Easy and Fast Key-dependent Affine Transformation of Square S-boxes

Marcel Martin
m.martin@ellipsa.eu

October 24, 2020

(Draft 2)

Abstract

Substitution boxes (S-boxes) are generally the only non-linear part of a block cipher. Using key-dependent S-boxes rather than static ones might increase the security of a block cipher but it would take too much time to build a good key-dependent S-box from scratch just before ciphering or deciphering. What we can do is to transform an existing S-box, assuming

1) the relevant cryptographic properties of the S-box are preserved;
2) the execution of the transformation is sufficiently fast.

Keywords: cryptography, block cipher, S-box, affine transformation.

1 Introduction

First, a few words to justify the title: the transformation is « easy » because its implementation does not require particular math resources (no binary matrix library, for instance) and it is « fast » because transforming an existing S-box takes only a few microseconds.

Notation used

$\mathbb{Z}_{2^n}$ The set of the unsigned integers modulo 2^n, $n > 0$.

$\mathbb{Z}_2^n$ The set of the column vectors $\overline{v} = {}^t(v_0, v_1, ..., v_{n-1})$ having n components in $\mathbb{Z}_2$, $n > 0$.

$\mathcal{V}(k)$ Vector $\overline{k}$ of $\mathbb{Z}_2^n$ equivalent to the integer k of $\mathbb{Z}_{2^n}$ [1].

$\mathcal{I}(\overline{v})$ Integer v of $\mathbb{Z}_{2^n}$ equivalent to the vector $\overline{v}$ of $\mathbb{Z}_2^n$ [2].

$\oplus$ The **xor** operator. Bitwise exclusive **or** over $\mathbb{Z}_{2^n}$ as well as over $\mathbb{Z}_2^n$ (over $\mathbb{Z}_2^n$, the $\oplus$ and $+$ operators are equivalent).

$\mathbb{M}_2^n$ The set of the $n \times n$ matrices having their coefficients in $\mathbb{Z}_2$, $n > 0$.

$*$ The multiplication operator of a matrix of $\mathbb{M}_2^n$ by a vector of $\mathbb{Z}_2^n$.
If $\overline{u} = M * \overline{v}$ then, $\forall i \in 0..n-1$, $u_i = \bigoplus_{0 \le k < n} (M_{i,k} \times v_k)$.

A (n, n) S-box is a mapping $S : \mathbb{Z}_2^n \to \mathbb{Z}_2^n$ where n is a fixed positive integer, $n \geq 2$. Such a S-box can be represented as an array of 2^n integers of $\mathbb{Z}_{2^n}$ [9].

Two (n, n) S-boxes S and T are said *affine equivalent* if $T = B \circ S \circ A$ where A and B are affine permutations. With affine permutations, many cryptographic properties of the transformed S-boxes have the same values than the ones of the initial S-boxes. Among others, the balancedness, the algebraic degree, the minimum degree, the non-linearity, the differential uniformity, the algebraic immunity, and the global avalanche characteristics [11] are identical [3, 4, 5]. The graph algebraic immunity indicators (degree and number of independent equations) [7] are also preserved [3].

From a computational point of view, in order to perform the affine transformation of a S-box, only two permutations (look-up tables) P and Q are needed. As a matter of fact, the S-box being coded as an array of 2^n integers of $\mathbb{Z}_{2^n}$, the proposed method simply consists in permuting the 2^n indices of this array according to P and, then, in permuting the 2^n values according to Q.

In what follows, most algorithms (more exactly, Free Pascal functions) are specialized for $(8, 8)$ S-boxes but it is not difficult to translate them in an other language and/or to adapt them so that they work with other dimensions.

For the Pascal code, the global type used is

```
type TArrayOfByte = array [Byte] of Byte; // array of 256 bytes
```

It is assumed that the *NextByte()* function, using a part of the secret key of a block cipher as a seed, behaves like a random generator having a uniform distribution on $\mathbb{Z}_{2^8}$.

[1] There is a bijective mapping between the integers of $\mathbb{Z}_{2^n}$ and the vectors of $\mathbb{Z}_2^n$ defined by the correspondence $(b_0 + 2b_1 + ... + 2^{n-1}b_{n-1}) \in \mathbb{Z}_{2^n} \longleftrightarrow {}^t(b_0, b_1, ..., b_{n-1}) \in \mathbb{Z}_2^n$.

[2] See the previous note.

[3] The graph algebraic immunity is invariant under CCZ-equivalence [4] and the affine equivalence of S-boxes implies their CCZ-equivalence [8, *Slide 53*].

2 Building of a permutation

Let M be an invertible matrix of $\mathbb{M}_2^8$ and let a be an integer of $\mathbb{Z}_{2^8}$. It is easy to build a permutation P of the $\mathbb{Z}_{2^8}$ integers such that, $\forall k \in \mathbb{Z}_{2^8}$, $P[k] = \mathcal{I}(M * \overline{k}) \oplus a$.

Let us first consider the case $a = 0$. $\overline{C_i}$ being the column vector #i of M, if we build P such that

$P[2^i] \quad \leftarrow \quad C_i \qquad$ (for i in 0..7),
$P[i \oplus j] \quad \leftarrow \quad P[i] \oplus P[j] \quad$ (for all the other cells of P),

then, $\forall k \in \mathbb{Z}_{2^8}$, $P[k] = \mathcal{I}(M * \overline{k})$. *(The proof is not difficult.)*

For the case $a \neq 0$, it is sufficient to xor with a all the values of the previously obtained permutation P (but this is not the way it is coded in the following Pascal function).

```
// ---------------------------------------------------------------------------
// → a, addend. If a = 0 then Result is equivalent to a linear transformation.
// ← Result, key-dependent permutation of the ℤ2⁸ integers (based on an invertible 8×8 binary
// matrix).
// ---------------------------------------------------------------------------
function GetKeyedSwap(a: Byte): TArrayOfByte;
  var
    P          : TArrayOfByte absolute Result; // P is an alias for Result
    FreeColumn : array [Byte] of Boolean;
    i, j       : Integer;
    c, t       : Byte;
begin
  FillChar(FreeColumn,SizeOf(FreeColumn),TRUE);
  FreeColumn[a] := FALSE;
  P[0] := a;

  // columns #0..#6
  j := 1;
  repeat
    // get a new independent column
    repeat
      c := NextByte;
    until FreeColumn[c xor a]; // proba: (256 − j)/256

    // update P and mark all linear combinations of columns already used
    for i := 0 to j − 1 do
    begin
      t := P[i] xor c;
      P[i xor j] := t;
      FreeColumn[t] := FALSE;
    end;
    j := j shl 1;
  until j = 128;
```

```
  // column #7 (no more need to update FreeColumn[])
  // get a new independent column
  repeat
    c := NextByte;
  until FreeColumn[c xor a]; // proba: 1/2

  // update P (type-casting P on the fly to a DWord or a QWord array might speed things up)
  for i := 0 to 127 do
    P[i xor 128] := P[i] xor c;
end;
// ------------------------------------------------------------------------------
```

Remarks

• By construction,

– if $a = 0$, $\left\{\begin{array}{rcl} P : \mathbb{Z}_{2^8} & \to & \mathbb{Z}_{2^8} \\ i & \mapsto & P[i] \end{array}\right\}$ is an automorphism of the group $(\mathbb{Z}_{2^8}, \oplus)$;

– if $a \neq 0$, P is a coset of the automorphism obtained with $a = 0$.

• Assuming the *NextByte()* function behaves like a uniform random generator, on average, obtaining the eight independent columns requires $\sum_{0 \leq i < 8} \frac{2^8}{2^8 - 2^i} \approx 9.603$ calls to this function. So, on average, in order to make P affine, there are only about 9.603 executions of the "**xor** a" instruction (instead of 256 if a simple loop were used).

• There are $N = 2^8 \times \prod_{0 \leq i < 8} (2^8 - 2^i) = 1\,369\,104\,324\,918\,194\,995\,200$ different possible permutations. Compared to powers of 2, we have $2^{70} < N < 2^{71}$.

3 Elimination of fixed points

Suppose we intend to xor all the values of a S-box S with the value a. If $(S[i] \oplus a = i)$ then i will be a fixed point.
Now, since $(S[i] \oplus a = i) \iff (S[i] \oplus i = a)$, if we fill up a boolean array T with the value FALSE and then if we set $T[S[i] \oplus i]$ to TRUE for all i, the cells $T[j]$ remaining FALSE, if any, indicate that xoring the S-box S with j will produce a direct fixed point free S-box. And we get all the possible j's with a single iteration.

```
// ------------------------------------------------------------------------------
// ↔ S, (8,8) S-box.
// ← Result, TRUE if, and only if, the modified S is (direct and opposite) fixed point free.
// ------------------------------------------------------------------------------
function MakeFixedPointFree(var S: TArrayOfByte): Boolean;
  var
    FixedPoint : array [Byte] of Boolean;
    i, j       : Byte;
```

```
begin
  FillChar(FixedPoint,SizeOf(FixedPoint),FALSE);
  for i := 0 to 255 do
  begin
    j := S[i] xor i;
    FixedPoint[j] := TRUE; // direct fixed point
    FixedPoint[not j] := TRUE; // opposite fixed point: S[i] xor (not i) = not (S[i] xor i)
  end;

  i := NextByte; // key-dependent initial index
  j := i;
  while FixedPoint[j] do
  begin
    j := (j + 1) and $FF; // since j is a byte, a simple "Inc(j);" would be OK
    if j = i then
      Exit(FALSE); // no solution
  end;

  // modify S (type-casting S on the fly to a DWord or a QWord array might speed things up)
  for i := 0 to 255 do
    S[i] := S[i] xor j; // useless whenever j = 0

  Result := TRUE;
end;
// ---------------------------------------------------------------------------
```

Remarks

• The final $FixedPoint$ array is palindromic since, with any byte j, **not** $j = 255 - j$. Therefore its size could be divided by 2 but all what I have tried led to a slower code.

• Though it is theoretically possible that *MakeFixedPointFree()* returns FALSE, it would seem it rarely occurs with S-boxes having good cryptographic properties [4]. (As a matter of fact, during the numerous tests I made, it never occurred.)

4 Affine transformation of a S-Box

The affine transformation $T = B \circ S \circ A$, where S and T are S-boxes and A and B are affine permutations, can be computed with $\mathbb{M}_2^n$ matrices and additions of $\mathbb{Z}_2^n$ vectors (see [10, *Slide 6*]). Using the permutations produced by $GetKeyedSwap()$, we can get the S-box T with

$$T[i] \leftarrow P_B[S[P_A[i]]], \forall i \in 0..2^n - 1.$$

Notice that, since $(T = B \circ S \circ A) \iff (T \circ A^{-1} = B \circ S)$, we could also build the S-box T with

$$T[P_A^{-1}[i]] \leftarrow P_B[S[i]], \forall i \in 0..2^n - 1.$$

[4] With $(8, 8)$ S-boxes: Minimum degree $= 7$, non-linearity ≥ 104 and differential uniformity ≤ 8.

```
// ---------------------------------------------------------------------------
// → S, (8,8) S-box.
// ← Result, key dependent (8,8) S-box. Result is (direct and opposite) fixed point free and it
// is affine equivalent to S.
// ---------------------------------------------------------------------------
function GetKeyedSBox(const S: TArrayOfByte): TArrayOfByte;
  var
    T    : TArrayOfByte absolute Result; // T is an alias for Result
    P, Q : TArrayOfByte;
    i    : Integer;
begin
  repeat
    P := GetKeyedSwap(NextByte);
    Q := GetKeyedSwap(0); // MakeFixedPointFree(T) will make Q affine
    for i := 0 to 255 do
      T[i] := Q[S[P[i]]];
  until MakeFixedPointFree(T);
end;
// ---------------------------------------------------------------------------
```

Remark

Due to the symmetry properties of S (if any), even if P and Q are different from the Identity permutation, T might be equal to S (see [2, §4.1, *Self-Equivalent S-boxes*]).

5 Examples of use

• Example 1

Table 1: Original AES S-box [6]

63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Some properties of the AES S-box:

- Balanced = TRUE
- Minimum degree = 7
- Non-linearity = 112
- Differential uniformity = 4
- Global avalanche characteristics:
  - Absolute indicator = 32
  - Sum-of-Squares indicator = 133120
- Graph algebraic immunity:
  - Degree = 2
  - Independent equations = 39
- Fixed points:
  - Direct = 0
  - Opposite = 0

Code used in order to make things easily reproducible:

```
var Seed : Byte;

function NextByte: Byte; inline;
begin
  Seed := Seed*5 + 131;
  Result := Seed;
end;

Seed := 1;
S := GetKeyedSBox(AES); // AES: array of 256 bytes
```

Table 2: S-box S returned by GetKeyedSBox(AES);

a6	25	43	31	d2	cc	ad	68	36	9f	eb	5a	6b	3c	22	c8
e9	17	df	11	67	1b	5c	08	8d	e0	53	99	d8	24	23	97
ae	59	89	ef	7d	bd	fe	a5	b2	2d	f3	cd	ca	5f	88	a7
ab	f5	6f	3d	ac	b0	7c	21	42	e6	0a	dc	84	f2	3f	7e
16	26	f7	27	93	ea	ce	63	1a	51	b3	70	8c	f4	83	9a
bc	38	e5	87	69	9c	c7	e2	f6	b6	35	be	aa	98	33	a4
18	54	a0	1e	f9	2e	4b	c4	b7	65	d3	3b	78	b9	05	95
ec	c5	1f	40	bb	15	b8	6d	9d	3e	72	61	45	0d	55	9e
c1	73	94	85	77	10	20	01	92	4f	c0	58	c3	a2	d1	c9
a9	71	56	12	a3	0f	ba	8e	64	04	4d	52	fb	8b	4c	a8
34	14	0c	0e	02	2c	db	30	13	b1	74	00	7b	76	d4	7f
1c	fc	e1	3a	44	4e	e4	9b	8a	03	91	48	e8	ff	62	06
07	86	49	4a	d5	dd	b4	af	5b	28	80	5e	60	f1	bf	e7
2b	f8	81	8f	da	de	82	ee	fd	d6	47	1d	57	6e	d0	e3
d7	c2	39	32	19	f0	cf	fa	37	ed	29	75	2f	6c	46	a1
2a	90	50	6a	7a	41	d9	09	66	c6	cb	b5	96	79	0b	5d

All the previously listed properties have the same values for the modified S-Box than for the AES S-box.
The average running time of the $GetKeyedSBox()$ function, obtained with Free Pascal 3.0.4 on a Intel i7-2600 processor (3.4 GHz), is equal to 3.3 µs.

• Example 2

In [1], the authors propose a method based on two small permutations to transform a S-box. Their method being a linear transformation, the integer permutation trick proposed here can be used to do the job.

In order to get the same results than the ones of their examples 3.5 & 3.7, we only need two functions: $GetSwap()$ and $GetSBox()$.

The global type used is

```
type TDynArrayOfByte = array of Byte; // dynamic array of bytes
```

```
// ---------------------------------------------------------------------------
// → p, permutation of 0..n − 1, n ≤ 8. p is regarded as the representation of a n×n permutation
// matrix M such that M_{p[i],i} = 1.
// ← Result, permutation of 0..2^n − 1 equivalent to M, i.e., to p.
// ---------------------------------------------------------------------------
function GetSwap(const p: array of Byte): TDynArrayOfByte;
  var
    n       : Integer;
    i, j, k, c : Byte;
begin
  n := Length(p);
  SetLength(Result,Integer(1) shl n);
  Result[0] := 0;
  i := 1;
  for j := 0 to n − 1 do
  begin
    c := Byte(1) shl p[j]; // c̄ is the column vector #j of a n×n permutation matrix
    for k := 0 to i − 1 do
      Result[k xor i] := Result[k] xor c;
    i := i shl 1;
  end;
end;
// ---------------------------------------------------------------------------
```

Remarks

• The parameter of *GetSwap()* is declared as ***array of*** *Byte* (i.e., "open" array). This way the function may be called with any array type: ordinary, open, and/or dynamic.

• Let R be the permutation returned by $GetSwap()$ and let M be the $n \times n$ permutation matrix equivalent to p (such that $M_{p[i],i} = 1$). Then, $\forall k \in \mathbb{Z}_{2^n}, R[k] = \mathcal{I}(M * \overline{k})$.

For instance, with $n = 8$, $p = (1, 2, 0, 3, 5, 7, 6, 4)$ and $k = 108$, we get

$$M = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \text{ and}$$

$M * \overline{k} = M * {}^t(0,0,1,1,0,1,1,0) = {}^t(1,0,0,1,0,0,1,1) = \mathcal{V}(1+8+64+128) = \mathcal{V}(201)$.
Of course, $R[108] = 201$.

```
// ---------------------------------------------------------------------------
// → S, (n,n) S-box (array of 2^n bytes), n ≤ 8.
// → p1, permutation of 0..n-1 (array of n bytes).
// → p2, permutation of 0..n-1 (array of n bytes).
// ← Result, S-box S modified according to the p1 and p2 permutations.
// ---------------------------------------------------------------------------
function GetSBox(const S,p1,p2: array of Byte): TDynArrayOfByte;
  var
    P, Q : TDynArrayOfByte;
    i    : Byte;
begin
  P := GetSwap(p1);
  Q := GetSwap(p2);
  SetLength(Result,Length(S));
  for i := 0 to High(S) do
    Result[i] := Q[S[P[i]]];
end;
// ---------------------------------------------------------------------------
```

Remark

Except for the numbers of fixed points, the properties (listed in Example #1) of the returned S-boxes have the same values than the ones of S.

(For the two following resulting S-boxes, I made use of a decimal representation to mimic what is given in the quoted paper.)

As in the example 3.5, page 10 of [1], the $(4,4)$ S-box returned by

```
GetSBox([9,13,10,15,11,14,7,3,12,8,6,2,4,1,0,5],[1,2,0,3],[3,2,0,1]);
```

is equal to

10	6	14	13	11	15	7	12	3	5	1	0	2	4	8	9

As in the example 3.7, page 14 of [1], the $(8, 8)$ S-box returned by

GetSBox(AES,[1,2,0,3,5,7,6,4],[1,0,2,3,7,5,4,6]);

is equal to

51	183	241	63	188	187	59	86	160	55	253	107	2	43	215	181
231	195	165	247	254	37	175	92	164	118	178	162	102	242	216	134
94	131	159	20	12	124	199	135	84	189	52	138	103	174	158	179
112	169	26	36	161	9	5	156	81	108	194	116	211	49	198	186
10	44	139	153	67	137	61	96	145	213	42	47	171	227	115	68
208	105	19	163	127	251	30	70	22	1	144	207	250	191	172	104
233	38	140	228	184	45	101	85	120	180	27	75	222	143	238	73
114	200	58	77	248	130	218	196	203	71	93	40	141	122	150	223
89	90	249	23	65	190	154	240	110	97	204	177	212	111	100	80
4	35	136	6	87	83	197	201	7	64	123	225	129	113	39	182
48	31	33	192	66	220	41	72	21	232	221	11	125	132	157	219
119	167	78	29	88	62	214	106	60	244	54	109	149	121	185	8
147	0	32	226	210	126	252	155	57	237	25	152	91	170	28	95
146	16	193	168	99	79	206	246	236	217	128	243	229	34	255	209
176	230	24	245	173	53	3	13	50	151	69	142	166	234	82	205
76	74	239	17	98	14	117	56	18	46	224	235	202	15	148	133

6 Conclusion

The proposed method is very easy to implement. From a practical point of view, it is sufficiently light and fast so that it can be embedded in a block cipher.

It should be noted that affine transformations modify some cryptographic properties of a S-box. Among others:

– the number of monomials of the univariate polynomial representation;
– the transparency order;
– the branch number;
– the DPA [5] signal-to-noise ratio.

These properties may become better... or the contrary.

***

First publication October 18, 2020

[5] Differential Power Analysis.

References

[1] **A Novel Method to Generate Key-Dependent S-Boxes with Identical Algebraic Properties**
Ahmad Y. Al-Dweik, Iqtadar Hussain, Moutaz S. Saleh & M. T. Mustafa (2019)

[2] **A Toolbox for Cryptanalysis: Linear and Affine Equivalence Algorithms**
Alex Biryukov, Christophe De Cannière, Ann Braeken & Bart Preneel (Eurocrypt, 2003)

[3] **On the behaviours of affine equivalent Sboxes regarding differential and linear attacks**
Anne Canteaut & Joëlle Roué (Eurocrypt, 2015)

[4] **On the Algebraic Immunities and Higher Order Nonlinearities of Vectorial Boolean Functions**
Published in *"Enhancing Cryptographic Primitives with Techniques from Error Correcting Codes"*, pp. 104-116.
Claude Carlet (2009)

[5] **Vectorial Boolean Functions for Cryptography**
Claude Carlet (2010)

[6] **Advanced Encryption Standard - FIPS 197**
FIPS (2001)

[7] **On some methods for constructing almost optimal S-Boxes and their resilience against side-channel attacks**
Reynier Antonio de la Cruz Jiménez (2018)

[8] **Analysing S-boxes**
Gregor Leander (2012)

[9] **Practical S-box Design**
Serge Mister & Carlisle Adams (SAC, 1996)

[10] **A Search Strategy to Optimize the Affine Variant Properties of S-boxes**
Stjepan Picek, Bohan Yang & Nele Mentens (2016)

[11] **GAC - the Criterion for Global Avalanche Characteristics of Cryptographic Functions**
Xian-Mo Zhang & Yuliang Zheng (1997)