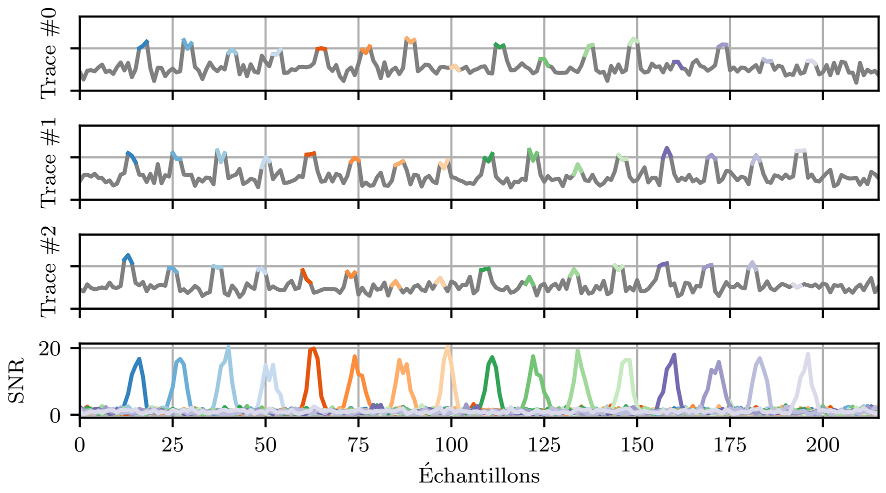

Figure 2: Examples of side-channel traces and SNR for an implementation of the SubBytes transformation protected by the addition of a random initial delay.

given time on a set of traces. In order to desynchronize the traces, the start of the algorithm execution is delayed. This is illustrated in Algorithm 2, where wait is a procedure that pauses the system for a time defined by its argument, and rand is a function generating a random number.

Algorithm 2 SubBytes transformation protected by adding a random initial delay

```
1: $r = \mathsf{rand}()$
2: $\mathsf{wait}(r)$
3: for $i = 0; i < 16; i++$ do
4:     $State[i/4][i\%4] = \texttt{S-Box}[State[i/4][i\%4]]$
   return State
```

A random initial delay countermeasure has an effect comparable to poor trace slicing around the target operation, or poor sensitization of the acquisition trigger signal. In all cases, while the informative parts of the traces are within a fixed and restricted interval $[t_0, t_0 + \Delta_t]$, the start of this time interval t_0 is not the same for all traces.

The impact of the random initial delay can be seen on the simulated traces in Figure 2. We note a horizontal misalignment of the traces. Whereas in the examples of the non-protagged case in Figure 1, we can see that the different traces were well aligned, in the case with a random initial delay we notice that we have a temporal shift of the traces.

The introduction of this countermeasure has several consequences for SNR.

On the one hand, the maximum value of the SNR is lower because the SNR is a univariate measure. Also, the countermeasure will mix informative and non-informative points in the same temporal position. On the other hand, the number of points that are considered informative is higher: the window of attack is wider.

When an implementation is protected by a random initial delay, the additional cost is limited. The execution time is increased by the added delay. Concerning the required randomness, only one random number is required per execution.

The practical effectiveness of this countermeasure is limited by the fact that the pattern that characterizes the execution of the sensitive part of the encryption algorithm, in this case the SubBytes transformation, remains identical. It is easy to identify this recurring pattern in a set of traces and resynchronize them using methods such as the convolution product (i.e. in the frequency domain), or cross-correlation.

2.1.2 Random delay interrupts

This countermeasure consists in inserting random delay interrupts between sensitive operations. These additional instructions do not modify the internal state of the device: they are sometimes referred to as *dummy operations*. To illustrate these *dummy operations*, we'll use some NOP instructions. The instruction NOP, for *no operation*, does nothing except increment the instruction pointer.

Algorithm 3 SubBytes transformation protected by insertion of random delay interrupts

```
1: for $i = 0; i < 16; i++$ do
2:     $r = rand()$
3:     for $j = 0; j < r; j++$ do
4:         NOP
5:     $State[i/4][i\%4] = \texttt{S-Box}[State[i/4][i\%4]]$
   return State
```

By inserting these instructions, the traces will become increasingly desynchronized during execution.

Inserting several small desynchronizations throughout execution makes resynchronizing traces with signal processing techniques more complex.

In Figure 3, we can observe simulated side-channel traces where random interrupts have been inserted.

When an implementation is protected with random delay interrupts, the extra cost is higher than with a countermeasure with random delay only. As far as the required randomness is concerned, the countermeasure requires sixteen random values this time, instead of just one previously. But this increase in the number of delays also makes trace resynchronization operations more complex. The execution time of the algorithm is not necessarily greatly increased.

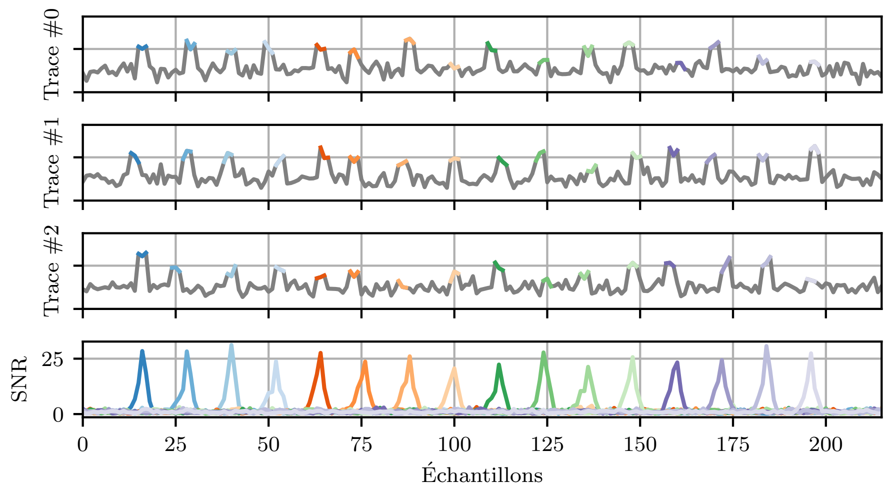

Figure 3: Examples of consumption and SNR traces for an implementation of the SubBytes transformation protected by insertion of random delay interrupts.

These temporal countermeasures are particularly effective against attacks requiring temporal precision.

2.1.3 Shuffling independent operations

Another possible countermeasure, designed to disperse points of interest, is to *shuffle* independent operations. The principle is to execute certain operations of the algorithm, which have the property of being independent, in a random order. In the case of the AES SubBytes transformation, the same operation is repeated sixteen times: reading a value from memory, substitution, writing a value to memory. The various iterations of this operation are independent of each other. Thus they can be performed in any random order without altering the final result. A pseudo-code for the countermeasure is written in Algorithm 4, where `shuffle` is a function that randomly shuffles an array that is passed as input.

The main advantage of shuffled operations, as opposed to previous countermeasures based on delay insertion, is that the general pattern observable in successive traces remains *identical*. This makes it difficult for an attacker to use signal processing techniques to realign the points in successive traces corresponding to the manipulation of the same intermediate value.

Algorithm 4 Transformation SubBytes protected by shuffling independent operations

```
1: $S = \{0, 1, \ldots, 15\}$
2: $S = \mathsf{shuffle}(S)$
3: for $i = 0; i < 16; i++$ do
4:     $j = S[i]$
5:     $State[j/4][j\%4] = \texttt{S-Box}\,[State[j/4][j\%4]]$
   return State
```

Figure 4 represents the traces associated with the countermeasure of shuffling independent operations. In this figure, we're only showing the operations that take place from the line 3 of Algorithm 4. We therefore ignore the part where the array is permuted to define the random order in which operations will be performed.

Figure 4 shows the impact of shuffling independent operations on simulated traces. We can see that the traces are well aligned, unlike countermeasures using random initial delays or unnecessary operations. What's more, we can see that at a given point in time, it's impossible to know which index is being used. Thus, the leaks corresponding to the sixteen independent operations are effectively superimposed.

Because of its impact on the traces, where any byte (index j in the loop) can be used at any time, the shuffling of independent operations behaves in a similar way to parallel hardware implementations, where all operations are carried out simultaneously. When the attacker is unable to exploit any information about the permutations used in the various executions, the number of traces needed to carry out an attack is multiplied by the number of operations required, in the case of AES SubBytes by 16.

In order to increase the number of operations available to swap the order of execution, instructions that are unnecessary for the desired calculation can be added. These unnecessary operations will disperse the information into more positions. This is particularly useful when the number of similar operations is small. Following the example of AES, the MixColumns transformation operates on the columns of the internal state. As there are only 4, the number of operations to be shuffled is too small and a brute-force attack is possible.

The overhead for a countermeasure using a shuffling of independent operations is mainly linked to the management of index permutation. There are several methods for managing a good permutation efficiently and with a negligible bias. We give a brief reference to this in the section 5 "To go further".

2.2 Attacks

An attacker is considered as someone wishing to carry out a univariate attack, such as a correlation power analysis (CPA) attack.

To attack implementations with countermeasures based on delay insertion, an attacker can proceed as if no countermeasure had been implemented. Here, the attacker will only consider the most informative point to find the correct

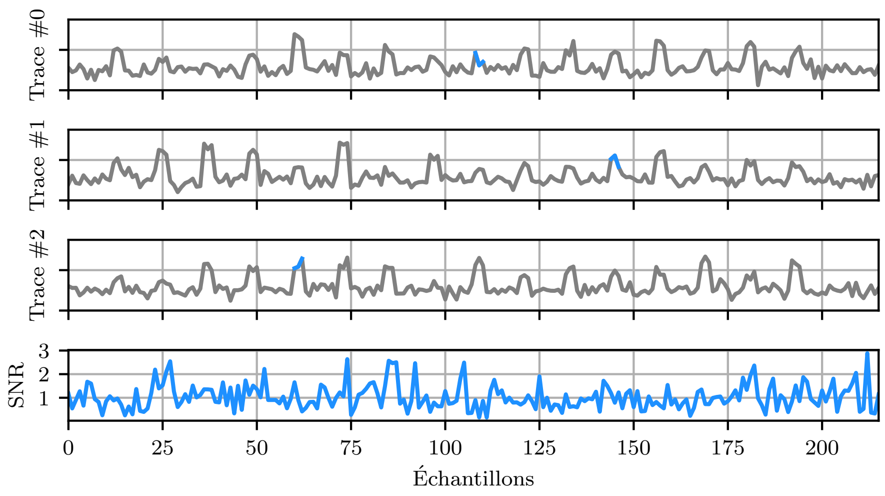

Figure 4: Examples of side-channel traces and SNR for a shuffling-protected implementation of the SubBytes transformation.

sub-key. In this case, other informative points not aligned with the first will be ignored.

To measure the effectiveness of the countermeasure, we can consider that the traces necessary for the attack are divided into two groups. In the first, we find the traces for which the point of interest corresponds to an informative point. The second set includes traces where the point of interest does not correspond to an informative point. If we consider that the countermeasure distributes information uniformly over m different time samples, then the first set will be $m - 1$ times less populated than the second. Since it is this first set alone that contributes to the success of the attack, an attacker will therefore need at least $(m - 1) \times N$ traces to achieve first-order success equivalent to an attack achievable with N traces on an unprotected implementation. What's more, the points in the second set are noisy, so the previous reasoning underestimates the number of traces.

This can be observed by comparing the evolution of the corrrelation curves of the CPA attack without countermeasures with those of the CPA attacks with delays in the attached notebook.

In the case of countermeasures based on delay insertion, the dispersion of informative points can be significant. The attack may then require a number of traces greater than the number of informative points and the number of points on which they can be found. If we have one informative point and it can be dispersed over 5 time samples, then the attack on the implementation with protection will require 5 times as many traces. Let's imagine there's an oracle that can sort traces where the informative point is at time t_0. If the countermeasure evenly distributes the informative points, the oracle will only see a fifth of the traces. On the other hand, the dispersion brought about by these countermeasures can be reduced by signal processing techniques. These techniques exploit the repetition of target instruction patterns in the trace to realign the traces.

In the case of countermeasures based on the shuffling of independent operations, the dispersion of the points of interest is limited by the number of independent operations, which is often smaller than the typical value of the delay or random interruption. On the other hand, the signal processing techniques described above are more difficult to apply in this case.

A second method of attacking protected implementations is to use the so-called *integration* technique. The idea this time is to no longer consider a single point as informative, but a set of points. For countermeasures based on delay insertion, we'll consider a window of width w and create a new trace T_N from the initial trace T_R. Each index sample i of the new trace T_N corresponds to the sum of w index samples i to $i + w - 1$ of the raw trace T_R. In general, starting from a raw trace with n points, if the attacker wants to perform integration with a window of size w, then he must perform the following transformation:

$$\forall i \in \{0, n-w\}, T_N[i] = \sum_{j=0}^{w-1} T_R[i+j].$$

Using the new trace T_N, the attacker can then carry out a classic attack. The advantage of the integration method is that the information points, if close enough to be included in the width window w, are no longer out of sync. It has been shown that when integration is used, the number of traces required for a successful attack is multiplied by a factor w , where w is the size of the integration window. This is due to the fact that integration will add points with no information and only noise, if there is a single informative point in the window.

We consider only the impact for unprofiled attacks. However, the impact of shuffling can be reduced when an attacker also exploits permutation generation with profiled attacks.

In the end, we can see that these countermeasures are a compromise between their cost in terms of execution and time, and their impact on the temporal dispersion of informative samples. This temporal dispersion is the basis of the effectiveness of these countermeasures.

3 Masking

The noise inherent in the operation of electronic circuits makes attacks more difficult, since it masks to some extent the relationship between the secret data manipulated and the trace obtained by observation of side-channels. The aim of masking is to force the attacker to combine several points on the trace in order to carry out the attack. As a result, the attacker will combine and amplify the noise at each point to retrieve the information he requires.

These methods are discussed in more detail in the relevant chapters of the book, and we refer the reader to part 1 of volume 2 for further details.

3.1 Countermeasures

The first countermeasures we saw were designed to shift the time at which a sensitive operation is carried out. There are other countermeasures designed not to de-synchronize traces, but to make operations non-sensitive, i.e. not directly manipulating sensitive data.

To achieve this, sensitive data is divided using a method known as secret sharing. The sharing outputs by the method is different for each trace and it is uniformly randomly chosen from the set of possible secret sharing. Secret sharing consists in dividing a secret x into d shares $\{x_i\}_{i=1}^{d}$, so that if an attacker observes $d-1$ shares, he obtains no information about the secret x. Only the possession of all d shares can reveal the original x secret.

The secret sharing used for the encryption algorithms we'll be considering here is most often a *boolean* sharing. This is the one we're going to describe. In this case, the secret x is shared in d independent shares. To do this, $d-1$ shares are chosen randomly and the last one is computed so that the relationship given in equation 1 is satisfied.

$$x = x_1 \oplus x_2 \oplus \cdots \oplus x_d \tag{1}$$

Thus, any set of $d-1$ shares is independent of the secret x : it's as if a *one-time pad* had been applied to the secret. Nevertheless, if a person knows the d shares, he or she can reconstruct x.

The next question is how to perform calculations on shared secrets. One solution is to recalculate the substitution table to take into account the masked data and refresh the mask as described in Algorithm 5.

Algorithm 5 Masked SubBytes transformation with table recalculation

Input State a table $4 \times 4 \times (d)$, the current state of the masked AES, d the number of shares (here $d = 2$).

Output State a table $4 \times 4 \times (d)$, AES status after SubBytes transformation.

```
1: for $i = 0; i < 16; i++$ do
2:     $r = rand()$
3:     $x_2 = \text{State}[i/4][i\%4][1]$
4:     for $j = 0; j < 256; j++$ do
5:         $\texttt{MS[j]} = \texttt{S-Box}[j \oplus x_2] \oplus r$
6:     $\text{State}[i/4][i\%4][0] = \texttt{MS}[\text{State}[i/4][i\%4][0]]$
7:     $\text{State}[i/4][i\%4][1] = r$
   return State
```

In Algorithm 5, a new table is computed to access the masked value. An input mask and an output mask are used to calculate all possible outputs according to the masked value. During this pre-computation, only one input mask and one output mask are used. In this way, there can be no leakage of information. Then the table is used with the second input mask to calculate the masked output of the S-box. As the pre-calculated table unmasks the input, calculates the output value and masks the output, when evaluating with the pre-computed table, no information can be obtained by observing the output mask. This ensures that both output masks form a partition of the S-box output.

The advantage of masking is that the secret is never manipulated directly at any time during the execution. Information leaks through side-channels are therefore independent of secret data .

The extra cost of masking is significant. On the one hand, it is linked to the generation of random numbers. So $d-1$ random bytes are needed per secret and per table to be recomputed. The extra cost in terms of execution time is also significant, particularly for table recomputation. There are other, more advanced methods for limiting this overhead. This can be achieved by exploiting sharing methods based on field operations on which computations are performed. However, masking remains much more costly than the desynchronization methods described in section 2.

3.2 Attacks

To attack an implementation protected by a masking countermeasure, the attacker will have to combine several points in the trace in order to recover the secret. As mentioned above, this will entail combining measurement errors. It's this combination of time samples, and therefore noise, that gives masking its security. It can be shown that masking has an exponential efficiency in the number of shares used in sharing over the number of traces needed to carry out an attack. If N traces were needed to carry out an side-channel attack on an unprotected implementation, it would take on the order of $N \times \sigma^{2d}$ to attack the protected implementation, where d is the number of shares in the sharing and σ is the standard deviation of the noise in the traces. In particular, we can perform a theoretical information analysis and show that information decreases exponentially. This means that the effectiveness of all attacks is impacted by masking.

To carry out an unprofiled CPA attack, the attacker must pre-process the traces in order to combine the points in the traces that are informative about each of the pieces of the share. There are several combination functions C available for this purpose. From the original trace T_R of size n and this combination function, the attacker will create a new trace of size n^d. For ease of reading, we give the formula for $d = 2$ in the equation 2, its generalization to higher d values being trivial.

$$\forall (i, j) \in \{0, n-1\}^d, T_N[i \times n + j] = C(T_R[i], T_R[j]). \tag{2}$$

The most frequently used combination functions are described below. Their effectiveness varies according to the level of noise in the traces.

- Centered product: for this combination function, it is necessary to compute the average of each point of the traces. For N attack traces $\{T_i\}_{i=1}^{N}$, the attacker calculates the average of the traces point by point $M_p = \frac{1}{N}\sum_{i=1}^{N} T_i[p]$, then constructs the new traces with the combination function $C(T[p_1], T[p_2]) = (T[p_1] - M_{p_1}) \times (T[p_2] - M_{p_2})$.
- Manhattan distance: this is the absolute value of the difference between the points: $C(T[p_1], T[p_2]) = |T[p_1] - T[p_2]|$.

The attacker can carry out the classic non-profiled attacks on this new trace.

For high noise levels, the centered product is the most efficient combination, and leads to a smaller number of necessary traces for the attack to be successful. Conversely, for lower noise levels, the absolute value of the difference or the sum of the points are more effective.

4 Combination of countermeasures

As we've just seen, masking provides a high level of resistance against side-channel attacks. However, for masking to be effective, it needs sufficient noise. We've also seen a number of countermeasures that increase noise. The question then arises: how can these countermeasures be combined?

The most effective countermeasure for adding noise, and one that still stands up to signal processing techniques, is the shuffling of independent operations. So it's natural to want to combine this countermeasure with masking. It can be shown that if shuffling of independent operations is applied to the pieces for each operation, masking amplifies the noise introduced by shuffling. This is described in Algorithm 6.

Algorithm 6 SubBytes transformation protected by masking with table recalculation and shuffling

Input State a table $4 \times 4 \times (d)$, the current state of the AES being masked, d the number of pieces (here $d = 2$).

Output State a table$4 \times 4 \times (d)$, AES status after SubBytes transformation.

```
1: $S = \{0, 1, \ldots, 15\}$
2: $S = \mathsf{shuffle}(S)$
3: for $i = 0; i < 16; i++$ do
4:     $\mathtt{R}[S[i]] = rand()$
5: for $j = 0; j < 256; j++$ do
6:     $S = \mathsf{shuffle}(S)$
7:     for $i = 0; i < 16; i++$ do
8:         $x_2 = \text{State}[S[i]/4][S[i]\%4][1]$
9:         $\mathtt{MS}[S[i]][j] = \mathtt{S\text{-}Box}[j \oplus x_2] \oplus \mathtt{R}[S[i]]$
10: $S = \mathsf{shuffle}(S)$
11: for $j = 0; j < 16; j++$ do
12:     $i = S[j]$
13:     $\text{State}[i/4][i\%4][0] = \mathtt{MS}[i][\text{State}[i/4][i\%4][0]]$
14: $S = \mathsf{shuffle}(S)$
15: for $j = 0; j < 16; j++$ do
16:     $i = S[j]$
17:     $\text{State}[i/4][i\%4][1] = \mathtt{R}[i]$
    return State
```

In Algorithm 6, two countermeasures are combined: masking with table recomputation and operation shuffling. For this, the 16 tables are recomputed at the same time and in a different order for each of the 256 outputs, to make it difficult for an attacker to combine the different information on the same input mask. Secondly, these different tables are also used in random order. This eliminates the need for memory to store the recalculated tables.

If the shuffling of independent operations is applied to secret shares (i.e. only to loop indices), then masking doesn't amplify the noise introduced by shuffling, but this solution is far more efficient in terms of calculation and random numbers to manage.

This brings us back to the main discussion. When a developer wants to implement a countermeasure to protect an implementation, what security can be expected and at what extra cost?

5 To go further

In this chapter we've tried to give an intuition of countermeasures against side-channel attacks. For the interested reader, we've provided a few references to further explore the various points made in this chapter.

- For countermeasures with random time interrupts, we recommend [4], and to explore countermeasures and attacks in more detail [7].
- For the countermeasure using the shuffling of independent operations, a detailed study is proposed in [10].
- For table recalculation-based masking, we cite [3] and for demonstrations of the exponential advantage of countermeasure we recommend [6].
- For evaluation of countermeasures and demonstrations of the benefits of various countermeasures, we refer to [8]. For analysis of the combination of independent operations and masking [1].
- For combination functions, we refer the reader to [9] and to [2] for proofs of the optimality of combination functions.

References

[1] Melissa Azouaoui et al. "Bitslice Masking and Improved Shuffling: How and When to Mix Them in Software?" In: *IACR Cryptol. ePrint Arch.* (2021), p. 951. URL: https://eprint.iacr.org/2021/951.

[2] Nicolas Bruneau et al. "Masks Will Fall Off - Higher-Order Optimal Distinguishers". In: *Advances in Cryptology - ASIACRYPT 2014 - 20th International Conference on the Theory and Application of Cryptology and Information Security, Kaoshiung, Taiwan, R.O.C., December 7-11, 2014, Proceedings, Part II*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. Lecture Notes in Computer Science. Springer, 2014, pp. 344–365. ISBN: 978-3-662-45607-1. DOI: 10.1007/978-3-662-45608-8_19. URL: https://doi.org/10.1007/978-3-662-45608-8%5C_19.

[3] Jean-Sébastien Coron. "Higher Order Masking of Look-Up Tables". In: *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, 2014, pp. 441–458. DOI: 10.1007/978-3-642-55220-5_25. URL: https://doi.org/10.1007/978-3-642-55220-5%5C_25.

[4] Jean-Sébastien Coron and Ilya Kizhvatov. "Analysis and Improvement of the Random Delay Countermeasure of CHES 2009". In: *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, 2010, pp. 95–109. ISBN: 978-3-642-15030-2. DOI: 10.1007/978-3-642-15031-9_7. URL: https://doi.org/10.1007/978-3-642-15031-9%5C_7.

[5] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. ISBN: 3-540-42580-2. DOI: 10.1007/978-3-662-04722-4. URL: https://doi.org/10.1007/978-3-662-04722-4.

[6] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, 2014, pp. 423–440. DOI: 10.1007/978-3-642-55220-5_24. URL: https://doi.org/10.1007/978-3-642-55220-5%5C_24.

[7] François Durvaux et al. "Efficient Removal of Random Delays from Embedded Software Implementations Using Hidden Markov Models". In: *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*. Ed. by Stefan Mangard. Vol. 7771. Lecture Notes in Computer Science. Springer, 2012, pp. 123–140. ISBN: 978-3-642-37287-2. DOI: 10.1007/978-3-642-37288-9_9. URL: https://doi.org/10.1007/978-3-642-37288-9%5C_9.

[8] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. "Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers". In: *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, 2009, pp. 171–188. ISBN: 978-3-642-04137-2. DOI: 10.1007/978-3-642-04138-9_13. URL: https://doi.org/10.1007/978-3-642-04138-9%5C_13.

[9] François-Xavier Standaert et al. "The World Is Not Enough: Another Look on Second-Order DPA". In: *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 5-9, 2010. Proceedings*. Ed. by Masayuki Abe. Vol. 6477. Lecture Notes in Computer Science. Springer, 2010, pp. 112–129. ISBN: 978-3-642-17372-1. DOI: 10.1007/978-3-642-17373-8_7. URL: https://doi.org/10.1007/978-3-642-17373-8%5C_7.

[10] Nicolas Veyrat-Charvillon et al. "Shuffling against Side-Channel Attacks: A Comprehensive Study with Cautionary Note". In: *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, 2012, pp. 740–757. ISBN: 978-3-642-34960-7. DOI: 10.1007/978-3-642-34961-4_44. URL: https://doi.org/10.1007/978-3-642-34961-4%5C_44.