Analysis and Improvements of the DPA Contest v4 Implementation

Shivam Bhasin[1], Nicolas Bruneau[1,2], Jean-Luc Danger[1,3], Sylvain Guilley[1,3], and Zakaria Najm[1]

[1] TELECOM-ParisTech, Crypto Group, Paris, France
[2] AST division, Rousset, France
[3] Secure-IC S.A.S., Rennes, France

Abstract. DPA Contest is an international framework which allows researchers to compare their attacks under a common setting. The latest version of DPA Contest proposes a software implementation of AES-256 protected with a low-entropy masking scheme. The masking scheme is called Rotating Sbox Masking (RSM) which claims first-degree security. In this paper, we review the attacks submitted against DPA Contest v4 implementation to identify the common loop holes in the proposed implementation. Next we propose some ideas to improve the existing implementation to resist most of the proposed attacks at affordable performance overhead. Finally we compare our implementation with the original proposal in terms of complexity and side-channel leakage.

Keywords: Side Channel Attacks, DPA Contest, Low Entropy Masking Schemes, Shuffling.

1 Introduction

Physical systems are now an integral part of our life. Such systems use embedded computers and sensors to perform desired computation based on feedback from physical processes and vice-versa. Some typical application of physical systems are in domains like health management, traffic management, data-centers, power-grids, etc. Given the critical nature of applications, it becomes an attractive target for all kinds of attacks. This brings in the need for security and privacy.

A common solution to security threats is to use cryptography. Modern cryptographic algorithms are based on strong mathematical problems and are considered secure from a theoretical view point. On the other hand, when these algorithms are implemented in a physical systems, they become vulnerable. These attacks which compromise the physical implementation of cryptography are known as physical attacks or "Side-Channel Attacks" (SCA [1,2]). In such cases, designers resort to countermeasures. Countermeasures for SCA tend to modify the implementation in a way that the mere basis of SCA is removed. Having said that, a perfect countermeasure is not possible to design. This is because certain non-linearities in the target device which are not under the control of designer leave the countermeasure imperfect. Therefore a common trend in SCA countermeasure research is to make the design harder to attack, given the design constraints. In this paper, we focus on symmetric ciphers that run as software codes on embedded computers.

Common countermeasures for software implementations of ciphers are masking and shuffling [3,4]. Another lesser studied countermeasure for software implementations is hiding [5]. All the countermeasures come at a significant cost overhead in terms of memory, time or both. Hiding based countermeasure makes the leakage uniform and independant of the data processed. Shuffling is a simple countermeasure which plays on randomizing the order of operations of the cipher. Masking on the other hand uses a random value called "mask" which is mixed with the sensitive data. The mixing is done by using different operations like XOR, addition, multiplication etc. Out of the three countermeasures for software implementations, masking is the most studied one.

Recently, researchers have started looking into the lightweight solutions for SCA countermeasures. These countermeasures are designed to resist not all but a selection of important and powerful attacks. One such countermeasure is Rotating Sbox Masking (RSM) which is a type of Low-Entropy Masking Scheme (LEMS). RSM was initially proposed for hardware implementations [6] and further tuned for software targets in [7]. We choose RSM because it has been studied widely by researchers worldwide under the framework of DPA Contest [8]. DPA Contest allows researchers to apply their attacks on a common set of available side-channel traces, in order to find the best attacks. During the fourth version of the contest i.e. DPA Contest v4 (DPACv4 [9]) the target was a AES-256 implementation protected with RSM running on an ATMEL AVR-163 microcontroller. Both the implementation and the traces were made available as a part of the framework.

In this paper, we review the attacks proposed in DPACv4 framework to identify the common pitfalls of the proposed implementation. Next we try to propose an improved implementation of RSM which does not suffer from some of the obvious and noted pitfalls. The rest of the paper is organised as follows: Sec. 2 provides general background on DPA contest and its latest version and RSM. In Sec. 3, we review the attacks proposed under the framework of DPACv4 with prime focus on non-profiled attacks to identify the main pitfalls in the implementations. A proposition to improve the original implementation of DPACv4 is given in Sec. 4 followed by security evaluation in Sec. 5. Finally conclusions are drawn in Sec. 6. Technical proofs are in appendix.

2 General Background

2.1 DPA Contest

DPA Contest is an international contest which allows researchers from all over the world to compete on a common ground. It was launched in 2008 and since then four versions of the contest have completed. The first version of the contest targeted an unprotected DES implementation running on a ASIC fabricated in ST 130 nm technology. Version 2 of the contest targeted a unprotected AES implementation running on an FPGA platform [25]. The database of traces of both implementations was made available online, with a goal to find the attack which recovers the secret key using minimum number of traces. The next version of the contest (v3) was an acquisition competition which focused on finding the best measurement setup. The latest or the fourth version of the contest was launched last year. This contest targets a protected AES-256 implementation on a 350 nm metal-3 layer ATMEL AVR-163 microcontroller. Protection applied is RSM which is a LEMS and discussed in next subsection.

2.2 Masking and RSM

Masking splits sensitive data $Z \in \mathbb{F}_2^n$ into $(d+1)$ variable random shares, noted $\boldsymbol{R} = (R_i)_{i \in [\![0,d]\!]}$, in such a way that the relation $R_0 \perp \cdots \perp R_d = Z$ is satisfied for a group operation $\perp$ [10].. Typically, $\perp = \oplus$, the exclusive-or (XOR) operation. Such schemes claim d^{th}-order security. When a cryptographic algorithm is modified to introduce masking, two computations are performed: masked sensitive and mask compensation computation. In software, this computation is performed in serial. The linear operations can be easily masked. Masking the non-linear sbox S involves computing $S(Z) \oplus M'$ from the variables M, $Z \oplus M$ and M' (new mask) without compromising with SCA resistance.

GLUT [11], a proposed solution, pre-computes a look-up table, associated to the function $S' : (X, Y, Y') \mapsto S(X \oplus Y) \oplus Y'$. This approach is very expensive in practice. Rotating Sbox Masking (RSM) is based on precomputed table look-ups at the same time reducing the area overhead of GLUT. The optimization comes from reusing sboxes and removal of computation of mask compensation. RSM is a LEMS but the low-entropy is covered for by carefully choosing the mask set M. From a security point of view, M is chosen such that the jth order moment of the conditional leakage $L^j | Z = z$ given a guess on the sensitive variable Z are all the same for $j = 1, 2, \cdots, d$. Thus only an attack of order $(d+1)$ can succeed. Under this constraint, the masks set M must be an orthogonal array of strength d [12].

The unmasking and masking which is integrated into the precomputed masked sbox removes the need for computation of corresponding mask compensation. The set of chosen mask M can be a public parameter however M should be shifted by a random offset before each encryption. The linear operations are masked by a simple XOR operation with precomputed constants applied at the end of each round. For a linear operation P, a mask m_i can be computed as $P(m_i) \oplus m_i$ on the fly or stored precomputed in memory. We refer interested readers to [6], [7] and [13] for details of RSM and its security analysis.

2.3 DPACv4 Implementation

DPACv4 targets an AES-256 implementation protected with RSM. It was mostly written in the `C` language, and compiled using `avr-gcc`. The overall algorithm running on the smartcard is described in Alg. 4 in Appendix A.

A quick glossary for Alg. 4 is as follows:

- $\mathsf{MaskedSubBytes}_i(X) = \mathsf{SubBytes}(X \oplus M_i) \oplus M_{i+1}$
- $\mathsf{MaskCompensation}_{\mathsf{offset}} = \mathsf{Mask}_{\mathsf{offset}} \oplus \mathsf{MixColumns}(\mathsf{ShiftRows}(\mathsf{Mask}_{\mathsf{offset}}))$
- $\mathsf{MaskCompensationLastRound}_{\mathsf{offset}} = \mathsf{Mask}_{\mathsf{offset}} \oplus \mathsf{ShiftRows}(\mathsf{Mask}_{\mathsf{offset}})$

The MaskedSubBytes operation firstly calls the 8 sboxes with even index followed by remaining 8 sboxes with odd index. The $\mathsf{Mask}_{\mathsf{offset}}$ operation applies 16 mask bytes to 16 state bytes according to the computed index. The mask set used for DPACv4 is:

$$M_{i\in[\![0,15]\!]} = \{\texttt{0x00}, \texttt{0x0f}, \texttt{0x36}, \texttt{0x39}, \texttt{0x53}, \texttt{0x5c}, \texttt{0x65}, \texttt{0x6a}, \\ \texttt{0x95}, \texttt{0x9a}, \texttt{0xa3}, \texttt{0xac}, \texttt{0xc6}, \texttt{0xc9}, \texttt{0xf0}, \texttt{0xff}\} \ .$$

3 Summary of Attacks Presented in DPACv4

Since the launch of DPACv4 in July 2013, 28 attacks have been submitted and evaluated. The results of all these attacks along with their brief description is available on the website of the contest. In general, the submitted attacks can be classified in two categories: *profiling based attacks* and *non-profiling based attacks*.

Some of the attacks submitted under the DPACv4 framework proved to be very efficient. For instance, in the profiling based attack category, 14 attacks have been proposed. The best attack in this category can break the implementation and recover the secret key with **a single trace** (attack phase). On the other hand, for the non-profiling based attacks, the best attack takes as low as **14 side-channel traces** to recover the secret key. In the following we focus on non-profiling attacks.

The first attack which is a univariate correlation power attack (CPA [2]) was proposed by Moradi et al. [14]. This attack exploits a vulnerability which arises from a basic design error. A vulnerability in RSM arises when a sbox input x_i masked with mask m_i, is written over by a sbox output y_i masked with m_{i+1} in the same register. The activity of the register can be written as $(x_i \oplus y_i) \oplus (m_i \oplus m_{i+1})$. Now under the RSM countermeasure both the mask m_i and m_{i+1} are balanced. The set of mask for RSM belong to a code and carefully chosen to satisfy certain properties and provide desired security. However, the composite mask $m_i \oplus m_{i+1}$ turns out to be unbalanced. This unbalance leads to a first-order leakage which can be exploited by a simple univariate CPA.

The next attack by Kanghong et al. unrolls in two steps. In the first step, the attacker tries to guess the value of initial offset used for each encryption. The attackers exploit the fact that the Hamming weight HW of mask $m_0 \oplus m_{15}$ is 8, while for all other mask combinations $(m_i \oplus m_{i+1})$ it is 4. This difference in Hamming weight can be observed in DPACv4 traces and the temporal location of this maximum difference gives an idea of the offset. In the second and final step, an attacker can group all traces with the same offset and launch a univariate

CPA attack to recover the secret key. Kanghong et al. used 69 traces to recover the key.

Thereafter several attacks exploiting the same vulnerability were proposed. Each time the method to determine the offset was novel. Junrong et al. propose two attack using maximal difference to determine the offset and recovering the key in 110 traces. Zhou et al. use maximal difference and pattern matching to determine the secret key in 14 traces. Next , Nakai et al. retrieve the offset using F-Test followed by a CPA to find the key in 43 traces. Another attacker who remains anonymous uses a first order CPA to first recover the offset followed by a DPA to find the key.

Two more attacks belonging to the non-profiled category were proposed under DPACv4. Zhou et al. attacked RSM using a second order CPA attacks. This attack exploited the joint leakage which came from combination of sbox output with input mask m_i and plaintext blinding with mask m_{i+1}. Although the individual leakages of plaintext blinding and sbox output are masked, the joint leakage becomes unmasked, which can be exploited by a CPA attacks. The other attack was a collision attack. Firstly the attacks detects collision using Pearson's correlation to compute the 15 key differences between first byte and other 15 bytes of the key. Next the whole key can be recovered by a simple brute force attack.

Apart from the DPA Contest framework, few other attacks were published on the implementation proposed in DPACv4. Kutzner et al. [21] proposed several attacks on the hardware and software implementations of RSM. Considering the software implementation (as of DPACv4), two attacks were proposed. The first attack guesses the offset followed by univariate CPA. In other words, it exploits the same vulnerability as majority of the attacks. The second attack proposed was by Kutzner et al. is indeed unique. It exploits a property called **constant difference** in the RSM mask. Authors discovered that the difference in mask between m_i and m_{i+8} is constant. In other words, $m_i \oplus m_{i+8}$ is constant. This property was used to mount a $1^{st} - order$ correlation enhanced collision attack [15] to recover the secret key. A third (simulated) attack was presented on hardware RSM in the same paper [21]. We noticed that this attack can also be a potential threat to software implementation of RSM. It exploits the fact that the mask m_i used by sbox S_0 in the first round is same as used by S_7 in last round, which allows collision attacks.

Few other papers were also published which attacked the DPACv4 traces. Belgarric et al. [16] demonstrated practical bivariate attacks (using preprocessing tools like Discrete Hartley Transform) by attacking in frequency domain. Moreover, Ye et al. [17] proposed a couple of attacks based on mutual information and collisions to exploit LEMS like RSM. All the attacks in those two papers were possible owing to the fact that the attacker is aware of the predictable sequence of AES operations.

To summarize the threats exploited by attacks submitted in DPACv4, we can identify four implementation pitfalls:

1. The mask (m_i, m_{i+1}) although balanced by itself, were not balanced when XORed together $((m_i \oplus m_{i+1}))$.

2. Mask (m_0, m_{15}) have a higher Hamming distance than other mask, which leaks the the value of the offset.
3. As the offset is incremented by a constant in every round, it lead to predictable sequence of operations which can be exploited by collision attacks.
4. The unaltered and predictable sequence of operations allows combination of points, thereby leading to second-order CPA and collision attacks.

4 Proposition for Improving DPACv4 Implementation

In this section, we propose some improvements to the original DPACv4 implementation based on our know-how of its pitfalls. These pitfalls are discussed and analyzed in the previous section. As stated earlier, designing a perfect countermeasure is not possible. Trying to thwart all attacks at once is not an obvious task. Of course, some solutions proposed by Rivain and Prouff [18] and Coron [19] can be applied. However, it would lead either to explosion in implementation cost. In the following, we attempt to boost the security level of the AES RSM implementation at reasonable cost overhead. We discuss each of the pitfall in detail and make an attempt to fix it.

The first pitfall arises from the fact that the value $m_i \oplus m_{i+1}$ exist in the implementation, directly or indirectly. As stated earlier the mask m_i and m_{i+1} are balanced, but the value $m_i \oplus m_{i+1}$ is unbalanced. We analysed the code of DPACv4 implementation. The original code was written in `C` language. It is compiled using `avr-gcc` to generate assembly code. If we check the original `C` code, $m_i \oplus m_{i+1}$ is never computed itself. However on compilation with `avr-gcc` certain instances of such nature may occur. `avr-gcc` reuses several general purpose registers and 2-stage pipeline to optimize the design. Suppose there exist a value $x \oplus m_i$ in a register or pipeline. This value is followed by $y \oplus m_{i+1}$. The side-channel activity at next clock will correspond to $x \oplus y \oplus m_i \oplus m_{i+1}$, which is unbalanced.

Now looking into the DPACv4 implementation, the result of plaintext blinding $x_i \oplus m_i$ stored in a register is overwritten by its sbox output $\mathsf{MaskedSubBytes}(x_i \oplus m_i)$. The latter term can be written as $\mathsf{SubBytes}(x_i) \oplus m_{i+1}$ It is well known that the activity of the register follows the value update model i.e. $x_i \oplus \mathsf{SubBytes}(x_i) \oplus m_i \oplus m_{i+1}$. Thus the accidental $m_i \oplus m_{i+1}$ leakage occurs which can be exploited in side-channel.

A straightforward way to avoid accidental computation of the form $m_i \oplus m_{i+1}$ in the implementation flow, is to rewrite the complete code in assembly language. However writing assembly code is a tedious and error-prone task. A common practice is to write only the sensitive modules of the code in assembly. This is considered as best practice to avoid any surprises from compilation. Another precaution which must be taken at this stage is register precharge. If we precharge every general purpose register to '0' value before writing in a new value, we can avoid all leakages of form $m_i \oplus m_{i+1}$. By ensuring these two conditions, one can get rid of accidental univariate leakage like the one presented in [14].

The second pitfall identified in DPACv4 implementation is that the value of offset is leaked in side-channel. In fact, the mask $(m_0 = 0x00, m_{15} = 0xFF)$ have considerable higher Hamming distance of 8. All other adjacent masks have a Hamming distance of 4, which can be identified in side-channel traces The fact that, after each sbox, $\mathsf{offset} \leftarrow \mathsf{offset} + 1$, allows to retrieve the offset since there is a constant temporal distance between mask $0x00$ and $0xFF$. We consider this vulnerability to be very serious as it was exploited by most of the attacks submitted under DPACv4. Once the offset is know to the attacker, the attacker can easily sort the traces with same offset. Same offset for a set of traces translates to same mask values *i.e.* a constant mask denoted by m_k. Since the mask is constant, the Pearson correlation $\rho(x \oplus m_k, y)$ simplifies to $\rho(x, y)$. This is equivalent to a totally unmasked implementation.

To protect against such attacks, we propose to use a random offset for each sbox. Although we use a random offset for each sbox, the basic set of 16 mask remains unchanged. Therefore all the security proofs which apply to RSM also apply to our implementation. The random offset is applied by using a random array of 16 independent indices. This array is generate to address the array of 16 masks independently for each sbox. Unlike the original implementation, this implementation can (sometimes) use same mask for multiple sboxes. Moreover by using independant offset for each sbox, we also solve the problem of collision attacks as proposed in [21]. The correlation-enhanced collision attack [21] exploits the fact that $SubBytes(x_i+k_i) = SubBytes(x_{i+8})+k_{i+8}+0x95$. By randomizing the manipulation of mask of indices i and $i+8$, this attack is no more possible, as m_i and m_{i+8} will not have same temporal distance. Similarly in [21] the collision attack exploiting the first and last round becomes irrelevant. The overhead associated with this countermeasure is that the set of $\mathsf{MaskCompensation}$ becomes very large to store in the memory. To solve this problem we compute the $\mathsf{MaskCompensation}$ on the fly which has a time penalty as overhead.

Finally, there were certain bivariate and higher order attacks proposed under the framework of DPACv4. In [7], authors tweak the original RSM scheme for software implementation to claim first degree security. Thus if higher order attacks work on RSM, it is as expected. There are two possible ways to boost the security level of this implementation. The first way is to modify the masking scheme in order to resist higher-order attacks [20]. On the other hand, one can combine countermeasures to boost the security level while keeping overhead in check. We choose the second method and use shuffling [4] as an additional countermeasure. As the prime targets of SCA are first and last rounds of AES, we only shuffle the order of sbox execution of first and last round of the AES. This shuffling is performed by drawing a random permutation for indices of execution of sboxes for first and last round for each encryption. In the middle rounds, the sboxes are executed as before i.e. 8 even sbox indices followed by remain 8 odd indices. Since the window of execution of the concerned sboxes will change, the selection of trace windows for combination will not be easy. For the same reason, attack proposed in [17] becomes impractical.

The attacks on DPAcv4 and the corresponding countermeasures proposed in this section are all summarized in Tab. 1.

Table 1. Attacks on DPACv4 implementation and corresponding countermeasures proposed in this article

Attacks \ Countermeasures		ASM (instead of C)	Registers precharge	One mask per sbox	Shuffling
First-order attack	[14]	x	x		
Recover the offset	[9]			.	x
Collision on the sbox	[21]			x	x
Collision 1st-last rounds	[21]			x	x
Bivariate attacks	[16]				x
MIA	[17]				x

4.1 Target Platform

To analyze our implementation we use the same platform as of DPACv4. The target is a 8-bit AVR microcontroller Atmega163 embedded in a smartcard. It contains 16Kb of in-system programmable flash, 512 bytes of EEPROM, 1Kb of internal SRAM and 32 general purpose working registers. The smartcard is read using a simple reader interface mounted on SASEBO-W board and controlled by Xilinx Spartan-VI FPGA. The traces are acquired using a LeCroy WaveRunner 6100A oscilloscope using an EM probe. The acquisition bandwidth is 200 MHz and the sampling rate $F_S = 500$ MS/s.

4.2 Implementation

The proposed implementation is written in assembly language and carefully checked to avoid most identified pitfalls. This implementation takes the well optimized Rijndael furious and DPACV4 implementations as references.

Tab. 2 compares the unprotected Rijndael furious and DPACv4 implementations with our improved design. Please note that the numbers includes the cost of key expansion as well as the embedded OS used in DPACV4. Rewriting the sensitive part in assembly actually accelerated the proposed design compared to original one. Please note that Tab. 2 does not take into account the cost of embedded CSPRNG which is used to generate the randomness needed for the shuffled masking scheme. We make sure that the blinding operation is performed in a specific order to avoid some horizontal attacks. Also, direct manipulation of private shares with known variables is avoided. For example the key should first be blinded with the random mask before blinding the plaintext. The improved algorithm running on the smartcard is described in Alg. 1.

Algorithm 1. Modified AES implementation to overcome pitfalls of DPACv4.

```
Input  : 16-bytes Plaintext $X$ $[\![X_0, X_1 \cdots X_{15}]\!]$,
         SubKeys, 15 16-bytes constants $\mathsf{RoundKey}[r]$, $r \in [\![0, 14]\!]$,
         16 masks of 8 bit, called $\mathsf{Mask}[]$
Output: 16-bytes Ciphertext $X$ $[\![X_0, X_1 \cdots X_{15}]\!]$

/* Draw 16 4-bit (uniformly random, unknown) offset[] for the key blinding */

/* Draw of 2 shuffling functions (uniformly random permutations), $\mathsf{Shuffle0}, \mathsf{Shuffle13} : [\![0, 15]\!] \to [\![0, 15]\!]$, bijective */

$\mathsf{RoundKey}[0] \leftarrow \mathsf{RoundKey}[0] \oplus \mathsf{Mask}[\mathsf{offset}[]]$
                              /* All rounds but the last one */
for $r \in [\![0, 12]\!]$ do
    $X = X \oplus \mathsf{RoundKey}[r]$                /* AddRoundKey */
    if $r = 0$ then
        for $i \in \mathsf{Shuffle0}([\![0, 15]\!])$ do
            $X_i = \mathsf{MaskedSubBytes}_{\mathsf{offset}[i]+r}(X_i)$
        end
    else
        for $i \in [\![0, 15]\!]$ do
            $X_i = \mathsf{MaskedSubBytes}_{\mathsf{offset}[i]+r}(X_i)$
        end
    end
    $X = \mathsf{ShiftRows}(X)$
    $X = \mathsf{MixColumns}(X)$
    for $i \in [\![0, 15]\!]$ do
        $\mathsf{MaskCompensation}[i] =$
        $\mathsf{ShiftRows}(\mathsf{MixColumns}(\mathsf{Mask}[\mathsf{offset}[i]+(r+1)])) \oplus \mathsf{Mask}[(\mathsf{offset}[i]+(r+1))]$
    end
    $X = X \oplus \mathsf{MaskCompensation}[]$
end
                              /* Last round */
$X = X \oplus \mathsf{RoundKey}[13]$
for $i \in \mathsf{Shuffle13}([\![0, 15]\!])$ do
    $X_{\mathsf{Shuffle}}[i] = \mathsf{MaskedSubBytes}_{\mathsf{offset}[i]+13}(X_i)$
end
$X = \mathsf{ShiftRows}(X)$
$X = X \oplus \mathsf{RoundKey}[14]$
                              /* Ciphertext unmasking */
for $i \in [\![0, 15]\!]$ do
    $\mathsf{MaskCompensationLastRound}[i] =$
    $\mathsf{ShiftRows}(\mathsf{Mask}[\mathsf{offset}[i] + 14]) \oplus \mathsf{Mask}[(\mathsf{offset}[i] + 14)]$
end
$X = X \oplus \mathsf{MaskCompensationLastRound}[]$
```

Table 2. Cost Complexity of the original (DPACv4) over the new implementation and Rijndael Furious

Architecture	Rijndael Furious	Original (protected)	Improved (protected)	Overhead
Code Size (bytes)	2596	11136	17847	60%
RAM (bytes)	1	8	12	50%
Number of cycles	3579	113600	16004	−86%

4.3 Shuffling Algorithms

To generate the shuffle, we propose two algorithms:

1. the first one (Alg. 2) generates a full entropy permutation of $[\![0, 2^n - 1]\!]$, and works in $\mathcal{O}(n^2 \log(2^n))$ time;
2. the second one (Alg. 3) generates a low entropy permutation of $[\![0, 2^n - 1]\!]$, but works in linear time $\mathcal{O}(n)$.

Alg. 2 redraws numbers repeatedly till there is no collision. Notice that we suggest to draw numbers in $\{0, 1, \ldots, 2^n - 1\}$, because it is easy to draw uniformly n bits. Instead, randomly drawing numbers in an interval is not trivial (applying a "modulo" would break the uniformity).

Lemma 1. *The expected running time of Alg. 2 is* $2^n \sum_{m=1}^{2^n} \frac{1}{m}$*, that is equivalent to* $\mathcal{O}(2^n \log(2^n))$ *for large values of* n*.*

Proof. See Appendix B.

Algorithm 2. Full Entropy Shuffling

```
input  : None
output: A permutation $\mathbb{F}_2^n \mapsto \mathbb{F}_2^n$
Initialize a vector of $2^n$ elements of $\mathbb{F}_2^n$ ;
for $\omega \in \{0, 1, \ldots, 2^n - 1\}$ do // Scrambling
    $r \leftarrow_{\mathcal{R}} \mathcal{U}([\![0, 2^n - 1]\!])$ ;
    while $r \in S[0, i-1]$ do
        $r \leftarrow_{\mathcal{R}} \mathcal{U}([\![0, 2^n - 1]\!])$ ;
    end
    $S[i] \leftarrow r$ ;
end
return $S$
```

Alg. 3 is inspired from the key scheduling of RC4.

Algorithm 3. Low Entropy Shuffling

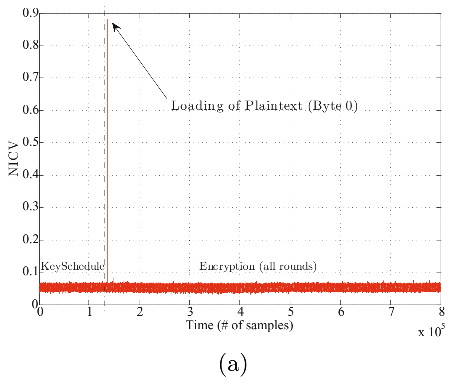
```
input  : $k[2^n]$, an array of $2^n$ elements of $\mathbb{F}_2^n$
output: A permutation $\mathbb{F}_2^n \mapsto \mathbb{F}_2^n$

$S[2^n]$, an array of $2^n$ elements of $\mathbb{F}_2^n$ ;
for $\omega \in \{0, 1, \ldots, 2^n - 1\}$ do // Initialisation
    $\mathsf{S}[\omega] \leftarrow \omega$
end
$j \leftarrow 0$ for $\omega \in \{0, 1, \ldots, 2^n - 1\}$ do // Scrambling
    $j \leftarrow j + S[\omega] + k[\omega mod 2^n]$;
    swap($S[\omega], S[j]$) ;
end
return $S$
```

5 Security Evaluation

We acquired 32K side-channel traces for the proposed implementation using the setup described above. The plaintexts and the key used were same as of DPACv4. The main aim is to check for any first-order or univariate leakage present in the implementation. To do so, we rely on leakage detection technique. More precisely we use Normalized Inter-Class Variance (NICV [22]). NICV detects any univariate leakage present in the side-channel traces and does not depend on a leakage model. It is computed with respect to public parameters like plaintext or ciphertext. NICV is expressed as:

$$NICV = \frac{\mathsf{Var}\left[\mathbb{E}\left[Y|X\right]\right]}{\mathsf{Var}\left[Y\right]},$$

where Y denotes side-channel traces and X represent a chosen part of plaintext or ciphertext. We compute NICV with respect to a input plaintext byte for the collected traces. The results are shown in Fig 1(a). It can be deduced from Fig 1(a), that no univariate leakage is present in the improved implementation in the SubBytes and further. We can see two big peaks during the initial AddRoundKey in figure which indicate presence of a possible univariate leakage. We further investigated the peak using a univariate CPA attacks. Indeed these peaks correspond to the loading of the raw plaintext byte into different section of the card i.e. memory and ALU. This leakage does not contain any information about the key used and therefore not sensitive. We can also see this non-sensitive leakage on the NICV computed on traces of the original implementation of DPACv4 as shown in Fig 1(b). Moreover in Fig. 1(b), we detect a univariate leakage related to the plaintext during the SubBytes operation. Such leakage can be sensitive. We further investigated the leakage in the SubBytes of original DPACv4 implementation. It turned out to be the same leakage as exploited by Moradi et al [14].



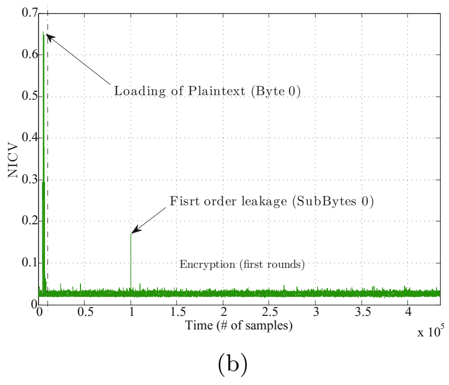


Fig. 1. NICV computed on the (a) proposed implementation; (b) original implementation of DPACv4

Next we investigate the leakage corresponding to the offset. In order to show that our new implementation is less prone to folding attacks than the DPACV4 implementation we check that traces contains less leakage points related to the offset value. We computed NICV with respect to the 4 bit offset used in the first implementation. We can see in Fig. 2 that there are 48 significant peaks. Those leakage points corresponds to the loading of the single 4 bit offset index used to address the mask table, the sBox and the mask correction table. If the device leaks in value, a single folding attack on one of those leakage point can be sufficient to recover the full key. If the offset is partially leaked at each leakage point , the attacker can exploit multiple leakage point to mount more robust folding attacks.

Then we computed NICV with respect to the first 4-bit offset of 16 on our new implementation. The results are shown in Fig 3. We can see three big peaks. Those tree peaks corresponds to the loading of the index that is used to read the mask,the sBox and the mask correction table in memory. Those leakage are sensitive because it provide information on the byte of mask used to blind the key. However, it is no longer possible to mount horizontal attacks since each byte of mask is selected by a different random 4 bit offset. Knowing those leakage point can however be used to mount "folding" type attacks, provided that the target leaks in value. If the target appears to leak in value, 16 folding attacks are however not sufficient to recover the full key because the attacker should also fold the dataset depending the 16 4-bit random shuffle. If the offset values are partially leaked , only 3 leakage point per offset nibble are available to guess the leaked value, which is not sufficient.

5.1 Insight on Horizontal Attacks

In this section, we compare a full entropic sbox masking against improved RSM proposed in Sec 4. A way to mask the non linear sbox is to use sbox

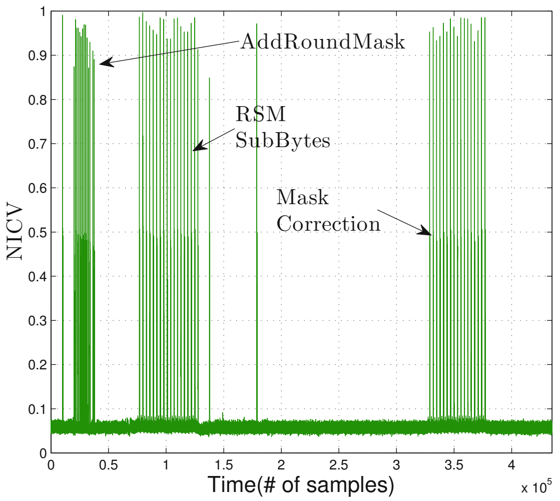


Fig. 2. NICV computed on the first 4 bit offset of the proposed implementation (first round + KeySchedule)

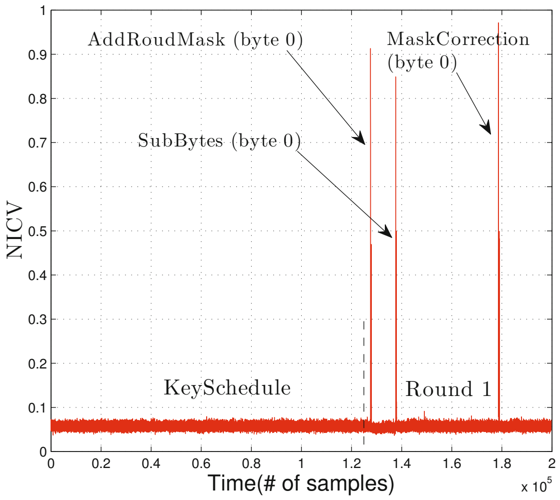


Fig. 3. NICV computed on the first 4 bit offset of the DPACV4 implementation (first round + KeySchedule)

recomputation [23]. As presented in [24] this kind of masking scheme can be defeated by using “horizontal” attacks to first recover the mask and then performing first order attacks. These attacks are possible due to the fact that the mask is used 256 times (in the case of AES) during the sbox recomputation. Mainly the input mask of the sbox is sequentially XORed with all the possible values in $\mathbb{F}_2^8$. These leakages allow an attacker to recover the value of the mask using for example a CPA.

In DPACv4, it was also possible to build ”horizontal” attacks to recover the random offset, and then the mask of all the sbox outputs. Indeed for each plaintext byte there was the leakage depending on the following operation: $x_i \oplus m_i$. Then as the sequence of mask is known there is only 16 possible guesses, corresponding to 16 masks, to recover the mask using for example a CPA.

Remark 2. Note that there are 256 different exploitable leakages in the case of the sbox recomputation and only 16 for RSM. But the results of the “horizontal” attacks on RSM allows to recover the mask of the sixteen bytes of the states whereas (depending on the implementation) the “horizontal” attack on the sbox recomputation allows to recover the mask of only one byte of the state.

In our proposition, a random offset is used to mask each byte and it is no longer possible to perform “horizontal” attack. Indeed it is necessary to guess 16^{16} values. Moreover, the shuffling makes the attack even more difficult as it is necessary to guess the 16! possible orders of plaintext.

The Success Rate is given by the formula [10]: $SR = 1 - e^{-n \times k}$ where n is the number of traces, in our case the number of different leakages depending on the mask, and k is a first order exponent (obtained from a Chernov bound). Figure 4 shows the difference of the success rate for the recovering the mask for improved RSM and the sbox recomputation.

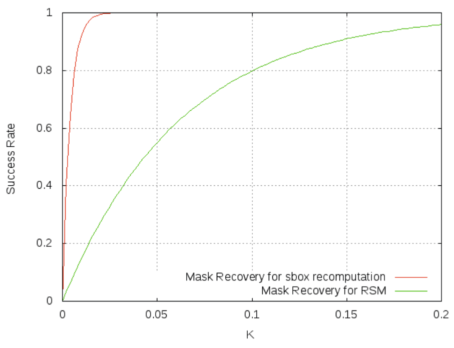


Fig. 4. Difference of Success Rate for normal sbox recomputation vs our proposition of RSM

Remark 3. The sbox recomputation can also be done in a random order but it is necessary to generate a permutation on 256 value. This generation could be costly (see Alg. 2 and 3).

6 Conclusions and Perspectives

LEMS has its own advantages and shortcomings. An example of LEMS was proposed in DPACv4, where researchers from all over the world were able to attack a common implementations. 18 profiled and non-profiled attacks were proposed revealing 4 major pitfalls of the proposed implementation. In this paper, we analyze these pitfalls and propose an improved implementation. Our results demonstrate that it is possible to resist the non-profiled attacks at an overhead of 27% in code size, 50% in memory and 1.5% in computation time.

Disclaimer

The exact specifications of the improved implementation of the DPA contest v4 will be posted on the official website and related social media [9].

Acknowledgments. Authors are grateful to Guillaume Duc for the animation of the DPA contests, and to all the DPA contest participants, who made these competitions live and very active.

References

1. Kocher, P.C., Jaffe, J., Jun, B.: Differential Power Analysis. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 388–397. Springer, Heidelberg (1999)
2. Brier, E., Clavier, C., Olivier, F.: Correlation Power Analysis with a Leakage Model. In: Joye, M., Quisquater, J.-J. (eds.) CHES 2004. LNCS, vol. 3156, pp. 16–29. Springer, Heidelberg (2004)
3. Herbst, C., Oswald, E., Mangard, S.: An AES Smart Card Implementation Resistant to Power Analysis Attacks. In: Zhou, J., Yung, M., Bao, F. (eds.) ACNS 2006. LNCS, vol. 3989, pp. 239–252. Springer, Heidelberg (2006)
4. Rivain, M., Prouff, E., Doget, J.: Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. Cryptology ePrint Archive, Report 2009/420 (2009), `http://eprint.iacr.org/2009/420`
5. Rauzy, P., Guilley, S., Najm, Z.: Formally Proved Security of Assembly Code Against Leakage. IACR Cryptology ePrint Archive 2013, 554 (2013)
6. Nassar, M., Souissi, Y., Guilley, S., Danger, J.L.: RSM: a Small and Fast Countermeasure for AES, Secure against First- and Second-order Zero-Offset SCAs. In: DATE, Dresden, Germany, pp. 1173–1178. IEEE Computer Society (2012) (TRACK A: "Application Design", TOPIC A5: "Secure Systems")
7. Bhasin, S., Danger, J.L., Guilley, S., Najm, Z.: A Low-Entropy First-Degree Secure Provable Masking Scheme for Resource-Constrained Devices. In: Proceedings of the Workshop on Embedded Systems Security, WESS 2013, Montreal, Quebec, Canada, pp. 7:1–7:10. ACM, New York (2013), doi:10.1145/2527317.2527324

8. TELECOM ParisTech SEN research group: DPA Contest (1st edn.) (2008–2009), `http://www.DPAcontest.org/`
9. TELECOM ParisTech SEN research group: DPA Contest (4th edn.) (2013–2014), `http://www.DPAcontest.org/v4/`
10. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards Sound Approaches to Counteract Power-Analysis Attacks. In: Wiener, M. (ed.) CRYPTO 1999. LNCS, vol. 1666, pp. 398–540. Springer, Heidelberg (1999)
11. Prouff, E., Rivain, M.: A Generic Method for Secure SBox Implementation. In: Kim, S., Yung, M., Lee, H.-W. (eds.) WISA 2007. LNCS, vol. 4867, pp. 227–244. Springer, Heidelberg (2008)
12. Hedayat, A.S., Sloane, N.J.A., Stufken, J.: Orthogonal Arrays, Theory and Applications. Springer series in statistics. Springer, New York (1999) ISBN 978-0-387-98766-8
13. Grosso, V., Standaert, F.-X., Prouff, E.: Low Entropy Masking Schemes, Revisited. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 33–43. Springer, Heidelberg (2014)
14. Moradi, A., Guilley, S., Heuser, A.: Detecting Hidden Leakages. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 2014. LNCS, vol. 8479, pp. 324–342. Springer, Heidelberg (2014)
15. Clavier, C., Feix, B., Gagnerot, G., Roussellet, M., Verneuil, V.: Improved Collision-Correlation Power Analysis on First Order Protected AES. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 49–62. Springer, Heidelberg (2011)
16. Belgarric, P., et al.: Time-Frequency Analysis for Second-Order Attacks. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 108–122. Springer, Heidelberg (2014)
17. Ye, X., Eisenbarth, T.: On the Vulnerability of Low Entropy Masking Schemes. In: Francillon, A., Rohatgi, P. (eds.) CARDIS 2013. LNCS, vol. 8419, pp. 44–60. Springer, Heidelberg (2014)
18. Rivain, M., Prouff, E., Doget, J.: Higher-Order Masking and Shuffling for Software Implementations of Block Ciphers. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 171–188. Springer, Heidelberg (2009)
19. Coron, J.-S.: Higher Order Masking of Look-Up Tables. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 441–458. Springer, Heidelberg (2014)
20. Rivain, M., Prouff, E.: Provably Secure Higher-Order Masking of AES. In: Mangard, S., Standaert, F.-X. (eds.) CHES 2010. LNCS, vol. 6225, pp. 413–427. Springer, Heidelberg (2010)
21. Kutzner, S., Poschmann, A.: On the Security of RSM — Presenting 5 First- and Second-order Attacks. In: Prouff, E. (ed.) COSADE 2014. LNCS, vol. 8622, pp. 299–312. Springer, Heidelberg (2014)
22. Bhasin, S., Danger, J.L., Guilley, S., Najm, Z.: Side-channel Leakage and Trace Compression Using Normalized Inter-class Variance. In: Proceedings of the Third Workshop on Hardware and Architectural Support for Security and Privacy, HASP 2014, pp. 7:1–7:9. ACM, New York (2014)
23. Akkar, M.-L., Giraud, C.: An Implementation of DES and AES, Secure against Some Attacks. In: Koç, Ç.K., Naccache, D., Paar, C. (eds.) CHES 2001. LNCS, vol. 2162, pp. 309–318. Springer, Heidelberg (2001)

24. Tunstall, M., Whitnall, C., Oswald, E.: Masking Tables – An Underestimated Security Risk. In: Moriai, S. (ed.) FSE 2013. LNCS, vol. 8424, pp. 425–444. Springer, Heidelberg (2014)
25. Clavier, C., Danger, J.-L., Duc, G., Abdelaziz Elaabid, M., Gérard, B., Guilley, S., Heuser, A., Kasper, M., Li, Y., Lomné, V., Nakatsu, D., Ohta, K., Sakiyama, K., Sauvage, L., Schindler, W., Stöttinger, M., Veyrat-Charvillon, N., Walle, M., Wurcker, A.: Practical improvements of side-channel attacks on AES: feedback from the 2nd DPA contest. Journal of Cryptographic Engineering, 1–16 (2014)

A Algorithm of DPACv4 Implementation

The algorithm running on the smartcard for DPACv4 is described in Alg. 4.

Algorithm 4. AES implementation used for the DPACv4 (Source: [9]).

```
Input  : 16-bytes Plaintext $X$ $[\![X_0, X_1 \cdots X_{15}]\!]$,
         Key, 15 16-bytes constants $\mathsf{RoundKey}[r]$, $r \in [\![0, 14]\!]$
Output: 16-bytes Ciphertext $X$ $[\![X_0, X_1 \cdots X_{15}]\!]$

Draw a random offset, uniformly in $[\![0, 15]\!]$
$X = X \oplus \mathsf{Mask}_{\mathsf{offset}}$                /* Plaintext blinding */

                          /* All rounds but the last one */
for $r \in [\![0, 12]\!]$ do
    $X = X \oplus \mathsf{RoundKey}[r]$                  /* AddRoundKey */
    for $i \in [\![0, 15]\!]$ do
        $X_i = \mathsf{MaskedSubBytes}_{\mathsf{offset}+i+r}(X_i)$
    end
    $X = \mathsf{ShiftRows}(X)$
    $X = \mathsf{MixColumns}(X)$
    $X = X \oplus \mathsf{MaskCompensation}_{\mathsf{offset}+1+r}$
end
                                  /* Last round */
$X = X \oplus \mathsf{RoundKey}[13]$
for $i \in [\![0, 15]\!]$ do
    $X_i = \mathsf{MaskedSubBytes}_{\mathsf{offset}+13+r}(X_i)$
end
$X = \mathsf{ShiftRows}(X)$
$X = X \oplus \mathsf{RoundKey}[14]$
                            /* Ciphertext unmasking */
$X = X \oplus \mathsf{MaskCompensationLastRound}_{\mathsf{offset}+14}$
```

B Proof of Lemma 1

The running time of Alg. 2 is probabilistic because of the conditional redraws at line 2. Let i, $0 \leq i < 2^n$, be the number of values already chosen. Then, a uniformly drawn value r in $[\![0, 2^n - 1]\!]$ is a new value with probability $(2^n - i)/2^n$. If it is not a new value, then j redraws are required, with probability

$$\left(\frac{i}{2^n}\right)^j \times \frac{2^n - i}{2^n} \ .$$

Thus, the average number of random number drawing is:

$$\begin{aligned} &1 + \sum_{j=1}^{+\infty} j \left(\frac{i}{2^n}\right)^j \times \frac{2^n - i}{2^n} \\ &= 1 + \sum_{j=1}^{+\infty} j \left(\frac{i}{2^n}\right)^{j-1} \times \frac{(2^n - i)i}{2^{2n}} \\ &= 1 + \frac{i}{2^n - i} \ . \end{aligned}$$

because $\frac{1}{(1-x)^2} = \sum_{i=1}^{\infty} ix^{i-1}$ for all $x \in \mathbb{R}$ such that $|x| < 1$.

Thus, the average time of Alg. 2 is

$$\begin{aligned} &\sum_{i=0}^{2^n-1} 1 + \frac{i}{2^n - i} \\ &= 2^n + \sum_{m=1}^{2^n} \frac{2^n - m}{m} \qquad (m \leftarrow 2^n - i) \\ &= 2^n \sum_{m=1}^{2^n} \frac{1}{m} \ . \end{aligned}$$

Now,

$$\lim_{N \to +\infty} \sum_{m=1}^{N} \frac{1}{m} - \ln N = -\gamma \ ,$$

where γ is the Euler-Mascheroni constant ($\gamma \approx 0.577$). Thus, the average running time of Alg. 2 is equivalent to $2^n \ln(2^n)$ when n tends to the infinity.