Variety of Scalable Shuffling Countermeasures against Side Channel Attacks

Nikita Veshchikov, Stephane Fernandes Medeiros
and Liran Lerman

Department of Computer Sciences, Université libre de Bruxelles,
Brussel, Belgium
E-mail: {nveshchi; stfernan; llerman}@ulb.ac.be

Received 1 April 2017; Accepted 14 June 2017;
Publication 12 July 2017

Abstract

IoT devices have very strong requirements on all the resources such as memory, randomness, energy and execution time. This paper proposes a number of scalable shuffling techniques as countermeasures against side channel analysis. Some extensions of an existing technique called Random Start Index (RSI) are suggested in this paper. Moreover, two new shuffling techniques Reverse Shuffle (RS) and Sweep Swap Shuffle (SSS) are described within their possible extensions. Extensions of RSI, RS and SSS might be implemented in a constrained environment with a small data and time overhead. Each of them might be implemented using different amount of randomness and thus, might be fine-tuned according to requirements and constraints of a cryptographic system such as time, memory, available number of random bits, etc. RSI, RS, SSS and their extensions are described using SubBytes operation of AES-128 block cipher as an example, but they might be used with different operations of AES as well as with other algorithms. This paper also analyses RSI, RS and SSS by comparing their properties such as number of total permutations that might be generated using a fixed number of random bits, data complexity, time overhead and evaluates their resistance against some known side-channel attacks such as correlation power analysis and template attack. Several of

Journal of Cyber Security, Vol. 5_3, 195–232.
doi: 10.13052/jcsm2245-1439.532

proposed shuffling schemes are implemented on a 8-bit microcontroller that uses them to shuffle the first and the last rounds of AES-128.

Keywords: Side channel analysis, countermeasures, hiding techniques, shuffling countermeasure, microcontroller, AES, lightweight shuffling.

1 Introduction

One of the most fast and wide spreading domains in modern digital world is the domain of mobile connected devices called the Internat of Things (IoT). This world is composed of embedded systems, small portable devices such as smartcards or microcontrollers. These interconnected devices are distributed among their users who can also be potential attackers, thus security of IoT is an important issue. A new type of attacks become very important in this context: attacks where the adversary has access to the attacked device. These attacks are among the most powerful attacks on cryptographic implementations, and they represent real-world thread to the IoT devices. Side channel attacks (SCA) are among the most efficient and strongest attacks of this type. Instead of targeting the algorithm (an abstraction), they focus on their implementations (real, physical devices).

Since side channel attacks on implementations of cryptographic algorithms were introduced to the scientific community [8, 9] a number of different countermeasures were suggested and studied in literature. Reordering of independent operations, generally referred as shuffling, was suggested as one of the possible countermeasures against side channel attacks [7, 18]. Small embedded devices have lots of constraints such as time, power consumptions, amount of memory that might be used or amount of random bits that a Random Number Generator (RNG) might generate per encryption. Thus, often lightweight security and lightweight countermeasures are privileged for implementations in such environments. Even less constrained environments often have strong requirements on parameters such as e.g., high throughput. Some countermeasures might heavily affect the execution speed of an algorithm (e.g., increase it by a factor of 6 or 7) [13], so even in such environments relatively lightweight countermeasure that would not heavily affect the performance of the device would often be the designer's choice.

This work proposes a set of scalable shuffling techniques that can be fine tuned to fit specific requirements and constraints of a given application. The designer can use our schemes according to the resources that are available in the system including random numbers, timing constraints and available memory.

1.1 Related Works

Shuffling has already been studied in the literature. Random Start Index (RSI) was applied on AES block cipher by Herbst *et al.* [7] and by Tillich *et al.* [21]. In their implementation of AES, operations were executed in a sequential order but with a different randomly chosen starting index: RSI starts by choosing a column index, this index will be the index of the first column to be processed (other columns will be processed sequentially). Furthermore, a second index is chosen for the starting line, for all of the columns. In their schemes, shuffling was combined with masking and applied only on the initial AddRoundKey, SubBytes and MixColumns of the first round and on MixColumn and AddRoundKey of round 9 as well as on SubBytes and AddRoundKey of the last round. RSI shuffling technique is very lightweight and may be easily implemented with virtually no overheads [14, 15]. The basic versions of RSI such as used by Tillich *et al.* [21] requires 4 random bits. We suggest several extensions of the RSI shuffling technique, our extensions are scalable i.e., they are flexible in terms of the number of random bits that might be used in order to shuffle instructions (e.g., from 1 up to 10 random bits in case of AES-128).

Fully random permutation, inside AES operations, was suggested by Tillich *et al.* [21] and applied by Rivain *et al.* [18]. They also combined masking and shuffling countermeasures.

Veyrat-Charvillon *et al.* [23] also studied the basic RSI shuffling technique and suggested improvements to Random Permutation (RP) shuffling technique by manipulating the program's control flow. RP technique allows to reorder all 16 bytes of one operation (such as SubBytes) of AES-128, it might generate all possible 16! permutations (which is roughly $2^{44.25}$) using 64 random bits [23]. This technique is not as lightweight as RSI but is able to generate much more permutations (all of them). Our shuffling schemes require less randomness than RP, but they also produce less permutations. Our shuffling techniques are as lightweight as RSI, but they allow to produce more than 16 different orderings.

RP shuffling technique was also suggested as an improvement (to be used in combination with masking) for the DPA contest v4 [1, 17] and was used for the DPA contest v4.2. Two different algorithms were suggested for the generation of the random permutation, one of them generates full entropy but it has higher big O complexity while the other is less computationally expensive but results in lower entropy [1].

Fernandes Medeiros [13] introduced a shuffling technique that he called SchedAES, it randomizes the sequence of instructions of the AES over several operations. This countermeasure takes precedence relations between operations into account in order to decide which instruction could be executed next. It allows to generate many different orderings of operations. Unfortunately, this technique requires additional data structures, a lot of random bits per execution (up to 3 000 per execution) and could not be used in very constrained environments.

All shuffling algorithms require randomness as input in order to generate a permutation, most of them are rigid and require a fixed amount of random bits for example, technique proposed by Veyrat-Charvillon *et al.* [23] for Random Permutation shuffling requires 64 random bits. The algorithm used by Tillich *et al.* [21] needs 4 random bits. Our shuffling algorithms allow to choose the number of random bits and thus allow to tune the implementation according to constraints and requirements of the system as well as to the amount of available resources.

1.2 Notations

We are going to use terms *shuffling technique* to denote a method or an algorithm that allows to reorder operations (instructions) of an algorithm without changing its final result. The term *shuffle* will be used to represent one possible ordering (one permutation) of all operations (or instructions) that are being shuffled. *Number of shuffles* will represent the total number of all possible different shuffles that can be generated using a particular shuffling technique.

The abbreviations LSB and MSB denote the least significant bit and the most significant bit respectively, LSBs and MSBs will be used for several least and most significant bits. We use terms *second LSB* and *second MSB* to denote the bit next to the LSB and MSB respectively e.g., if the index of the LSB of a byte is 0 and the index of the MSB is 7 then the index of the second LSB is 1 and the index of the second MSB is 6. Terms *third LSB (MSB)* or *fourth LSB (MSB)* are used in the same manner.

We will refer to a random number generator used by a cryptographic system as RNG.

Since it is possible to shuffle one or several rounds of an algorithm as well as several operations per round and each of these operations might be done in one or several clock cycles (depending on the used hardware), we are going to use the term *random bits available per unit of time* where the unit of time might represent one clock cycle, one operation, one round or one encryption.

1.3 Structure of This Paper

The rest of this paper is organized as follows. Section 2 presents our new shuffling schemes as well as their extensions using SubBytes operation of AES-128 block cipher as an illustration. Section 3 compares our shuffling techniques among them as well as with couple of other techniques from several points of view. Section 4 sums up the analysis and discusses our results. Finally, Section 5 concludes this paper and gives a list of suggestions for further improvements and future works.

2 Scalable Shuffling Techniques

Here we are going to describe 3 scalable shuffling techniques. For sake of simplicity all examples presented in this section are given for the `SubBytes` operation (application of the S-box) on the state of 128-bit version of AES block cipher. This section presents shuffling techniques on the example of the first round of AES, but same shuffling techniques might be applied to any number of rounds depending on system's requirements and amount of available resources (time, memory, amount of random bits, etc). All presented shuffling schemes might be easily adapted for other operations of AES as well as for other algorithms.

Most of the shuffling techniques suggested in this section are based on the fact that the internal state might be seen as a vector or as a matrix. Indeed, in the memory of a computer, a vector of size 16, a 4×4 matrix or even a 2×8 matrix are all just tables of 16 memory units (in our case bytes).

2.1 Random Start Index

Basic version Random Start Index (RSI) shuffling for AES-128 represents the AES state as a vector of 16 elements. S-box is applied to all 16 bytes one by one starting from a randomly chosen index (between 0 and 15). This shuffling technique requires 4 random bits and it gives us 16 possible starting indexes (and 16 different shuffles in total).

Two different variations of the basic RSI technique might be implemented, those techniques generalize RSI and might be applied with less or more random bits (between 1 and 10 bits in our AES-128 examples).

Vector-RSI

Vector RSI (V-RSI) extension, uses the same representation of the AES-128 state as the basic RSI, the state is used as a vector and a random start index

might be chosen with less than 4 bits of randomness. It might be done by giving a fixed value to all missing bits, by reusing some of the available random bits (eventually by combining them) or even by combining these two approaches, see Figure 1.

For example, if we have only 3 available random bits for the RSI, we can fix the position of the missing one as the LSB of the starting index and always assign its value to 0. In this case we will have 8 possible shuffles with only even numbers as starting indexes, see Figure 2(a).

Here is another example, lets say we have only 2 random bits per unit of time and we would like to use V-RSI shuffling. We can fix those random bits as two LSBs of the index, fix the value of the MSB of the index to 1 and assign the value of the second MSB to the exclusive-or (XOR) of the two available random bits. If we use this algorithm to generate the starting index we will be able to generate 4 different shuffles that might start with indexes 8, 11, 13 or 14, see Figure 2(b). This last example is not

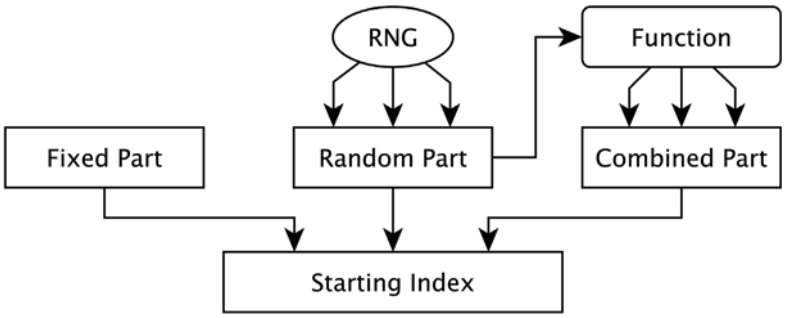

Figure 1 Structure of V-RSI index generation. The order of bits coming from different parts might be chosen arbitrarily.

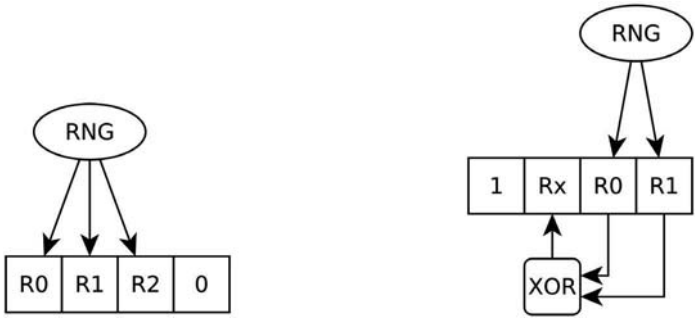

Figure 2 Example of V-RSI use with 2 and 3 available random bits.

practical, especially in software implementations, since it requires additional computations. Nevertheless, it is worth noting that we can actually choose any starting index for an implementation by choosing how to assign values to missing bits that are needed in order to have a random index.

An idea of using 3-part computation (fixed part, random part and combined part) in order to generate a random index might be used in a scenario when the shuffling scheme used by the device is the same for all devices, but each *instance* is different thanks to different combination functions and different fixed parts. In such scenario each device will use a different shuffling, thus an attacker will have less chances of being able to profile one device in order to attack another one.

Matrix-RSI

The second extension, that we call Matrix RSI (M-RSI), of the RSI technique handles the internal state of AES as a matrix and treats it row by row. Since the state is handled row by row, we can just apply the V-RSI on each row. Since all rows are handled separately, we can also start with any row i.e., we can reuse V-RSI technique in order to choose the starting row. This technique might allow us to shuffle SubBytes operation with 1 to 10 random bits and can give us from 2 to 1024 possible shuffles.

Table 1 shows how M-RSI might be applied on AES-128 state depending on the number of available random bits per unit of time. This table is structured as follows, *All rows* part shows how we can go through all rows and *Cells in a row* part shows how we may handle all cells in one row. Following notations are used: Fixed – normal, non-random algorithm is used (e.g., 0, 1, 2, 3); *Rand(n)* – starting index is chosen using n random bits; S – same random numbers are used to get the starting index in each row, D – different random bits are used to generate the starting index in different rows. For example, if we have 6 available random bits and we want to use M-RSI, according to the table we might use 2 bits in order to choose a random row to start with and we can also use 1 bit per row in order to choose a random starting index in each row (using V-RSI with 1 bit on a vector of 4 bytes).

Notice that this table only gives some examples of how to use M-RSI per number of available bits, multiple combinations might be implemented for some numbers e.g., for 4 bits we might also use 2 bits in order to choose a starting row and then use 2 random bits in order to choose a random start cell (same in each row). Some of these choices might be more efficient and/or more secure than others.

Table 1 Examples of M-RSI use on 4 × 4 AES-128 state using different number on random bits

	All Rows		Cells in a Row		
Available Random Bits	Bits	Handling	Bits	Handling	Number of Shuffles
1	1	Rand(1)	0	Fixed	2
2	2	Rand(2)	0	Fixed	4
3	2	Rand(2)	1	Rand(1), S	8
4	0	Fixed	4	Rand(1), D	16
4*	2	Rand(2)	2	Rand(2), S	16
5	1	Rand(1)	4	Rand(1), D	32
6	2	Rand(2)	4	Rand(1), D	64
8	0	Fixed	8	Rand(2), D	256
9	1	Rand(1)	8	Rand(2), D	512
10	2	Rand(2)	8	Rand(2), D	1024

*The second version with 4 bits offers more security, see Section 3 and Table 5.

Unfortunately, we were not able to find a "nice" combination that could be implemented efficiently (that does not need special cases, when implemented) for 7 available random bits.

2.2 Reverse Shuffle

The idea behind the simplest version of Reverse Shuffle (RS) technique is the following: AES-128 state is used as a vector of 16 bytes, S-box is applied to all bytes of the state following forward or reversed order (depending on the value of 1 random bit). For example, if the value of the random bit is 0 we may go through the state from byte 0 to byte 15 and if the value of the random bit is 1 we can go through bytes in the reversed order (from 15 to 0).

Matrix-RS

RS might be extended by using the state of AES-128 as a $m \times n$ matrix instead of a vector (where $m \times n$ is the size of the original vector, 16 in our case), we are going to call this extension Matrix-RS (M-RS). We will specify the exact M-RS version by using the notation M-RS $m \times n$. Note that M-RS 1×16 gives us the original simple RS.

The idea behind M-RS 4×4 is the following: we can use RS on each row (of 4 bytes) as well as for all rows (start from row 0 or row 3 in the matrix). It allows us to use from 1 up to 5 random bits for shuffling. For example, if we have 4 random bits we can go through all rows in forward order (no randomness required), we can also go through all cells in each row in forward

or reversed order (different order for all rows, 4 bits of randomness), see example in Figure 3, also see Table 2.

Table 2 shows how M-RS 4 × 4 might be applied on AES-128 depending on the number of available random bits. This table uses following notations: *Rand* means that indexes are handled in forward or reversed order randomly, *Fixed* means that same fixed order is used to go through cells in a row (or rows in the matrix); *S* means that same random bits are used on several rows[1], *D* means that different random bits are used for all rows.

Since a 16 byte AES-128 state might be represented as a matrix in several different ways (matrix of different size), we may use it to our advantage while using more or less random bits for shuffling. If we want to use more than 5 random bits and generate more shuffles we can use M-RS 8 × 2 shuffle, it will allow us to use up to 9 random bits (1 bit per row and 1 bit for all rows) and generate 512 shuffles.

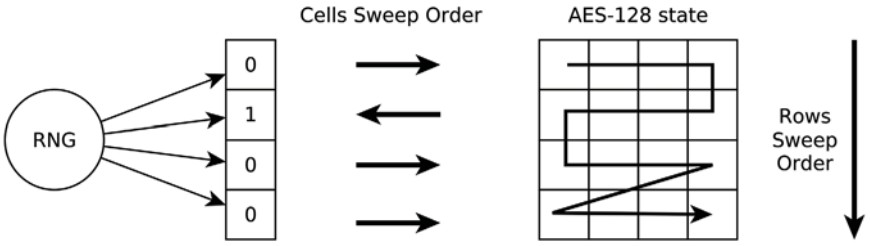

Figure 3 Example of M-RS 4 × 4 with 4 available random bits.

Table 2 Examples of M-RS use on 4 × 4 AES-128 state using different number on random bits

	All Rows		Cells in a Row		
Available Random Bits	Bits	Handling	Bits	Handling	Number of Shuffles
1	1	Rand	0	Fixed	2
2	1	Rand	1	Rand, S	4
3	1	Rand	2	Rand, 2S	8
4	0	Fixed	4	Rand, D	16
5	1	Rand	4	Rand, D	32

[1]2S in line 3 means that same bits are reused 2 times on 2 different rows and then different random bits are used on 2 other rows.

2.3 Sweep Swap Shuffle

The idea of Sweep Swap Shuffle (SSS) is based on the fact that the state of AES-128 might be represented as a $m \times n$ matrix (e.g., a 4×4 or a 2×8 matrix). A matrix might be handled row-by-row or column-by-column. SSS might also be implemented e.g., by swapping pieces of code that go through row and column indexes. In order to specify how a vector is represented as a matrix we will use the notation SSS $m \times n$. Figure 4 shows two possible orders of SSS 4×4.

Part SSS

The idea behind Part-SSS (P-SSS) extension of SSS technique is the following: a state of AES-128 might be broken into several equal parts e.g., 2 parts of 8 bytes. An SSS technique could be then applied to each part separately, it would allow us to create more different shuffles (by using more than 1 random bit). We will use the notation Pk-SSS $m \times n$ in order to specify the number k of identical parts that we want to use. Note that P1-SSS 4×4 gives us the original SSS 4×4. See example of P2-SSS 2×4 in Figure 5.

By using P-SSS on AES-128 we can generate up to 16 shuffles by using 1 to 4 random bits, see Table 3.

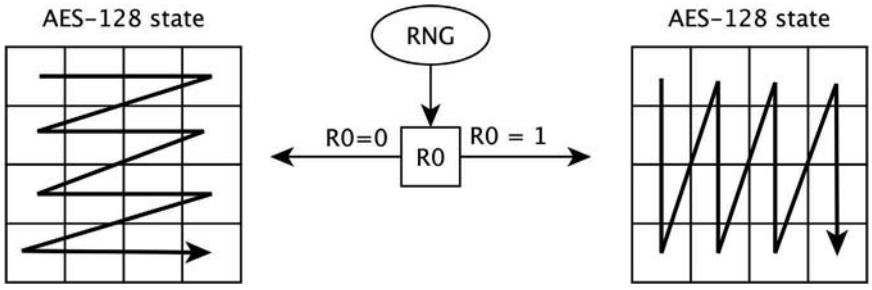

Figure 4 Going through bytes of AES-128 state matrix with SSS 4×4.

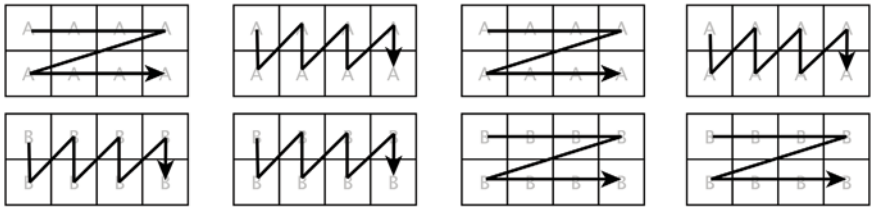

Figure 5 Possible shuffles of AES-128 state with P2-SSS 2×4 technique.

Table 3 Examples of P-SSS use on AES-128 state using different number on random bits

Random Bits (and k)	Technique	Shuffles
1	P1-SSS 4×4	2
2	P2-SSS 2×4	4
4	P4-SSS 2×2	16

Multidimensional SSS

The idea behind Multidimensional SSS (MD-SSS) extension of SSS technique is based on the fact that a vector might be seen as a multidimensional matrix[2]. For example the state of AES-128 might be seen as $2 \times 4 \times 2$ matrix, also see examples on Figure 6. It allows us to go through all dimensions in any order, e.g., in 2 dimensions the state might be handled row by row or column by column (go through the first dimension then through the second one or the other way around).

To specify a version of SSS we are going to use the notation MD-SSS $d_1 \times d_2 \times d_D$, where *D* is the number of dimensions and d_i is the size of the state in the dimension *i*. The number of shuffles that can be generated with MD-SSS depends on the number of dimensions that is used to represent

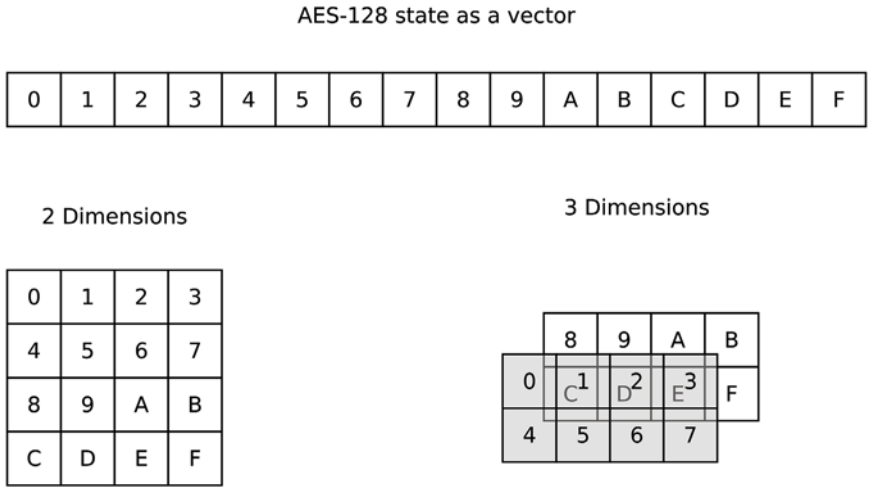

Figure 6 Examples of representations of AES-128 state as multidimensional matrices.

[2]It is important to note, that we can think about the state as if it was a three dimensional matrix for the purpose of shuffling, but it does not mean that the state has to be represented and manipulated as such during the entire algorithm.

Table 4 Examples of MD-SSS use on AES-128 state with different number on random bits

D	Random Bits	State Representation	Shuffles
2	1	2×8	2
3	3	$2 \times 4 \times 2$	6
4	5	$2 \times 2 \times 2 \times 2$	24

the state for the shuffling. Since we can choose any ordering of dimensions to handle the state, the number of different shuffles that might be generated is given by $D!$ and thus the number of necessary random bits is given by $\lceil log_2 D! \rceil$. Table 4 gives several examples of MD-SSS used with AES-128 state using different number of available random bits.

3 Analysis

In order to analyse our shuffling algorithms as well as to compare them to the existing schemes we introduce a couple of new terms and definitions.

3.1 Randomization

Randomization range of a shuffling technique is the biggest interval where the shuffling algorithm operates and where the shuffled operations might be reordered.

A randomization interval of a shuffling technique might be one operation (e.g., AddRoundKey), one round (or several operations of one round), several rounds or the entire algorithm. If the same shuffling technique is applied on SubBytes operation of all rounds of AES, then the randomization interval of this technique is still one operation (SubBytes) since instructions in between different SubBytes are not reordered among them.

The randomization range of all our shuffling techniques is one operation (SubBytes, as presented in Section 2). RP also has a randomization range of one operation. SchedAES has a very wide randomization range and it allows to generate many different shuffles but requires a huge amount of randomness.

Fully randomized instruction (or operation) is an instruction that might be reordered (and executed) at any instant in time by a given shuffling technique inside of its randomization range without changing the final result of the algorithm that is being shuffled.

A partially randomized instruction (or operation) is an instruction that is not fully randomized, but that might be reordered and executed at least 2 different instants in time by a given shuffling technique inside of its randomization range without changing the final result of the algorithm that is being shuffled.

An unrandomized instruction (or operation) is an instruction that is always executed at the same moment in time inside of the randomization range using a shuffling technique.

A shuffling algorithm is fully randomized if all instructions inside of its randomization range are fully randomized. If at least one instruction is unrandomized or only partially randomized, than the shuffling technique is partially randomized.

RSI applied to SubBytes is fully randomized in its basic version, but if we use less random bits (as in V-RSI) it becomes only partially randomized. Different versions of M-RSI might be fully randomized or partially randomized depending on choices made during the implementation (different number of random bits used to choose the start index for rows and inside of each row).

RS and its extensions are always partially randomized and it does not have unrandomized instructions if used with AES.

Unfortunately SSS is partially randomized and have unrandomized instructions since some bytes are always used at the same moment in time, indeed the S-box is always applied on the first byte at the beginning and on the last byte at the end of the SubBytes operation. Moreover, if we use e.g., SSS 4×4 on AES-128 S-box on bytes 0, 5, 10 and 15 are unrandomized since these bytes are situated on the diagonal of the 4×4 matrix. In general, handling a square matrix row by row or column by column does not change the position of elements on the diagonal. Thus, SSS $k \times k$ will have k unrandomized instructions.

In order to analyse all of the proposed shuffling schemes, we executed each one of them through the entire range of possible random inputs that each algorithm could receive as a parameter. For every algorithm we generated a heatmap of all possible positions in time when a SubBytes operation can take place on every single byte id. See examples of such plots on Figure 7, other figures are available in the Appendix 9. We can see that for each scheme available positions for each byte are uniformly randomly distributed, with the exception of 4 bytes of SSS (the bytes that are situated on the diagonal of the matrix). The exact patterns that we can observe on there heatmaps depend on the way the scheme was implemented (i.e. which bits were chosen to be fixed and which are random, recall Figures 1 and 2).

We also generated same type of heatmap for the RP shuffling scheme, see Figure 8. We used the implementation of RP shuffling scheme from DPA Contest v 4.2 [17]. It is currently impossible to enumerate all possible inputs (randomness) required by this shuffling scheme in a reasonable amount of time, so the heatmap from Figure 8 is generated using 2^{35} permutations. Using this approximation we can see that the ratio between the highest number of

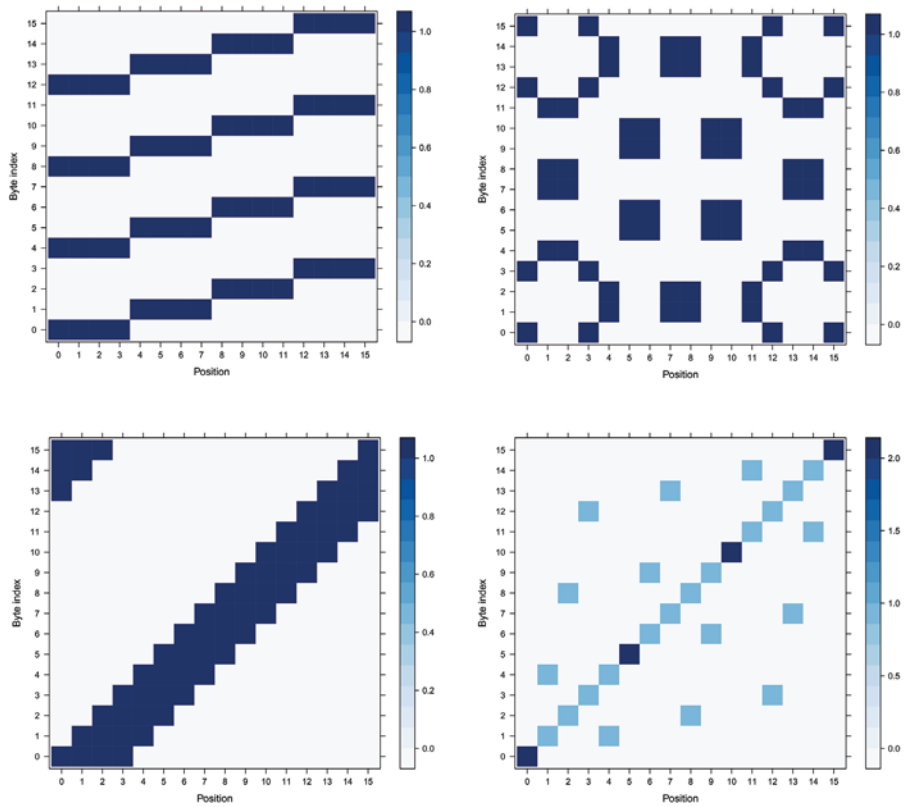

Figure 7 Examples of heatmaps of positions when the SubBytes operation takes place for every byte.

occurrences of a byte at a given position to the lowest number is equal to 1.000116, which is less than 0.01 % of difference (approximately 2^{-13}).

3.2 Number of Shuffles

Optimal shuffling algorithm is an algorithm that is able to generate 2^n different shuffles using n random bits.

If we have n random bits of information we will be able to generate at most 2^n different values. If a shuffling algorithm uses n bits of randomness and generates less than 2^n different shuffles, then it is not an optimal shuffling algorithm (from the point of view of information theory).

RS, RSI and all of their extensions use n bits in order to generate 2^n shuffles, see Tables 1 and 2, thus these schemes are optimal, however it is

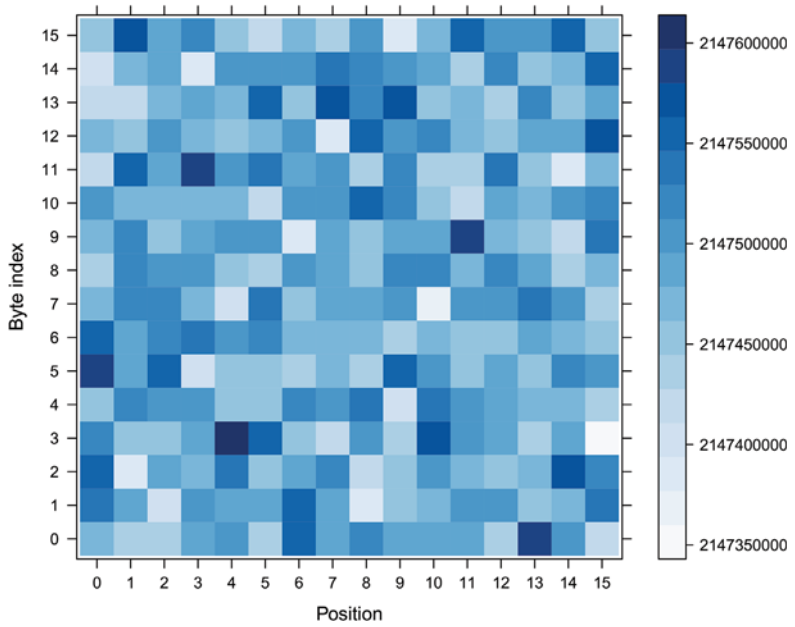

Figure 8 Heatmap of Random Permutation (RP) shuffling scheme. Implementation from DPA Contest v4.2 [1].

not always the case of SSS. The simple version of SSS is optimal as well as P-SSS, but not the MD-SSS since for a MD-SSS we can obtain $D!$ shuffles (where D is the number of dimensions) and $\forall \cdot a, b \in \mathbb{N}, (a > 2) \rightarrow a! \neq 2^b$.

RP is able to generate all $k!$ possible permutations of the state (k is the number of bytes that have to be shuffled), but it is not optimal since it requires more than $log_2(k!)$ random bits, the implementation proposed by Veyrat-Charvillon *et al.* [23] requires 64 bits of randomness.

Doubling the number of instants when an operation could be executed increases the amount of traces required for a successful attack roughly by a factor of four [12]. Thus, in a perfect scenario, a shuffling algorithm that generates more different permutations offers more security (because there should be more possibilities of different operations being performed at a given moment in time), however it is not always true. It is very important to notice, that some particular cases of RSI and RS extensions do not always improve the strength of the countermeasure when more random bits are used. For example, in the simplest version of RS it can generate only two permutations (forward and reversed), thus we know that we have only two possible indexes at each moment in time. If one would use 4×4 M-RS with 4 random bits

as suggested in Table 2 when rows are always handled in forward order while each row might be handled in the forward or the reversed order, we still have only two possible indexes that might be used at each moment in time. Same reasoning applies in some other cases, thus not all versions of each scheme give a security increase when more random bits are used, for more details see Table 5.

Table 5 Min and max number of different SubBytes operations that might occur at a fixed moment in time i.e. the number of different bytes of the state that might be handled at a given moment in time during shuffling. Results for different techniques are given according to the examples given in the paper

		Operations		
Technique	Random Bits	Min	Max	Total Shuffles
RP [23]	64	16	16	16!
RP•	45	16	16	16!
RSI	4	16	16	16
V-RSI	1	2	2	2
	2	4	4	4
	3	8	8	8
	4	16	16	16
M-RSI 4×4	1	2	2	2
	2	4	4	4
	3	8	8	8
	4	2	2	16
	4*	16	16	16
	5	4	4	32
	6	8	8	64
	8	4	4	256
	9	8	8	512
	10	16	16	1024
RS	1	2	2	2
M-RS 4×4	1	2	2	2
	2	4	4	4
	3	4	4	8
	4	2	2	16
	5	4	4	32
P1-SSS 4×4	1	1	2	2
P2-SSS 2×4	2	1	2	4
P4-SSS 2×2	4	1	2	16
MD-SSS	1	1	2	2
	3	1	3	6
	5	1	4	24

Result for RP• represents the theoretical lower bound on the number of necessary random bits, $\lceil log_2 16! \rceil = 45$.

Nevertheless, when we increase the number of random bits in a scheme we always increase the number of different shuffles that could be generated. Thus, from this perspective, the security of a scheme increases i.e., when an attacker learns the position of one byte it gives him less information about positions of all other bytes.

3.3 Resources

In addition to randomness, shuffling algorithms also require a certain additional memory and time. In order to support RP one needs to use an additional data structure (of the same size as the internal state of the algorithm, so its memory overhead is $O(k)$, where k is the size of the state). Depending on the algorithm RP might also require additional time overhead $O(k)$ up to $O(k^3)$ [1]. Our extensions of RS, RSI and SSS do not require as much memory, their memory overhead is limited to a couple of variables (generally to recompute and hold new indexes), in other words their memory overhead is $O(1)$. The only exception might be MD-SSS, where we need to store a small table of the size equal to the number of dimensions, which is always smaller than the size of the original internal state; in this case the memory overhead is $O(log(k))$.

Shuffling as a countermeasure also results in a time overhead. The exact time overhead might vary depending on the implementation and on the available hardware. We did several experiments on a `ATMega 328P` 8-bit microcontroller, all our code was written in `C++`. The microcontroller used an external 16 MHz clock. We implemented some of the variations of shuffling techniques that were described in Section 2. We applied several techniques on the SubBytes operations of the first and the last round of AES-128. The only detail that changed between different implementations were the two calls to functions that implemented different versions of SubBytes. In order to measure the time we encrypted 10 000 random plaintexts with different random bits as inputs to our shuffling techniques. Table 6 presents our results including and excluding the time needed for the generation of random bits (for shuffling). The first and the last rounds used same random bits for shuffling. We can see that most of the overhead comes from the RNG. The source code is available on our website[3]. All calculations (of indexes for memory accesses during shuffling) did not use conditional branching in any way dependent on the random bits used for the shuffling in order to prevent possible SPA and Timing attacks.

[3] http://www.ulb.ac.be/di/dpalab/download.html

Table 6 Execution time of 10 000 executions AES-128 encryptions with different shuffling techniques applied to the SubBytes operation of the first and the last rounds. Columns Including RNG and Excluding RNG give information including and excluding time for the generation of random numbers required for shuffling. Time is given in milliseconds, overhead is in %

		Including RNG		Excluding RNG	
Algorithm	RND Bits	Time	Overhead	Time	Overhead
No shuffling	0	13197	0.0	13197	0.0
RS	1	14500	9.9	13547	2.7
M-RS 4×4	1	14201	7.6	13246	0.3
	2	14438	9.4	13486	2.1
	3	14480	9.7	13528	2.5
	4	14362	8.8	13412	1.6
	5	14465	9.6	13514	2.4
SSS 4×4	1	14394	9.0	13441	1.8
V-RSI	1	14426	9.3	13473	2.1
	2	14426	9.3	13473	2.1
	3	14424	9.3	13473	2.1
	4	14423	9.3	13472	2.1
M-RSI 4×4	1	14186	7.5	13232	0.3
	2	14185	7.5	13232	0.3
	3	14322	8.5	13369	1.3
	4	14323	8.5	13372	1.3
	5	14425	9.3	13474	1.3
	6	14423	9.3	13475	1.3
	8	14319	8.5	13373	1.3
	9	14397	9.1	13452	1.9
	10	14398	9.1	13452	1.9

We can see that the overhead is relatively small and reasonable, but not negligible. Different techniques give slightly different time overheads which give the ability to choose depending on timing constraints that are imposed to a system.

These results might probably be improved by implementing other variants of our shuffling techniques or by implementing them in the assembly code (while taking into account the architecture of the microcontroller).

3.4 Resistance against Side-Channel Attacks

We analyse 3 different scenarios which represent 3 attackers of different strength in order to show the differences between presented shuffling schemes.

Unprofiled attack

Correlation Power Analysis (CPA) [3, 6] is considered to be among the strongest non-profiled Differential Power Analysis (DPA) attacks [8]. We have tested several versions of our shuffling schemes against CPA attack. The CPA was conducted using the Hamming Weight (HW) leakage model on simulated power traces. We have implemented the following algorithm:

$$r = Sbox(k \oplus m)$$

where r is the resulting 4×4 state, k is a 16 bytes fixed key and m is a 16 bytes message. The application of $Sbox$ function was shuffled using different shuffling schemes. In order to simulate power traces we used SILK [22] simulator with the following parameters: Hamming weight as the leakage function and the noise variance was set to 2. Same parameters were used for all simulations with different shuffling schemes.

Figure 9 shows the estimated number of traces that is necessary in order to extract the key with various shuffling techniques used as countermeasures. As expected, the number of traces increases with the number of different bytes that might be handled at a fix moment in time (due to shuffling). The success rate of this attack on each scheme is shown in Figure 12 (Appendix 6).

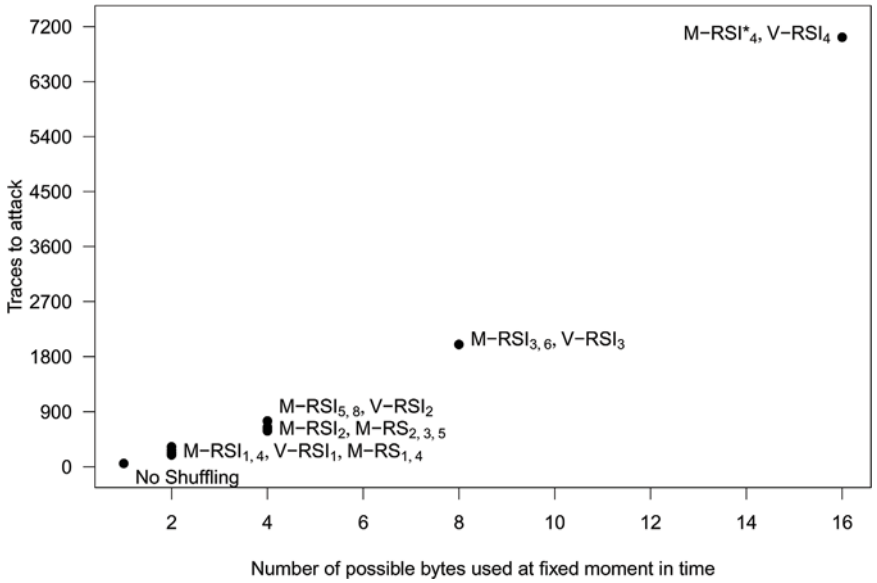

Figure 9 Number of traces needed to extract the key using CPA on different implementations.

The same CPA was applied to all shuffling techniques. among the presented algorithms, there are several techniques that give same advantage against a classic versions of DPA-like attacks, but some of them may generate more different permutations in total (see Table 5) and should make implementation specific attacks more difficult.

Implementation specific unprofiled attack

We used same simulation parameters and same algorithms and applied a CPA attack in a scenario when the attacker is aware of the shuffling scheme and knows the details of its implementation. We applied a preprocessing technique called integration to the power traces before applying the correlation power attack. In this scenario we sum all points which could be related to the attacked byte according to the shuffling scheme that is used, see Figure 7 and Figures of the Appendix 9. Thus, for a scheme where a byte can be handled in d different positions we integrate d points, for example we integrated 4 points while attacking M-RSI 4×4 with 2 random bits, those points correspond to positions where byte 0 can potentially be handled (i.e., points that correspond to bytes 0, 1, 2 and 3 in a scenario without shuffling, see Figure 7(a)). To sum up, we suppose that the attacker knows exactly where a given byte can be handled in a power traces.

Figure 10 shows the number of traces needed to successfully extract the secret (lowest number of traces where the success rate of the attack equals one). More details on some attacks are in Figure 13 (Appendix 7). We can see, that this technique is more powerful than a "simple" CPA against all shuffling schemes. Nevertheless, our results show that the more bytes can be found in a particular position (same moment in time during the execution of an algorithm), the more difficult this attack becomes.

Profiled attack

We used a Gaussian Template Attack (TA) [5] in a scenario when an attacker has profiling capabilities and when he has the knowledge of the shuffling scheme (i.e., he knows all points in time when a byte can be handled in the same way as in the unprofiled CPA with integration). However, in this scenario the attacker does not control the randomness during the profiling, which corresponds to a real case scenario (the randomness for cryptographic operations is generated inside of the device), thus an attacker does not know the full permutation (order of bytes during a single execution)[4].

[4]Even in a case when an attacker knows all random values, he might choose not to use them in order to speed up the profiling phase of the attack by building less templates.

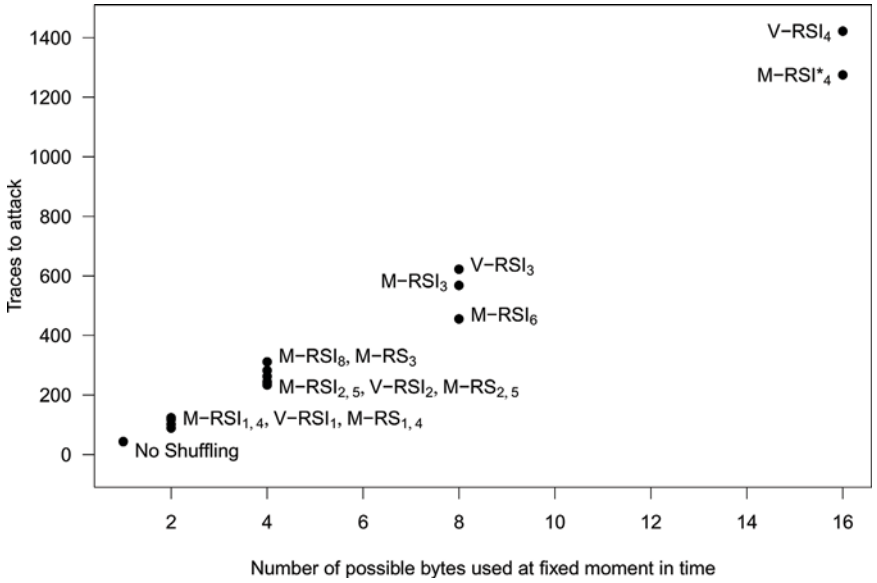

Figure 10 Number of traces needed to extract the key using CPA with integration as a preprocessing technique.

For each target value (attacked byte) the template corresponds to an average and a covariance matrix. The complexity of the parameters' estimation for each of these templates depends on the number of points that have to be considered during an attack. The number of points in each attack depends on the number of points in time when a byte might be handled and thus the number of points depends on the shuffling scheme. We used 5 000 traces per target value in order to build all profiles.

The number of traces needed to extract the key with high probability is shown in Figure 11 (the success rate of each attack can be found in Figure 14 in Appendix 8). These results shows us that the success of an attack depends on the number of possible bytes that could be handled at the same moment in time (due to shuffling), which is also the case for two other attacks. We can also note that the TA is better than the CPA with integration when the number of points used for the TA is low (i.e., when the number of positions where an given byte can be handled by the shuffling scheme is low or in other words when the number of possible bytes used at a fixed moment in time is low) e.g., see Figures 13 and 14 for 2 possible positions. However, the TA is less effective than the CPA with integration of points for schemes that result in permutations where a byte could be in many different positions (many points

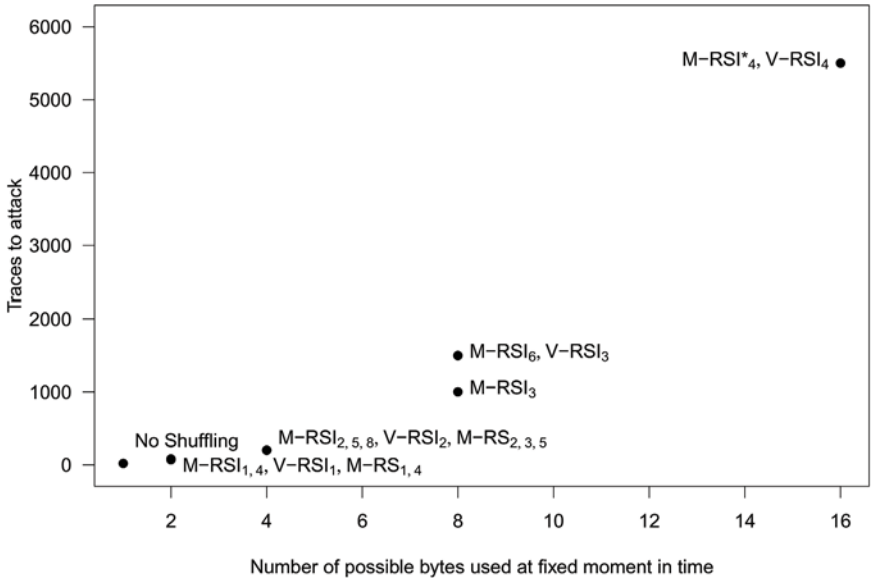

Figure 11 Number of traces needed to extract the key using TA on different implementations.

to consider in the TA), compare Figures 13 and 14 for 16 possible bytes used at a fixed moment. The advantage of the TA compared to the CPA with integration decreases when more points have to be taken into account due to the fact the TA suffers from the estimation error in high dimensionality contexts.

Targeting the RNG

Another implementation specific technique that an attacker might be used in order to attack these schemes could also be implemented. An attacker that knows the exact implementation of the shuffling countermeasure that was used might try to recover random bits used to shuffle the bytes and then extract the key using this knowledge (by finding the positions of shuffled operations using known random numbers). This technique was used to attack a masking scheme of a DPA Contest [10, 11]. Basically, the attacker targets the random number generator which allows to effectively remove the security mechanism that uses randomness. Thus, all masking and shuffling schemes are vulnerable to attacks that can successfully target the random number generator.

Attacks on other shuffling schemes

Our results with all three attacks suggest that the difficulty of attacking a given shuffling scheme mostly comes with the number of positions where a given

byte can be handled during an execution of the cryptographic algorithm. To be more precise, all schemes that can put a given byte at d positions require the same number of traces to extract the key for a given attack, see Figures 9, 10 and 11 where all points of the same column overlap or lie close to each other. Moreover, we can observe that the success rate of each attack on all schemes that result in putting a given byte to the same number of positions also closely follow each other, see Figures 12, 13 and 14.

Using these observations, we conclude that a given attack on any scheme S will give the same performance that this same attack on a scheme S' that can shuffle a byte into the same number of positions as the scheme S. Thus, an attack on the first byte of the SSS is identical to attacking an unprotected implementation, while an attack on the second byte will perform as an attack on a byte of e.g., V-RSI with 1 random bit (see Table 5 and Figures 7(d) and 15(a)). Same resoning can be applied to P-SSS and MD-SSS schemes. Thus, the RP scheme is as difficult to attack as M-RSI 4×4 with 10 bits or V-RSI with 4 random bits (see Table 5).

Nevertheless, it is important to note, that this reasoning holds if the RNG is not biased and if the implementation under attack does not have additional flaws that an attacker can exploit nor additional sources of information available to the attacker. This result can also vary in case if additional countermeasures are applied with a shuffling scheme.

3.5 Applications & Modifications

It is easy to apply RSI, RS, SSS and their extensions to SubBytes or AddRoundKey operations of AES-128 since each of them operates only on one byte of the state at a time and does not have any precedence requirements inside of the operation. In order to apply these techniques to ShiftRows or MixColumns operations we may simply consider a row or a column as a memory unit (instead of a byte).

Same techniques might be adapted for 192-bit and 256-bit versions as well as for other operations of AES cipher by using more random bits to handle additional rows. RSI, RS, SSS and their extensions might be also applied to other algorithms. These techniques should be applicable if parts of the state are used independently from each other during some computations e.g., the application of the S-box in ciphers like Blowfish [19], DES [16] or PRESENT [2].

We can also combine RS, RSI, SSS and their extensions in order to obtain more different shuffles, e.g., RS might be used with 10-bit version

of M-RSI on the AES-128 in order to get 2048 different shuffles by using 11 random bits.

Finally, it is worth noting that not all techniques presented here (as well as their extensions) are equally practical and are equally secure (with a given number of random bits). Nevertheless, we considered that all versions with their extensions should be presented for the sake of completeness of this work. For example, SSS is not as practical as RSI extensions for security, optimality as well as penalty reasons; however, we think that SSS is a nice case study for theory.

4 Discussion

All of the shuffling schemes that we propose and describe are similar i.e., each one suggests a way of going through all indexes of the state in a particular order that could be easily implemented with a small overhead. Most of these techniques could be seen as extensions and generalizations of the random starting index shuffling scheme.

Our scalable shuffling schemes can offer different number of shuffles and different number of positions (moments in time) where a particular byte is handled, overall it results in different levels of security. In order to choose which shuffling scheme to implement we advise the designers of a cryptosystem to consider the following criteria in given order:

- Number of available random bits
- Timing constraints
- Number of different operations that could be handled at a given moment in time
- Number of shuffles

The first criterion is related to the basic constraints of the system, so the designer should use as much randomness as he can in order to increase the security. The second criterion changes depending on the given hardware and implementation, so it has to be tested for each platform individually, however, our results show that all shuffling schemes that we present result in very similar timing overhead. Results of all our attacks suggest that a scheme which generates shuffles such that higher number of different bytes that could potentially be handled at a fixed moment in time (from the beginning of the execution) increases the difficulty of an attack. Thus a designer of a cryptosystem should choose a shuffling algorithm that maximizes this number. Finally, the number of different shuffles that a given shuffling scheme can

produce does not influence the number of traces that is needed in order to mount a successful attack. However, mounting some profiled attacks becomes increasingly difficult when the number of shuffles grows since an attacker would have to create a model per shuffle [4]. This parameter can also help in case if the attacker can find out a position of one byte in order to reduce the remaining uncertainty on the positions of other bytes.

Our results based on side-channel analysis using 3 different attackers show that the number of different operations that might occur at a given moment in time produces the biggest effect on the success rate of an attack. This result hold even for attacks that take into account the implemented scheme. From this perspective, the SSS scheme presents a disadvantage because it does not shuffle all bytes, some of the bytes always remain at a fixed position in a trace. However, SSS could still be interesting in practice, because it could be implemented using only a couple of additional instructions on many hardware platforms (without taking into account the random number generator) e.g., conditional swap instruction (MSWAPF) available on TMS320x2803x or using compare-and-exchange (CMPXCHG) that is available on many Intel and AMD processors. SSS could also be easily combined with other shuffling schemes thus giving a boost to the security of the system.

A specific type of attack could also be mounted against all of the presented shuffling schemes. If an attacker targets the random number generator in order to find out the ordering that is generated by the shuffling scheme, she can effectively remove the protection given by the countermeasure. This type of attack could be mounted on any shuffling of masking countermeasures [10]. Thus, algorithmic countermeasures that rely on randomness require a secure random number generator that could not be easily targeted through side-channel analysis.

5 Conclusions and Future Works

Often it is important to be able to choose among several different countermeasures, since some of them might be implemented more efficiently on a given platform (e.g., the hardware might have special instructions available), that is why it is quite important to have an entire set of different countermeasures that might offer similar performances. This is especially the case when the developed system has a lot of strong constraints, which is the case in mobile applications such as used in IoT.

We presented a couple of new scalable shuffling techniques (RS and SSS) and a wide variety of their extensions as well as several different extensions of an existing shuffling scheme (RSI). The main advantage of our proposals is the

fact that they allow developers to fine-tune the countermeasure to their needs. It is possible to adjust parameters of our shuffling techniques depending on the requirements and constraints of the cryptosystem such as time (or throughput), code (or hardware) size and amount of available random bits per unit of time. In other words, our proposals are not as rigid as other existing shuffling schemes.

We have compared RSI, RS and SSS extensions between them as well as with couple of other shuffling techniques such as RP from the several points of view: data overhead, required amount of random bits and number of different shuffles (orderings) that might be generated as well as their relative strength against CPA attack on a simulator. We also implemented AES-128 block cipher using 21 of the proposed extensions of RSI, RS and SSS on an 8-bit microcontroller in order to compare their time overhead over an unprotected implementation. In our implementations shuffling is applied on SubBytes operations of the first and the last rounds of AES-128, but it could easily be extended to more operations, other versions of AES as well as other ciphers. Presented techniques offer different levels of security given a fixed number of random bits, however some of them are easier to implement than others depending on the target hardware platform, so even less secure techniques might be useful.

Our results show that the main parameter that influences the success rate of an attack against a shuffling technique is the number of different bytes that might be handled at a fixed moment in time, but not the number of shuffles (during an attack on one byte). However, the number of shuffles increases the difficulty of mounting a profiled attack and it also increases the difficulty of attacking the entire key (position of one byte gives less information on positions of the others), thus it is also an important parameter.

This techniques should not be used as a stand alone countermeasure. It should be combined with masking techniques e.g., as suggested for the DPA Contest v4 [1], especially since some studies show that masking techniques are much more efficient in presence of noise [20], additional noise could be provided by shuffling.

As a future work it would be interesting to implement same countermeasures on different hardware platforms in order to analyze whether some of them might be better adapted to particular platforms. It would also be interesting to explore different combinations of RSI, RS, SSS and their extensions. Some of them might not be practical, others might be efficient and generate more permutations. Finally, it would be nice to further extend these shuffling techniques in order to allow them to shuffle several operations of the same round at once, like SchedAES [13].

Appendix

6 Simple CPA

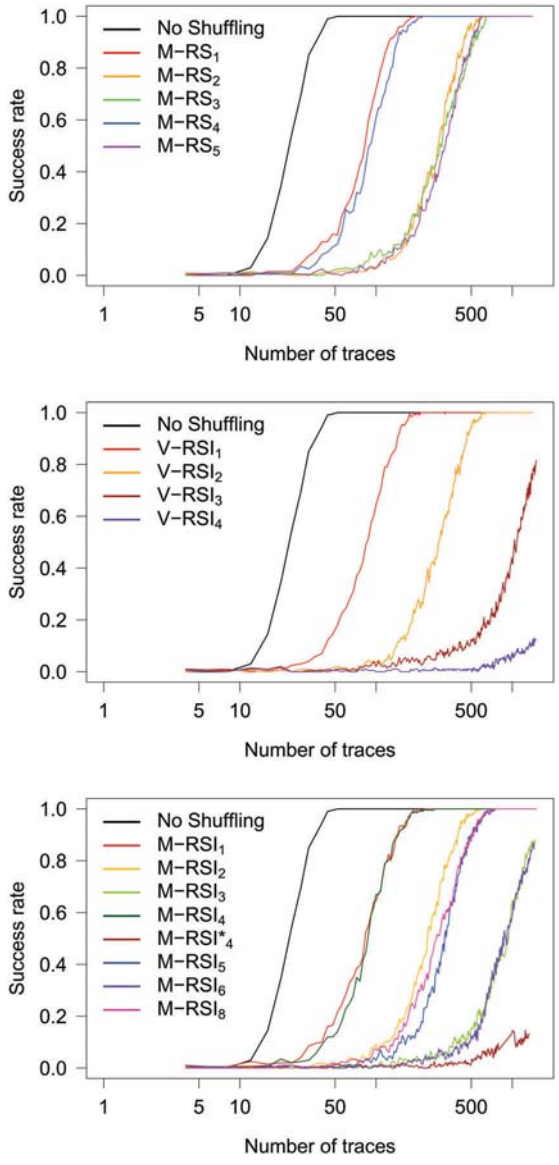

Figure 12 The success rate of an non-profiled correlation power analysis against different shuffling techniques. Horizontal axis is logarithmic.

7 CPA with Integration

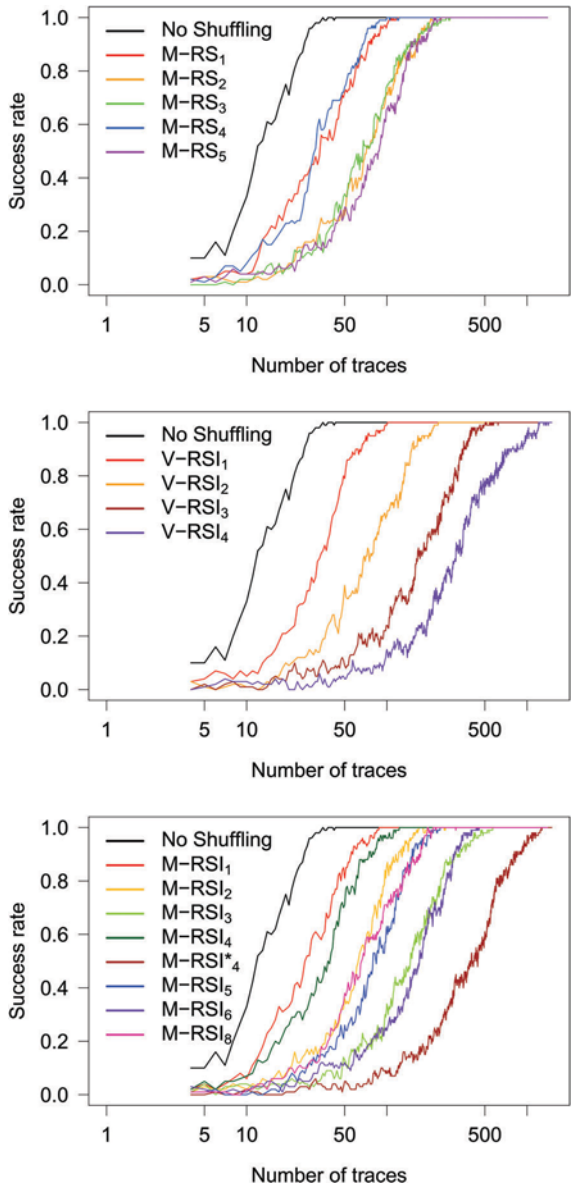

Figure 13 The success rate of an non-profiled correlation power analysis with integration (preprocessing) against different shuffling techniques. Horizontal axis is logarithmic.

8 Template Attack

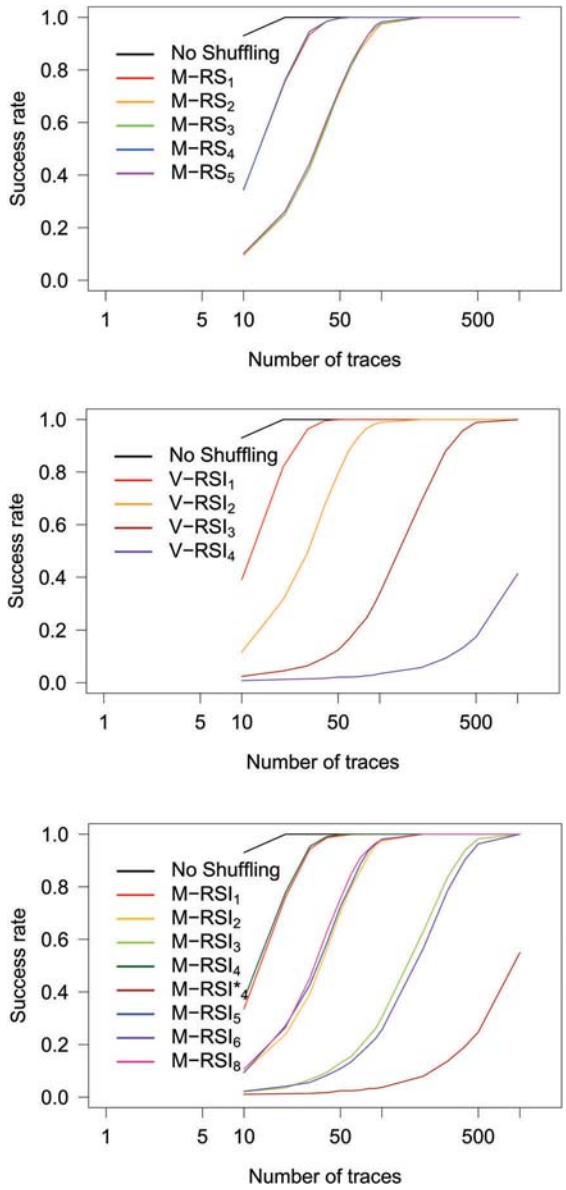

Figure 14 The success rate of a (profiled) template attack against different shuffling techniques. Horizontal axis is logarithmic.

9 Heatmaps of Byte Positions Distributions

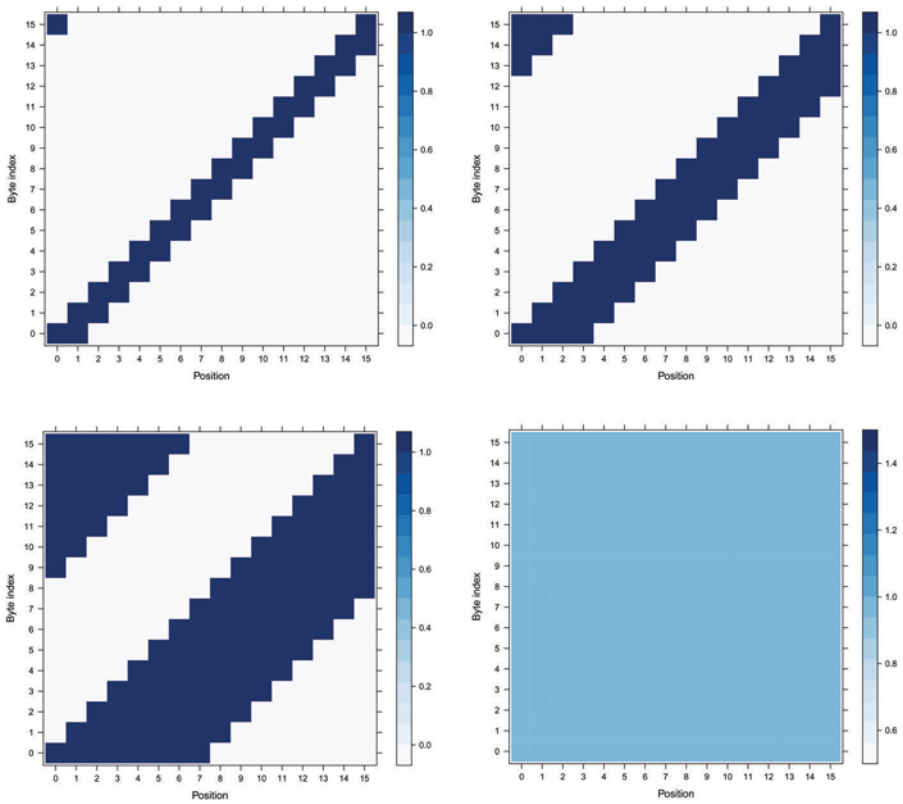

Figure 15 V-RSI Heatmap of positions when the SubBytes operation takes place for every byte.

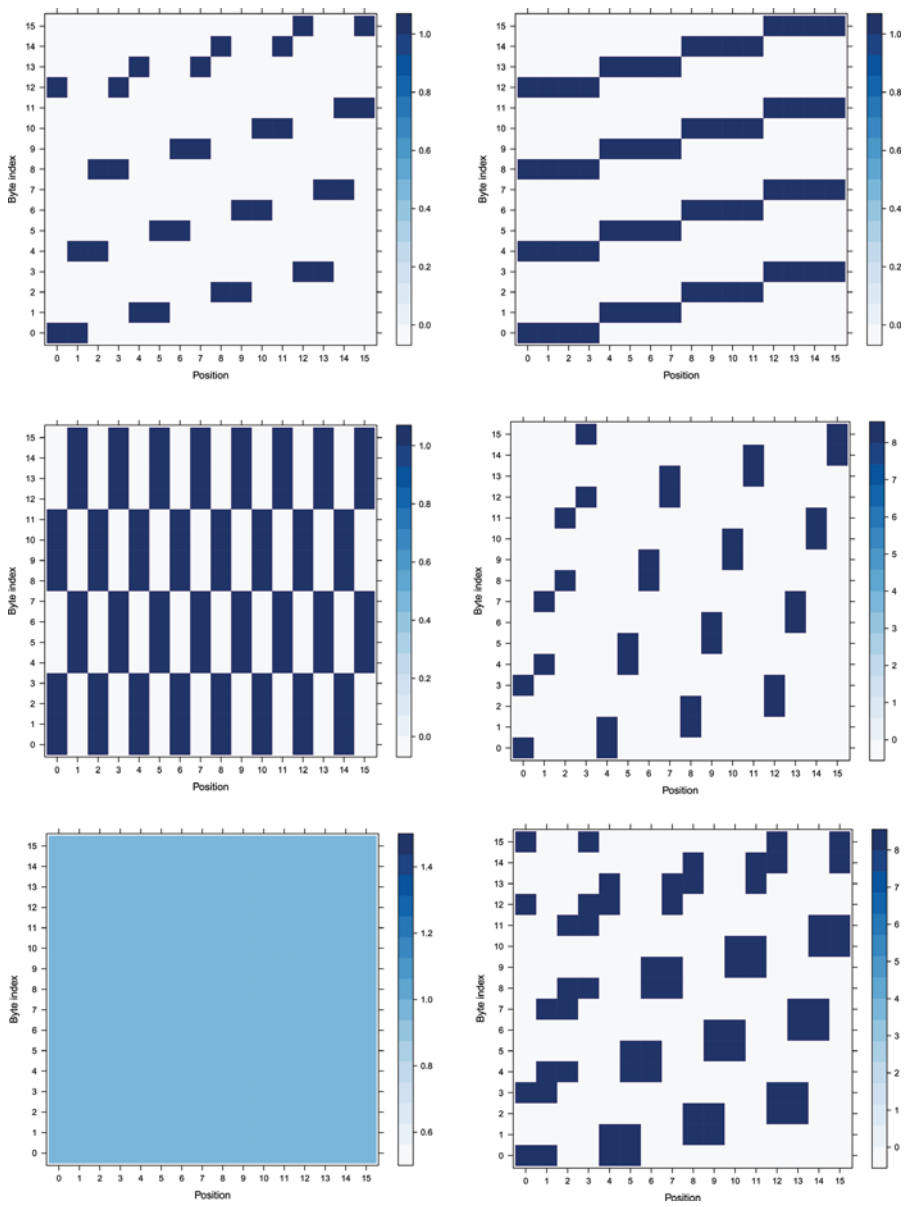

Figure 16 M-RSI 4×4 Heatmaps of positions when the SubBytes operation takes place for every byte.

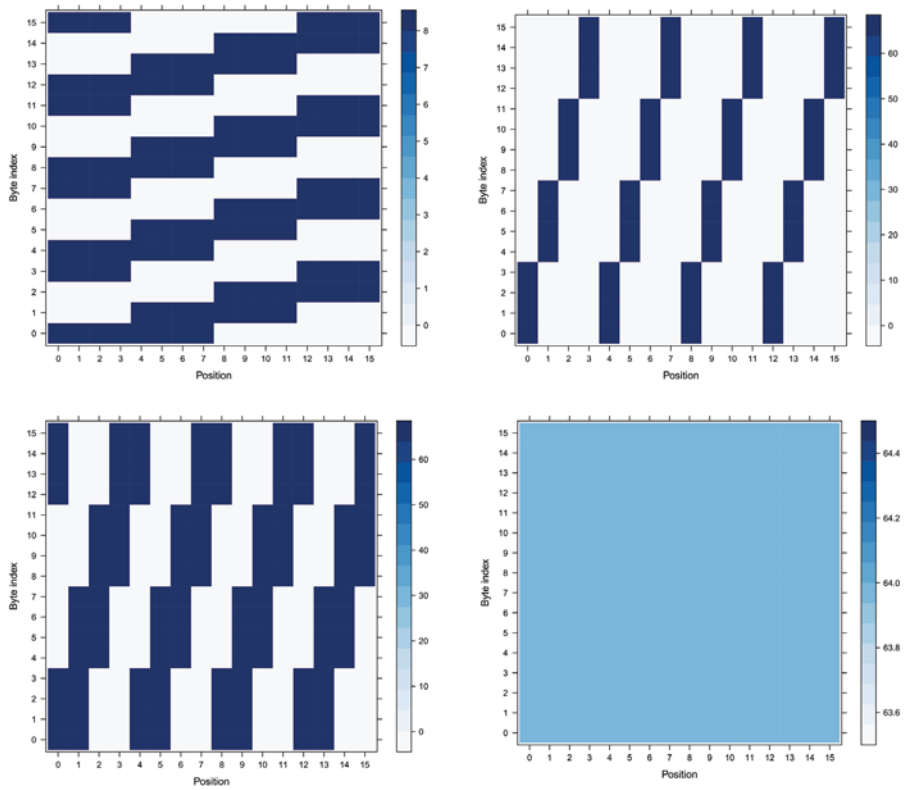

Figure 17 M-RSI 4 $\times$ 4 Heatmaps part 2 of positions when the SubBytes operation takes place for every byte.

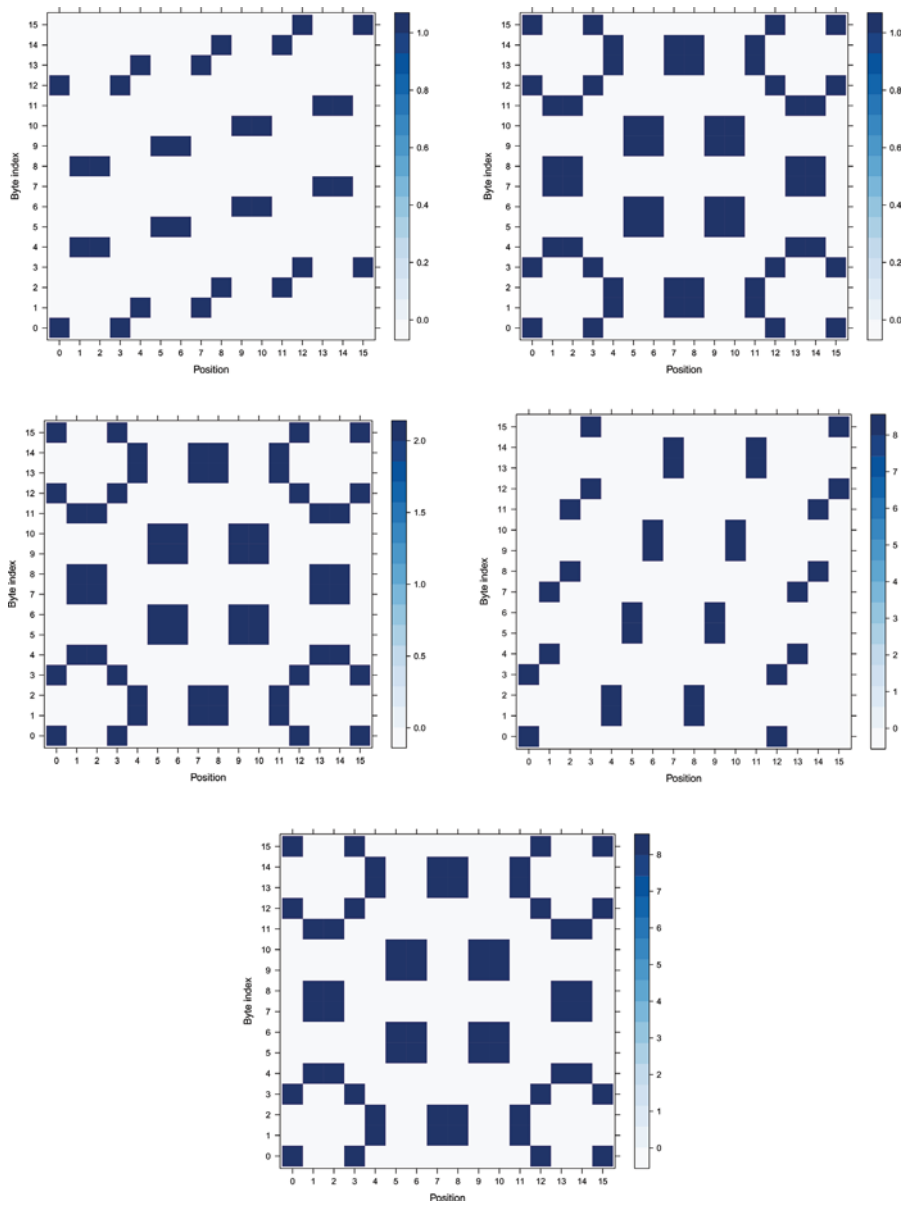

Figure 18 M-RS 4×4 Heatmaps of positions when the SubBytes operation takes place for every byte.

Acknowledgements

The research of L. Lerman is funded by the Brussels Institute for Research and Innovation (Innoviris) for the SCAUT project. The research of S. Fernandes Medeiros is funded by the Région Wallone.

References

[1] Bhasin, S., Bruneau, N., Danger, J.-L., Guilley, S., and Najm, Z. (2014). "Analysis and improvements of the dpa contest v4 implementation," in *Security, Privacy, and Applied Cryptography Engineering*, eds R. S. Chakraborty, V. Matyas, and P. Schaumont (Cham: Springer), 201–218.

[2] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., *et al.* (2007). "Present: an ultralightweight block cipher," in *Proceedings of the 9th International Workshop Cryptographic: Hardware and Embedded Systems-CHES, 2007*, eds P. Paillier and I. Verbauwhede (Berlin: Springer), 450–466.

[3] Brier, E., Clavier, C., and Olivier, F. (2004). "Correlation power analysis with a leakage model," in *Cryptographic Hardware and Embedded Systems-CHES 2004*, eds M. Joye, and J. J. Quisquater (Berlin: Springer), 16–29.

[4] Bruneau, N., Guilley, S., Heuser, A., Rioul, O., Standaert, F.-X., and Teglia, Y. (2016). "Taylor expansion of maximum likelihood attacks for masked and shuffled implementations," in *Proceedings of the Advances in Cryptology – ASIACRYPT 2016 – 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4–8, 2016, Proceedings, Part I, Lecture Notes in Computer Science*, Vol. 10031, eds J. H. Cheon and T. Takagi (Berlin: Springer), 573–601.

[5] Chari, S., Rao, J. R., and Rohatgi, P. (2002). "Template attacks," in eds B. S. Kaliski Jr., Çetin Kaya Koç, and C. Paar, Proceedings of the 4th International Workshop: Cryptographic Hardware and Embedded Systems – CHES 2002, Redwood Shores, CA, USA, August 13–15, 2002: Lecture Notes in Computer Science, Vol. 2523, (Berlin: Springer), 13–28.

[6] Coron, J.-S., Kocher, P., and Naccache, D. (2001). "Statistics and secret leakage," in *Financial Cryptography*, ed. Y. Frankel (Berlin: Springer), 157–173.

[7] Herbst, C., Oswald, E., and Mangard, S. (2006). "An AES smart card implementation resistant to power analysis attacks," in *Proceedings*

of the 4th International Conference, ACNS 2006: Applied Cryptography and Network Security, Singapore, June 6–9, 2006: Lecture Notes in Computer Science, Vol. 3989, eds J. Zhou, M. Yung, and F. Bao (Berlin: Springer), 239–252.

[8] Kocher, P., Jaffe, J., and Jun, B. (1999). "Differential power analysis," in *Proceedings of the Advances in Cryptology-CRYPTO'99*, (Berlin: Springer), 388–397.

[9] P. C. Kocher. (1996). "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," in *Proceedings of the CRYPTO: Lecture Notes in Computer Science*, Vol. 1109, ed. N. Koblitz (Berlin: Springer), 104–113.

[10] Lerman, L., Bontempi, G., and Markowitch, O. (2015). A machine learning approach against a masked AES – reaching the limit of side-channel attacks with a learning model. *J. Cryptogr. Eng.* 5, 123–139.

[11] L. Lerman, S. Fernandes Medeiros, G. Bontempi, and O. Markowitch. (). "A machine learning approach against a masked AES," in *Proceedings of the 12th International Conference, CARDIS 2013: Smart Card Research and Advanced Applications, Berlin, Germany, November 27–29, 2013. Revised Selected Papers, Lecture Notes in Computer Science, Vol. 8419*, eds A. Francillon and P. Rohatgi (Berlin: Springer), 61–75.

[12] Mangard, S., Oswald, E., and Popp, T. (2008). *Power Analysis Attacks: Revealing the Secrets of Smart Cards*, Vol. 31. Berlin: Springer Science & Business Media.

[13] Fernandes Medeiros, S.(2012). "The schedulability of aes as a countermeasure against side channel attacks," in *Proceedings of the SPACE: Lecture Notes in Computer Science*, Vol. 7644, eds A. Bogdanov and S. K. Sanadhya (Berlin: Springer), 16–31.

[14] Medwed, M., Standaert, F.-X., Großsch¨adl, J., and Regazzoni, F. (2010). "Fresh re-keying: security against side-channel and fault attacks for low-cost devices," in *Proceedings of the Progress in Cryptology–AFRICACRYPT 2010*, (Berlin: Springer), 279–296.

[15] Moradi, A., Mischke, O., and Paar, C. (2011). "Practical evaluation of dpa countermeasures on reconfigurable hardware," in *Proceedings of the 2011 IEEE International Symposium on Hardware-Oriented Security and Trust* (HOST), (Piscataway, NJ: IEEE), 154–160.

[16] NIST FIPS PUB. 46-3. (1977). *NIST FIPS PUB. 46-3 data encryption standard. Federal Information Processing Standards*. Gaithersburg, MD: National Institute of Standards and Technology.

[17] TELECOM ParisTech SEN Research Group (2013). *DPA Contest*. Availble at: http://www.dpacontest.org

[18] Rivain, M., Prouff, E., and Doget, J. (2009). "Higher-order masking and shuffling for software implementations of block ciphers," in C. Clavier and K. Gaj, *Proceedings of the 11th International Workshop, Lausanne, Switzerland, September 6–9, 2009: Cryptographic Hardware and Embedded Systems – CHES 2009: Lecture Notes in Computer Science*, Vol. 5747, (Berlin: Springer), 171–188.

[19] Schneier, B. (1994). "Description of a new variable-length key, 64-bit block cipher (blowfish)," in *Fast Software Encryption*, ed. R. Anderson (Berlin: Springer), 191–204.

[20] Standaert, F.-X., Veyrat-Charvillon, N., Oswald, E., Gierlichs, B., Medwed, M., Kasper, M., *et al.* (2010). "The world is not enough: Another look on second-order dpa," in *Proceedings of the Advances in Cryptology-ASIACRYPT 2010*, (Berlin: Springer), 112–129.

[21] Tillich, S., Herbst, C., and Mangard, S. (2007). "Protecting AES software implementations on 32-bit processors against power analysis," in *Proceedings of the 5th International Conference: Applied Cryptography and Network Security, ACNS 2007, Zhuhai, China, June 5–8, 2007: Lecture Notes in Computer Science*, Vol. 4521, eds J. Katz and M. Yung (Berlin: Springer), 141–157.

[22] Veshchikov, N. (2014). "Silk: High level of abstraction leakage simulator for side channel analysis," in *Proceedings of the 4th Program Protection and Reverse Engineering Workshop, PPREW-4*, (New York, NY: ACM), 3:1–3:11.

[23] Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., and Standaert, F.-X. (2012). "Shuffling against side-channel attacks: a comprehensive study with cautionary note" in *Proceedings of the Advances in Cryptology ASIACRYPT 2012: Lecture Notes in Computer Science*, Vol. 7658, eds X. Wang and K. Sako (Berlin: Springer), 740–757.

Biographies

Nikita Veshchikov got his Bachelor in Computer Sciences in 2009 at Université Libre de Bruxelles (ULB) in Belgium. He continued studies in the same field and got a Master in Computer Sciences with advanced studies of embedded systems in 2011 at the same university. During his master thesis he studied reverse engineering and anti-patching techniques. Since 2011 Nikita works as a teaching assistant while also working on his PhD thesis in the field of side-channel attacks. His is mostly interested in simulators and automated tools for side-channel analysis and computer assisted secure development. He is also interested in lightweight secure implementations.

Stephane Fernandes Medeiros got his Bachelor (in 2007) and his Master (in 2009) degree in computer sciences at the Université libre de Bruxelles (ULB), Belgium. He worked on his PhD in the domain of software countermeasures against side-channel attacks while being a teaching assistant at ULB, he got his PhD in 2015. Now Stephane works as a postdoctoral researcher at the Université libre de Bruxelles, he is mainly working on security protocols for small embedded devices.

Liran Lerman received the PhD degree in the department of Computer Science at the Université libre de Bruxelles (in Belgium) in 2015. In 2010, he received with honors (grade magna cum laude) the master degree from the same university. During his PhD thesis, he was a teaching assistant and a student doing research as part of a Machine Learning Group (MLG) and the Cryptography and Security Service (QualSec). Currently, he is a post-doctoral researcher of the QualSec. His research relates to machine learning, side-channel attacks and countermeasures.