A new DPA Countermeasure based on Permutation Tables

Jean-Sébastien Coron

University of Luxembourg
jean-sebastien.coron@uni.lu

Abstract. We propose and analyse a new countermeasure against Differential Power Analysis (DPA) for the AES encryption algorithm, based on permutation tables. As opposed to existing AES countermeasures, it does not use random masking. We prove that our new countermeasure is resistant against first-order DPA; we also show that it is quite efficient in practice.

1 Introduction

The AES [4] encryption algorithm is with DES the most widely used encryption algorithm. However, it is easy to see that without modification, AES is vulnerable to Differential Power Analysis as introduced by Kocher *et al.* in [6, 7]. A DPA attack consists in extracting information about the secret key of a cryptographic algorithm, by studying the power consumption of the electronic device during the execution of the algorithm. The attack was first described on the DES encryption scheme, but it was soon understood that the attack could easily be extended to other symmetrical cryptosystems such as the AES, and also to public-key cryptosystems such as RSA and Elliptic-Curves Cryptosystems [3].

A common technique to protect secret-key algorithms against side-channel attacks consists in masking all data with a random integer, as suggested in [2]. The masked data and the random integer are then processed separately and eventually recombined at the end of the algorithm. An attacker trying to analyse power consumption at a single point will obtain only random values; therefore, the implementation will be secure against first-order Differential Power Analysis (DPA). In order to obtain valuable information about the key, the attacker must correlate the power consumption at multiple points during the execution of the algorithm; this is called a High Order Differential Power Analysis (HO-DPA); such attack usually requires a much larger number of power consumption curves, which makes it infeasible in practice if the number of executions is limited (for example, by using a counter). Many AES countermeasures have been described based on random masking [1, 5, 9, 10, 12].

In this article we propose a different countermeasure against DPA for AES, based on permutation tables. The main difference with existing AES countermeasures is that it avoids random masking; in practice this can be an advantage because random masking is subject to numerous patents [7]. We prove that our

countermeasure is resistant against first-order DPA (like the random masking countermeasure) and we show that its efficiency is comparable to that of the random masking countermeasure.

It works as follows: at initialisation time a randomised permutation table is generated in RAM; this permutation table is then applied to the message and to the key; then all intermediate variables that appear during the course of the algorithm remain in permuted form; eventually the inverse permutation is applied to obtain the resulting ciphertext.

We also describe a technique to reduce the RAM consumption of our permutation table countermeasure, at the cost of increasing the running time. Our technique is based on a compression scheme proposed in [11] for the classical random masking countermeasure; here we adapt this scheme to permutation tables. We show that this variant is also secure against first-order DPA. Finally, we also provide the result of implementations that show that our countermeasure is reasonably efficient in practice, as it is only roughly four times slower than the classical masking countermeasure, for a comparable RAM requirement (see Table 1 in Section 7 for a detailed comparison).

2 The AES encryption algorithm

In this section we recall the main operations involved in the AES algorithm. We refer to [4] for a full description. AES operates on a 4×4 array of bytes $s_{i,j}$, termed the state. For encryption, each AES round (except the last) consists of four stages:

1. **AddRoundKey**: each byte of the state is xored with the round key $k_{i,j}$, derived from the key schedule:

$$s_{i,j} \leftarrow s_{i,j} \oplus k_{i,j}$$

2. **SubBytes**: each byte of the state is updated using an 8-bit S-box:

$$s_{i,j} \leftarrow S(s_{i,j})$$

3. **ShiftRows**: the bytes of the state are cyclically shifted in each row by a certain offset; the first row is left unchanged.
4. **MixColumns**: the bytes of the state are modified column by column as follows:

$$\begin{aligned}
s'_{0,c} &\leftarrow (\mathsf{02} \cdot s_{0,c}) \oplus (\mathsf{03} \cdot s_{1,c}) \oplus s_{2,c} \oplus s_{3,c} \\
s'_{1,c} &\leftarrow s_{0,c} \oplus (\mathsf{02} \cdot s_{1,c}) \oplus (\mathsf{03} \cdot s_{2,c}) \oplus s_{3,c} \\
s'_{2,c} &\leftarrow s_{0,c} \oplus s_{1,c} \oplus (\mathsf{02} \cdot s_{2,c}) \oplus (\mathsf{03} \cdot s_{3,c}) \\
s'_{3,c} &\leftarrow (\mathsf{03} \cdot s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\mathsf{02} \cdot s_{3,c})
\end{aligned}$$

The pseudo-code for AES encryption with a 128-bit key is recalled in Figure 1 in Appendix. The word array w contains the round keys that are generated by the key-schedule algorithm. We refer to [4] for a more detailed description.

For decryption, each round (except the last) consists in the following operations:

1. **InvShiftRows**: is the inverse of the ShiftRows operation. The bytes of the state are cyclically shifted in each row by a certain offset; the first row is left unchanged.
2. **InvSubBytes**: is the inverse of the SubBytes operation. The inverse S-box S^{-1} is applied on each byte of the state.
3. **AddRoundKey**: the operation is equal to its own inverse.
4. **InvMixColumns**: is the inverse of the MixColumns operation. The bytes of the state are modified column by column as follows:

$$
\begin{aligned}
s'_{0,c} &\leftarrow (\mathtt{0e} \cdot s_{0,c}) \oplus (\mathtt{0b} \cdot s_{1,c}) \oplus (\mathtt{0d} \cdot s_{2,c}) \oplus (\mathtt{09} \cdot s_{3,c}) \\
s'_{1,c} &\leftarrow (\mathtt{09} \cdot s_{0,c}) \oplus (\mathtt{0e} \cdot s_{1,c}) \oplus (\mathtt{0b} \cdot s_{2,c}) \oplus (\mathtt{0d} \cdot s_{3,c}) \\
s'_{2,c} &\leftarrow (\mathtt{0d} \cdot s_{0,c}) \oplus (\mathtt{09} \cdot s_{1,c}) \oplus (\mathtt{0e} \cdot s_{2,c}) \oplus (\mathtt{0b} \cdot s_{3,c}) \\
s'_{3,c} &\leftarrow (\mathtt{0b} \cdot s_{0,c}) \oplus (\mathtt{0d} \cdot s_{1,c}) \oplus (\mathtt{09} \cdot s_{2,c}) \oplus (\mathtt{0e} \cdot s_{3,c})
\end{aligned}
$$

The pseudo-code for the inverse cipher is recalled in Figure 2 in Appendix. Finally, the key-schedule is based on the following operations:

1. **SubWord**: takes a four-byte input word and applies the S-box S to each of the four bytes.
2. **RotWord**: takes a word $[a_0, a_1, a_2, a_3]$ as input and performs a cyclic permutation to return $[a_1, a_2, a_3, a_0]$.
3. **Xor with Rcon**: takes as input a 32-bits word and xor it with the round constant word array $\mathsf{Rcon}[\mathsf{i}] = [(\mathtt{02})^{i-1}, \mathtt{00}, \mathtt{00}, \mathtt{00}]$, for round $1 \leq i \leq 10$.

We refer to [4] for a full description of the key-schedule.

3 The Permutation Table Countermeasure

In this section we describe our basic countermeasure with permutation tables. A variant with a time-memory trade-off is described in Section 6.

Our countermeasure consists in using a randomised representation of the state variables, using two independent 4-bit permutation tables p_1 and p_2 that are freshly generated before each new execution of the algorithm. More precisely, every intermediate byte $x = x_h \| x_l$, where x_h is the high nibble and x_l is the low nibble, will be represented in the following form:

$$P(x) = p_2(x_h) \| p_1(x_l)$$

This permuted representation is then used throughout the execution of AES. Eventually the original representation is recovered at the end of the encryption algorithm, by applying the inverse permutation.

3.1 Generation of Permutation Tables p_1 and p_2

The 4-bit permutation tables p_1 and p_2 are generated for each new execution of the algorithm as follows. Let s_0 be a fixed 4-bit permutation table; one can take for example:

$$s_0 = [14, 6, 0, 5, 9, 1, 4, 15, 8, 10, 12, 2, 3, 13, 11, 7]$$

One defines a sequence of permutations s_i for $i \geq 0$ as follows:

$$s_{i+1}(x) = s_0\left(s_i(x) \oplus k_i\right)$$

where each $k_i \in \{0,1\}^4$ is randomly generated. The 4-bit permutation p_1 is then defined as $p_1 := s_n$ for some n (in practice, one can take $n = 4$). One applies the same procedure to generate the other table p_2 (with independently generated k_i's). Every intermediate byte $x = x_h \| x_l$ that appear in AES is then represented as:

$$P(x) = p_2(x_h) \| p_1(x_l)$$

Therefore, P is a 8-bit permutation; its storage requires 16 bytes of RAM. In the following we explain how to use this permuted representation throughout the AES operations, so that the intermediate data are never manipulated in clear.

3.2 AddRoundKey

Given $P(x)$ and $P(y)$ we explain how to compute $P(x \oplus y)$ without manipulating x and y directly (since otherwise this would give a straightforward DPA attack).

We define the following two 8-bit to 4-bit *xor-tables*; for all $x', y' \in \{0,1\}^4$:

$$\mathsf{XT}_4^1(x', y') := p_1(p_1^{-1}(x') \oplus p_1^{-1}(y'))$$
$$\mathsf{XT}_4^2(x', y') := p_2(p_2^{-1}(x') \oplus p_2^{-1}(y'))$$

Those two tables require a total of 256 bytes in memory. Then given $p_1(x)$ and $p_1(y)$ one can compute $p_1(x \oplus y)$ using:

$$\mathsf{XT}_4^1(p_1(x), p_1(y)) = p_1(x \oplus y)$$

for all $x, y \in \{0,1\}^4$. Similarly, we have:

$$\mathsf{XT}_4^2(p_2(x), p_2(y)) = p_2(x \oplus y)$$

Using those two tables we define the following function for all $x', y' \in \{0,1\}^8$, where $x' = x'_h \| x'_l$ and $y' = y'_h \| y'_l$:

$$\mathsf{XT}_8(x', y') = \mathsf{XT}_4^2(x'_h, y'_h) \| \mathsf{XT}_4^1(x'_l, y'_l)$$

Then given $P(x)$ and $P(y)$, one can compute $P(x \oplus y)$ as:

$$\mathsf{XT}_8(P(x), P(y)) = P(x \oplus y) \tag{1}$$

The AddRoundKey operation can then be implemented as:

$$s'_{i,j} \leftarrow \mathsf{XT}_8(s'_{i,j}, k'_{i,j})$$

where $s'_{i,j} = P(s_{i,j})$ and $k'_{i,j} = P(k_{i,j})$. It is therefore necessary to use the permuted representation for the round keys. We further describe how this is done by modifying the key-schedule operations (see sections 3.9, 3.10 and 3.11).

3.3 SubBytes

Let S be the AES SBOX. We define the following randomised permutation S':

$$S'(x) = P(S(P^{-1}(x)))$$

Given $P(x)$, the permuted representation of $S(x)$ is then computed as:

$$P(S(x)) = S'(P(x))$$

The SubBytes operation on the permuted state variables can then be computed using the table S' as follows:

$$s'_{i,j} \leftarrow S'(s'_{i,j})$$

The randomised table $S'(x)$ requires 256 bytes in RAM.

3.4 ShiftRows

No modification is required.

3.5 MixColumns

Using $\mathsf{03} \cdot x = (\mathsf{02} \cdot x) \oplus x$, the MixColumns operation can be written as follows (first line):

$$s'_{0,c} \leftarrow (\mathsf{02} \cdot s_{0,c}) \oplus (\mathsf{02} \cdot s_{1,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus s_{3,c}$$

Therefore, we must be able to compute $P(\mathsf{02} \cdot x)$ from $P(x)$. For any $x = x_h \| x_l \in \{0,1\}^8$, from $x = (0\|x_l) \oplus (x_h\|0)$ we have:

$$\mathsf{02} \cdot x = (\mathsf{02} \cdot (0\|x_l)) \oplus (\mathsf{02} \cdot (x_h\|0))$$

and using equation (1), we obtain:

$$P(\mathsf{02} \cdot x) = \mathsf{XT}_8\big(P(\mathsf{02} \cdot (0\|x_l)), P(\mathsf{02} \cdot (x_h\|0))\big) \tag{2}$$

Therefore, for all $x' \in \{0,1\}^8$ where $x' = x'_h \| x'_l$, we define the following two tables (4-bit to 8-bit):

$$\mathsf{ML}(x'_l) := P\Big(\mathsf{02} \cdot \big(\quad 0 \quad \| \; p_1^{-1}(x'_l)\big)\Big)$$
$$\mathsf{MH}(x'_h) := P\Big(\mathsf{02} \cdot \big(\; p_2^{-1}(x'_h) \; \| \quad 0 \quad\big)\Big)$$

which gives for any $x_l, x_h \in \{0,1\}^4$:

$$\begin{aligned} \mathsf{ML}(p_1(x_l)) &= P\big(\mathsf{02} \cdot (\ 0\ \|\ x_l\ )\big) \\ \mathsf{MH}(p_2(x_h)) &= P\big(\mathsf{02} \cdot (\ x_h\ \|\ 0\ )\big) \end{aligned} \tag{3}$$

Using equations (2) and (3), given $P(x) = p_2(x_h)\|p_1(x_l)$, we can therefore compute $P(\mathsf{02} \cdot x)$ as follows:

$$P(\mathsf{02} \cdot x) = \mathsf{XT}_8(\mathsf{ML}(p_1(x_l)), \mathsf{MH}(p_2(x_h)))$$

For simplicity, given $P(x) = x'_h\|x'_l$, we denote:

$$D_2(x'_h\|x'_l) = \mathsf{XT}_8\big(\mathsf{ML}(x'_l),\ \mathsf{MH}(x'_h)\big)$$

so that for any $x \in \{0,1\}^8$:

$$P(\mathsf{02} \cdot x) = D_2(P(x)) \tag{4}$$

The first line of the MixColumns operation with permuted data can therefore be written as:

$$s''_{0,c} \leftarrow \mathsf{XT}_8\left(D_2(s'_{0,c}), \mathsf{XT}_8\left(D_2(s'_{1,c}), \mathsf{XT}_8\left(s'_{1,c}, \mathsf{XT}_8(s'_{2,c}, s'_{3,c})\right)\right)\right)$$

and the other three lines are implemented analogously. The storage of the two ML and MH tables requires 32 bits of RAM.

3.6 InvShiftRows

The algorithm remains the same.

3.7 InvSubBytes

This is similar to the SubBytes algorithm: we define the following randomised permutation S'^{-1}:

$$S'^{-1}(x) = P(S^{-1}(P^{-1}(x)))$$

Therefore, the InvSubBytes operation on the permuted state variables is computed using table $S'^{-1}(x)$ as follows:

$$s'_{i,j} \leftarrow S'^{-1}(s'_{i,j})$$

Note that one can generate the randomised table $S'^{-1}(x)$ from $S(x)$ only, so that it is not necessary to store $S^{-1}(x)$ in ROM, using the fact that:

$$S'^{-1} = P \circ S^{-1} \circ P^{-1} = (P \circ S \circ P^{-1})^{-1}$$

3.8 InvMixColumns

The first line of the InvMixColumns operation is as follows:

$$s'_{0,c} \leftarrow (\mathtt{0e} \cdot s_{0,c}) \oplus (\mathtt{0b} \cdot s_{1,c}) \oplus (\mathtt{0d} \cdot s_{2,c}) \oplus (\mathtt{09} \cdot s_{3,c})$$

We have the following relations in $\mathsf{GF}(2^8)$:

$$\mathtt{0b} = \mathtt{09} \oplus \mathtt{02}, \quad \mathtt{0d} = \mathtt{0c} \oplus \mathtt{01}, \quad \mathtt{0e} = \mathtt{0c} \oplus \mathtt{02} \tag{5}$$

Therefore only two tables for computing the multiplication by 09 and 0c are required, from which multiplication by 0b, 0d and 0e can be computed without additional tables. More precisely, for all $x \in \{0,1\}^4$, we build the following 4-bit to 8-bit tables:

$$\begin{aligned}
\mathsf{TL}(x'_l) &:= P\Big(\mathtt{09} \cdot \big(\quad 0 \quad \| \; p_1^{-1}(x'_l)\big)\Big)\\
\mathsf{TH}(x'_h) &:= P\Big(\mathtt{09} \cdot \big(\; p_2^{-1}(x'_h) \| \quad 0 \quad\big)\Big)\\
\mathsf{RL}(x'_l) &:= P\Big(\mathtt{0c} \cdot \big(\quad 0 \quad \| \; p_1^{-1}(x'_l)\big)\Big)\\
\mathsf{RH}(x'_h) &:= P\Big(\mathtt{0c} \cdot \big(\; p_2^{-1}(x'_h) \| \quad 0 \quad\big)\Big)
\end{aligned}$$

Storing those four tables requires 64 bytes in RAM. Then, as in section 3.5, writing $x' = P(x) = x'_h \| x'_l = p_2(x_h) \| p_1(x_l)$, we obtain:

$$\begin{aligned}
P(\mathtt{09} \cdot x) &= \mathsf{XT}_8\big(\mathsf{TH}(x'_h),\ \mathsf{TL}(x'_l)\big)\\
P(\mathtt{0c} \cdot x) &= \mathsf{XT}_8\big(\mathsf{RH}(x'_h),\ \mathsf{RL}(x'_l)\big)
\end{aligned}$$

As previously we denote:

$$\begin{aligned}
D_9(x'_h \| x'_l) &= \mathsf{XT}_8\big(\mathsf{TH}(x'_h),\ \mathsf{TL}(x'_l)\big)\\
D_c(x'_h \| x'_l) &= \mathsf{XT}_8\big(\mathsf{RH}(x'_h),\ \mathsf{RL}(x'_l)\big)
\end{aligned}$$

Using equations (5), we also denote:

$$\begin{aligned}
D_b(x) &= \mathsf{XT}_8(D_9(x), D_2(x))\\
D_d(x) &= \mathsf{XT}_8(D_c(x), x)\\
D_e(x) &= \mathsf{XT}_8(D_c(x), D_2(x))
\end{aligned}$$

The first line of the InvMixColumns operation can then be rewritten as follows:

$$s''_{0,c} \leftarrow \mathsf{XT}_8\left(D_e(s'_{0,c}), \mathsf{XT}_8\left(D_b(s'_{1,c}), \mathsf{XT}_8\left(D_d(s'_{2,c}), D_9(s'_{3,c})\right)\right)\right)$$

and the other three lines are rewritten analogously.

3.9 SubWord

The SubWord operation on the modified state variables is implemented like the SubByte operation.

3.10 RotWord

The RotWord operation remains unmodified.

3.11 Xor with Rcon

Let

$$R(i) = \mathtt{02}^{i-1}$$

for all $1 \leq i \leq 10$. We have $R(0) = \mathtt{01}$ and $R(i) = \mathtt{02} \cdot R(i-1)$ for all $1 \leq i \leq 10$. Therefore, letting $R'(i) := P(R(i))$, we have:

$$R'(i) = P(R(i)) = P(\mathtt{02} \cdot R(i-1)) = D_2(R(i-1))$$

Therefore the Rcon constant can be computed using the function $D_2(x)$ defined in section 3.5.

4 Security

In this section we show that our countermeasure in resistant against first-order DPA. This is due to the following lemma:

Lemma 1. *For a fixed key and input message, every intermediate byte that is computed in the course of the randomised AES algorithm has the uniform distribution in* $\{0,1\}^8$.

Proof. The proof follows directly from the fact that any intermediate AES data x is represented as $P(x)$, where $P(x_h \| x_l) = p_2(x_h) \| p_1(x_l)$ is the randomised permutation. From the construction of $p_1,\ p_2$, this implies that $P(x)$ is randomly distributed in $\{0,1\}^8$. □

The previous lemma implies that an attacker who makes statistics about power-consumption at a single point gets only random values; hence, the countermeasure is resistant against first-order DPA (like the random masking countermeasure). In order to obtain valuable information about the key, the attacker must correlate the power consumption at multiple points during the execution of the algorithm; this is called a High Order Differential Power Analysis (HO-DPA); such attack usually requires a much larger number of power consumption curves, which makes it infeasible in practice if the number of executions is limited (for example, by using a counter).

5 A Compression Scheme

A very nice compression scheme for SBOX tables has been proposed in [11]. This compression scheme works for SBOXes with a random mask; we recall it in this

section.[1] Then in Section 6 we show how to adapt it to our permutation table countermeasure.

Let $S(x)$ for $x \in \{0,1\}^8$ be a 8-bit to 8-bit SBOX. One defines $S_1(x)$ and $S_2(x)$ such that $S(x) = S_2(x)\|S_1(x)$ for all $x \in \{0,1\}^8$, where $S_1(x)$ and $S_2(x)$ are 4-bit values. Let $r_1, r_2 \in \{0,1\}^8$ and $s \in \{0,1\}^4$ be random masks, and let define the randomised table:

$$T(x) = S_1(x \oplus r_1) \oplus S_2(x \oplus r_2) \oplus s \tag{6}$$

which is a 8-bit to 4-bit table. Let $x' = x \oplus (r_1 \oplus r_2)$ be a masked data; we have from (6):

$$S_1(x) = S_1(x' \oplus r_1 \oplus r_2) = T(x' \oplus r_2) \oplus S_2(x') \oplus s$$
$$S_2(x) = S_2(x' \oplus r_1 \oplus r_2) = T(x' \oplus r_1) \oplus S_1(x') \oplus s$$

which gives:

$$S_1(x) \oplus s = T(x' \oplus r_2) \oplus S_2(x') \tag{7}$$
$$S_2(x) \oplus s = T(x' \oplus r_1) \oplus S_1(x') \tag{8}$$

Therefore given the masked data x' one can obtain a masked $S_1(x)$ and a masked $S_2(x)$, by using the randomised table T. The advantage is that the size of T is only half the size of a classically randomised SBOX table; here the storage of T requires only 128 bytes of RAM instead of 256 bytes for a classically randomised AES SBOX. More precisely, the algorithm is as follows:

InitTable(r_1, r_2, s)
1. Write $S(x) = S_2(x)\|S_1(x)$
2. Generate randomised table $T(x) = S_1(x \oplus r_1) \oplus S_2(x \oplus r_2) \oplus s$ for all $x \in \{0,1\}^8$

TableLookUp(x', r_1, r_2, s)
1. Generate a random $t \in \{0,1\}^4$
2. $u \leftarrow T(x' \oplus r_2) \oplus S_2(x')$
3. $v \leftarrow T(x' \oplus r_1) \oplus S_1(x') \oplus t$
4. Output $y = v\|u \in \{0,1\}^8$ and $r' = (s \oplus t)\|s$.

Here we add an additional masking step with t so that the values u and v are not masked with the same nibble s; from equations (7) and (8), we obtain that the value returned by TableLookUp is such that $y = S(x) \oplus r'$.

It is easy to see that this countermeasure is resistant against first-order DPA, as all the variables that appear in the TableLookUp function are uniformly distributed. Note that the tables S_1 and S_2 are only stored in ROM; they don't need to be randomised because in the TableLookUp algorithm those tables are accessed at point x' which is already randomised.

[1] In [11] the countermeasure is described in plain English which makes it difficult to understand; here we attempt to provide a more clear description.

It is also easy to see that the countermeasure can be generalised to any SBOX input and output size. Moreover, one can obtain a better compression factor by splitting $S(x)$ into more shares; for example, a 8-bit SBOX could be split into 8 tables (one for each output bit); then the resulting randomised T table would be 8 times smaller, at the cost of increasing the running time for every table look-up.

6 Time memory Trade-offs

In this section we describe three time-memory trade-offs. The goal is to reduce the RAM requirement of the permutation table countermeasure described in Section 3, for implementation on low-cost devices, at the cost of increasing the running time. The main time-memory tradeoff is based on the SBOX compression scheme recalled in the previous section. The second idea consists in using a single XOR table $\mathsf{XT}_4^1(x', y')$ instead of two XOR tables $\mathsf{XT}_4^1(x', y')$ and $\mathsf{XT}_4^2(x', y')$ as in Section 3.2. The third idea consists in removing tables TL, TH, RL and RH, by using a "Double-And-Add" approach. In Section 7 we describe the results of practical implementations of these time-memory trade-offs.

6.1 Compressed SBOX

The compression scheme of [11] recalled in the previous section was used for random masking; here we show how to adapt this compression scheme to our permutation table countermeasure.

We define a new permutation $P'(x) = p'_2(x_h) \| p'_1(x_l)$ where p'_1 and p'_2 are 4-bit permutations tables which are generated like p_1 and p_2. As previously, we write:

$$S(x) = S_2(x) \| S_1(x)$$

where $S_1(x)$ and $S_2(x)$ are 4-bit nibbles. We then define a randomised table:

$$T(x') = p_1\left(S_1\left(P^{-1}(x')\right)\right) \oplus p_2\left(S_2\left(P^{-1}\left(P'^{-1}(x')\right)\right)\right) \tag{9}$$

The randomised table $T(x')$ is a 8-bit to 4-bit table; therefore, its storage requires 128 bits in memory, instead of 256 bytes for the randomised table $S'(x')$ in Section 3.3. Writing $x' = P(x)$, we obtain from equation (9):

$$p_1(S_1(x)) = T(P(x)) \oplus p_2\left(S_2\left(P^{-1}\left(P'^{-1}\left(P(x)\right)\right)\right)\right)$$

This shows that given $P(x)$ we can compute $p_1(S_1(x))$ using randomised table T and table S_2 stored in ROM. Similarly, writing $x' = P'(P(x))$, we obtain from equation (9):

$$p_2(S_2(x)) = T\left(P'(P(x))\right) \oplus p_1\left(S_1\left(P^{-1}\left(P'\left(P(x)\right)\right)\right)\right)$$

This shows that given $P(x)$ we can compute $p_2(S_2(x))$ using randomised table T and table S_1 stored in ROM. Therefore, given $P(x)$ we can compute:

$$P(S(x)) = p_2\left(S_2(x)\right) \| p_1\left(S_1(x)\right)$$

using the following operations:

$\mathsf{InitTable}(P, P')$:

1. Write $S(x) = S_2(x) \| S_1(x)$
2. Generate table $T(x') = p_1\left(S_1\left(P^{-1}(x')\right)\right) \oplus p_2\left(S_2\left(P^{-1}\left(P'^{-1}(x')\right)\right)\right)$ for all $x' \in \{0,1\}^8$.

$\mathsf{TableLookUp}(x', T, P, P')$

1. $u \leftarrow T(x') \oplus p_2\left(S_2\left(P^{-1}\left(P'^{-1}\left(x'\right)\right)\right)\right)$
2. $v \leftarrow T\left(P'(x')\right) \oplus p_1\left(S_1\left(P^{-1}\left(P'\left(x'\right)\right)\right)\right)$
3. Output $v \| u$

Given the tables P, P', T and denoting $F(x') = \mathsf{TableLookUp}(x', T, P, P')$, the $\mathsf{SubBytes}$ operation on the permuted state variables is computed as follows:

$$s'_{i,j} \leftarrow F(s'_{i,j})$$

The table T requires 128 bytes in memory, and the additional permutations P' and P'^{-1} require 32 bytes in memory, so in total 160 bytes are required (instead of 256 bytes for the randomised table S'). The $\mathsf{InvSubBytes}$ operation on the permuted state variables with compressed inverse SBOX S^{-1} is obtained by replacing S by S^{-1} in the previous equations.

6.2 Single XOR table

In Section 3.2 two 8-bit to 4-bit xor-tables are defined. In this section, we show that it is sufficient to define only one 8-bit to 4-bit xor-table. As in Section 3.2, we define:

$$\mathsf{XT}_4^1(x', y') := p_1(p_1^{-1}(x') \oplus p_1^{-1}(y'))$$

We also define the 4-bit to 4-bit permutations:

$$p_{12}(x) = p_1(p_2^{-1}(x)) \tag{10}$$
$$p_{21}(x) = p_2(p_1^{-1}(x)) \tag{11}$$

and we store those two permutation tables in RAM. Then we can define the function:

$$\mathsf{XT}_4^2(x', y') := p_{21}(\mathsf{XT}_4^1(p_{12}(x'), p_{12}(y')))$$

From equations (10) and (11) we obtain that for all $x_h, y_h \in \{0,1\}^4$:

$$\mathsf{XT}_4^2(p_2(x_h), p_2(y_h)) = p_{21}(\mathsf{XT}_4^1(p_1(x_h), p_1(y_h)) = p_{21}(p_1(x_h \oplus y_h)) = p_2(x_h \oplus y_h)$$

Therefore, the XT_4^2 function takes the same values as the XT_4^2 table defined in Section 3.2. The advantage is that only 16 bytes of RAM are necessary to store the permutation tables p_{12} and p_{21} instead of 128 bytes for the previous XT_4^2 table.

6.3 Double and Add for InvMixColumns

The first line of the InvMixColumns operation is as follows:

$$s'_{0,c} \leftarrow (\mathtt{0e} \cdot s_{0,c}) \oplus (\mathtt{0b} \cdot s_{1,c}) \oplus (\mathtt{0d} \cdot s_{2,c}) \oplus (\mathtt{09} \cdot s_{3,c})$$

In this section we show how to avoid the four tables TL, TH, RL and RH, using a Double and Add method. More precisely, using the existing D_2 function (see equation (4)) we define the following functions:

$$\begin{aligned}
D_4(x') &:= D_2(D_2(x')) \\
D_8(x') &:= D_2(D_4(x')) \\
D_9(x') &:= \mathsf{XT_8}(D_8(x'), x') \\
D_b(x') &:= \mathsf{XT_8}(D_9(x'), D_2(x')) \\
D_c(x') &:= \mathsf{XT_8}(D_8(x'), D_4(x')) \\
D_d(x') &:= \mathsf{XT_8}(D_c(x'), x') \\
D_e(x') &:= \mathsf{XT_8}(D_c(x'), D_2(x'))
\end{aligned}$$

Therefore no additional table is required beyond the already defined ML and MH tables. The first line of the InvMixColumns operation can then be rewritten as follows:

$$s''_{0,c} \leftarrow \mathsf{XT_8}\left(D_e(s'_{0,c}), \mathsf{XT_8}\left(D_b(s'_{1,c}), \mathsf{XT_8}\left(D_d(s'_{2,c}), D_9(s'_{3,c})\right)\right)\right)$$

and the other three lines are rewritten analogously.

6.4 Security

As for our basic permutation table countermeasure, all the intermediate variables that appear in the time-memory tradeoff variants have the uniform distribution:

Lemma 2. *For a fixed key and input message, every intermediate byte that is computed in the course of the randomised AES algorithm with any of the three time-memory trade-offs has the uniform distribution in* $\{0,1\}^8$.

Therefore, the three previous time-memory tradeoffs are secure against first-order DPA.

7 Implementation

We summarise in Table 1 the timings observed and RAM needed for each AES operation. The timings are based on an implementation in C on a 2 GHz laptop under Linux. The RAM requirements are based on Tables 2 and 3 in Appendix. These timings show that the new countermeasure is reasonably efficient, as it is only roughly four times slower than AES with masking countermeasure, for a comparable amount of memory. We note that the time-memory tradeoffs enable to spare 272 bytes in RAM compared to our basic permutation table countermeasure.

Countermeasure	Timing (μs)	RAM (bytes)
AES encryption without countermeasure	2.2	214
AES encryption with masking	4.0	474
AES encryption with basic permutation tables	11	778
AES encryption with permutation tables + trade-offs	27	570
AES decryption without countermeasure	4.0	214
AES decryption with masking	9.5	474
AES decryption with basic permutation tables	23	842
AES decryption with permutation tables + trade-offs	42	570

Table 1. Timings obtained using a C implementation of AES without countermeasure, with masking countermeasure, and with proposed countermeasure: basic permutation table, and with time-memory trade-offs, on a 2 GHz laptop under Linux

8 Conclusion

We have described a new countermeasure against DPA for AES, that does not use random masking. Our countermeasure is provably secure against first order DPA, and reasonably efficient compared to the classical random masking countermeasure. We have also described three time-memory tradeoffs to reduce the RAM requirement; this can be useful for smart-card implementations.

Acknowledgments: Thanks to Christophe Clavier for suggesting the "Double and Add" approach in Section 6.3.

References

1. M. L. Akkar and Christophe Giraud, *An Implementation of DES and AES, Secure against Some Attacks*, proceedings of CHES 2001. Springer-Verlag.
2. S. Chari, C. S. Jutla, J. R. Rao and P. Rohatgi, *Towards Sound Approaches to Counteract Power-Analysis Attacks*, Proceedings of Crypto'99.
3. J.S. Coron, *Resistance against Differential Power Analysis for Elliptic Curve Cryptosystems*, Proceedings of CHES 1999, pp. 292-302.
4. J. Daemen and V. Rijmen, *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer-Verlag, 2002. ISBN 3540425802.
5. J. D. Golic and C. Tymen, *Multiplicative Masking and Power Analysis of AES*, Proceedings of CHES 2002.
6. P. Kocher, J. Jaffe and B. Jun, *Differential Power Analysis*, Proceedings of CRYPTO '99, LNCS.
7. P. Kocher et al, DES and Other Cryptographic processes with Leak Minimization for smartcards and other cryptosystems. US 6,278,783 B1, Jun. 3, 1999. Available at: `http://www.cryptography.com/technology/dpa/licensing.html`.
8. IBM Corporation, Space-efficient, side-channel attack resistant table lookups, Application Patent 20030044003.
9. E. Oswald, S. Mangard, N. Pramstaller and V. Rijmen, *A Side-Channel Analysis Resistant Description of the AES S-box*, Proceedings of FSE 2005, LNCS 3557, pp. 413-423.

10. E. Oswald and K. Schramm, *An Efficient Masking Scheme for AES Software Implementations*. Proceedings of WISA 2005, LNCS 3786, Springer, pp. 292-305, 2006
11. J.R. Rao, P. Rohatgi, H. Scherzer and S. Tinguely, *Partitioning attacks: Or How to rapidly Clone Some GSM Cards*, Proceedings of the 2002 IEEE Symposium on Security and Privacy, 2002.
12. J. Wolkerstorfer, E. Oswald and M. Lamberger, *An ASIC implementation of the AES SBoxes*. Proceedings of CT-RSA 2002. Springer-Verlag.

A AES pseudo-code

```
Cipher(byte in[16], byte out[16], word w[44])
begin
  byte state[16]
  state=in
  AddRoundKey(state,w[0,3])
  for round=1 to 9
     SubBytes(state)
     ShiftRows(state)
     MixColumns(state)
     AddRoundKey(state,w[round*4,(round+1)*4-1])
  end for
  SubBytes(state)
  ShiftRows(state)
  AddRoundKey(state,w[40,43])
end
```

Fig. 1. Pseudo-code for AES encryption

B List of Tables

We summarise in Table 2 the list of randomised tables required for each operation. Note that for decryption, we also need table S' to compute the key-schedule, but this table can be discarded at the end of the key-schedule. The RAM requirement for those randomised tables is summarised in Table 3.

```
InvCipher(byte in[16], byte out[16], word w[44])
begin
  byte state[16]
  state=in
  AddRoundKey(state,w[40,43])
  for round=9 downto 1
    InvShiftRows(state)
    InvSubBytes(state)
    AddRoundKey(state,w[round*4,(round+1)*4-1])
    InvMixColumns(state)
  end for
  InvShiftRows(state)
  InvSubBytes(state)
  AddRoundKey(state,w[0,3])
end
```

Fig. 2. Pseudo-code for AES decryption

Operation	Tables required
AES encryption, basic	P, P^{-1}, XT_4^1, XT_4^2, S', ML, MH
AES decryption, basic	P, P^{-1}, XT_4^1, XT_4^2, S'^{-1}, $\mathsf{ML}, \mathsf{MH}, \mathsf{RL}, \mathsf{RH}, \mathsf{TL}, \mathsf{TH}$
AES encryption, tradeoffs	P, P^{-1}, XT_4^1, ML, MH, p_{12}, p_{21}, P', P'^{-1}, T
AES decryption, tradeoffs	P, P^{-1}, XT_4^1, ML, MH, p_{12}, p_{21}, P', P'^{-1}, T

Table 2. Randomised tables required for each AES operation, for the basic permutation table countermeasure, and with the three time-memory tradeoffs.

Operation	RAM (bytes)
Permutations P and P^{-1}	32
Xor-table XT_4^1	128
Xor-table XT_4^2	128
Randomised SBOX S'	256
Randomised SBOX S'^{-1}	256
Tables ML, MH	32
Tables TL, TH, RL, RH	64
Permutations p_{12} and p_{21}	16
Permutations P' and P'^{-1}	32
Table T	128

Table 3. Memory requirement for the randomised tables