Chapter X

MASKED INVERSION IN GF(2^N) USING MIXED FIELD REPRESENTATIONS AND ITS EFFICIENT IMPLEMENTATION FOR AES

SHAY GUERON[1,2], ORI PARZANCHEVSKY[1] and OR ZUK[1,3]
[1] Discretix Technologies, Netanya, ISRAEL
[2] Department of Mathematics, University of Haifa, Haifa, 31905, ISRAEL
[3] Faculty of Physics, Weizmann Institute of Science, Rehovot, ISRAEL
shay@math.haifa.ac.il, ori.parzan@discretix.com, or.zuk@weizmann.ac.il

This paper describes an efficient method for protecting implementations of inversions in GF(2^n) against DPA attacks. The general method combines two techniques, both of which were proposed in the context of AES S-Box design: a) the simplified multiplicative mask, and b) the use of mixed field representations for the AES S-box. Here, we modify the masking procedure and make it suitable for situations where the inversion is performed in a preferred field representation that differs from the representation in which the input/output are given. For n=8 in particular, we provide the details of the mask updates that are required for the complete AES round. Our results indicate that significantly increased efficiency is gained when this method is used to construct a hardware implementation of AES, protected against DPA attacks.

X.1. INTRODUCTION

Multiplicative masking for protecting implementations of the AES S-Box against Differential Power Analysis (DPA) attacks was originally proposed in [1], and was later simplified in [9]. Interestingly, this method has a broader range of applications. For example, it can also be applied to other ciphers whose S-Box design is based on inversion in finite fields (e.g. Camellia [2] and Zodiac [5]).

Masking a data chunk, x, is achieved by using $x+r$ in the AES round, where r (the mask) is a random bit stream whose length equals to the length of x, and + denotes addition in GF(2^8) (i.e., bytewise XOR). This way, the true input (x) is never used in clear. Consequently, an attacker cannot collect statistics based on the input/output to the S-Box, which is required in order to mount a DPA attack. Neither can he control the actual data that is being processed by the circuit, even when he is able to feed it with some chosen inputs.

To eventually obtain the correct cipher output, one must be able to remove the cumulative effects of the mask on the components of the AES round. The effect on the Key-xor, the affine transformation (of the S-Box), the Shift Row and the Mix Column phases can be easily accounted for, because these parts of the AES round are affine/linear functions over $GF(2^8)$. However, it is less straightforward to account for the effect on the nonlinear step, namely on the $GF(2^8)$ inversion, which is part of the S-Box operation.

The multiplicative masking method is designed to solve the above problem. Reference [1] proposes a method that involves two random variables, $x_{i,j}$, $y_{i,j}$, where $x_{i,j}$ is an additive mask and $y_{i,j}$ is a nonzero multiplicative mask. A simplification of this method, requiring a smaller number of operations and therefore a smaller area hardware circuit, was proposed in [9]. We point out that the simplified masking is the special case of [1], using the substitution $y_{i,j} = x_{i,j}$ (it requires, additionally, that the additive mask is nonzero). The simplified multiplicative masking method tackles the problem by: applying a random mask *r*, processing xr (instead of $x + r$) during the inversion (hence the name), and then transforming $(xr)^{-1}$ to $x^{-1} + r$, in order to continue the affine/linear operations of the AES round with the original (additive) mask, *r*. The sequence of operations for the simplified mask [9] is $x + r \xrightarrow[\times r]{} xr + r^2 \xrightarrow[+r^2]{} (xr) \xrightarrow[inverse]{} (xr)^{-1} \xrightarrow[+1]{} (xr)^{-1} + 1 \xrightarrow[\times r]{} x^{-1} + r$, where the operations (multiplication, addition and inversion) are over $GF(2^8)$ with the reduction polynomial $X^8 + X^4 + X^3 + X + 1$, as AES defines [7]. Note that for this method to be successful, the random byte *r* must be nonzero.

Implementing this technique involves two multiplications, one square and two additions in $GF(2^8)$. Assuming that no time delays are allowed, a hardware implementation must include additional circuitry: one circuit for squaring in GF(2^8), and two copies of a multiplication circuit. The associated hardware overhead is significant. While computing squares in $GF(2^8)$ is cheap, the multiplication circuits are very costly (see Appendix A for details). This fact makes the proposed masking technique less attractive for low resources environments such as smartcards.

Our goal here is to derive a more efficient and more secure method for implementing such masking. The underlying idea is to replace the costly multiplications in $GF(2^8)$ by several cheaper computations in another field (e.g., $GF(2^4)$). This approach was proposed, for example in [8] (restated in [4]), for reducing the cost of the $GF(2^8)$ inversion (256 bytes lookup table), which is part of the AES S-Box. No actual implementation is described in [8] and [4], and the details of the appropriate conversions between $GF(2^8)$ and $GF(2^4)$ are not mentioned. An attempt to apply the masking technique with mixed field representations encounters two problems: a) how to handle the effect of the conversions between the two field representations, and b) how to handle (efficiently) the additional cipher transformations besides inversion. We show here how to resolve these difficulties, and provide the necessary details for applying this technique to AES implementation. We also overcome the security weakness pointed out in [3]. Our study indicates that this approach indeed produces a smaller area S-Box.

X.2. MASKING INVERSION IN GF(2^N) WITH MIXED FIELD REPRESENTATIONS

Consider the binary field $GF(2^n)$, given in some polynomial representation (denoted *Rep1*) with the irreducible polynomial p_0 (i.e., the elements are considered as coefficients of polynomials over some prime field, where multiplication is done modulo p_0). Suppose that *Rep2* is a different representation of this field. Clearly, there exists at least one isomorphism between *Rep1* and *Rep2*. Since each representation of a finite field is a linear space of dimension n over $GF(2)$, and each isomorphism is a linear transformation, there exists an $n \times n$ binary matrix $\boldsymbol{M}$ that converts elements in *Rep1* to their representation in *Rep2*. The matrix $\boldsymbol{M}^{-1}$ converts from *Rep2* to *Rep1*. In fact, there are n such conversion matrices $\boldsymbol{M}$ for the following reason: each of the n roots of p_0 is a generator of the field, and the set of roots is invariant under field isomorphism. Therefore (since the multiplicative group of the field is cyclic), any isomorphism is uniquely determined by setting one pair of corresponding roots. We point out that this observation suggests a practical method for generating all of the isomorphism matrices. From the point of view of hardware implementation, the conversion between representations is cheap (multiplication by a fixed binary matrix). Therefore, improved efficiency can be expected if the operations in *Rep2* are cheaper to implement than their analogs in *Rep1*.

X.2.1. Inversion

Inversion in $GF(2^n)$, using mixed field representations (say *Rep1* and *Rep2*), is done in the following way: a certain isomorphism is chosen. The input, *x,* given in *Rep1*, is transformed to *Rep2*, and is then inverted. Then, the inverse isomorphism is applied, to obtain the desired inverse in *Rep1*. If $\boldsymbol{M}$ denotes the $n \times n$ conversion matrix from *Rep1* to *Rep2*, and T denotes the field inversion operator in *Rep2*, then for x in *Rep1*, $\boldsymbol{M}x$ is its image in *Rep2*, and the inverse of x is obtained by $\boldsymbol{M}^{-1}T(\boldsymbol{M}x)$.

X.2.2. Masked Inversion: computing $x^{-1}+r$ in GF(2^n) without exposing x

The true input, x, is masked with a random field element $r \neq 0$ and $x+r$ becomes the actual input to the inversion circuit. Conversion to the new representation yields $x+r \rightarrow \boldsymbol{M}(x+r) = \boldsymbol{M}x + \boldsymbol{M}r$. The sequence of operations $\boldsymbol{M}x + \boldsymbol{M}r \xrightarrow[+r]{} \boldsymbol{M}x + \boldsymbol{M}r + r \xrightarrow[+\boldsymbol{M}r]{} \boldsymbol{M}x + r$ can recover $\boldsymbol{M}x + r$, but we choose not to do so. Since $\boldsymbol{M}$ is a regular matrix, and the random mask r is nonzero, $\boldsymbol{M}r$ is also a valid random mask (i.e., it can also take any nonzero value in with equal probability). Therefore, we may securely process $\boldsymbol{M}x+\boldsymbol{M}r$ through the inversion circuit, and "update" the mask from r to $\boldsymbol{M}r$. After the inversion is carried out, the result is converted back to the original representation, by multiplying it by $\boldsymbol{M}^{-1}$, and the final output is $x^{-1} + r$ (i.e., the original mask is now recovered).

Figure X.1 illustrates the mixed representation masked inversion circuit. The cost of this masked inversion is two n-length vector additions in *GF(2)* (XOR), three multiplications of a vector by an $n \times n$ bit binary matrix, and two (presumably cheaper) multiplications plus one square in the new chosen representation.

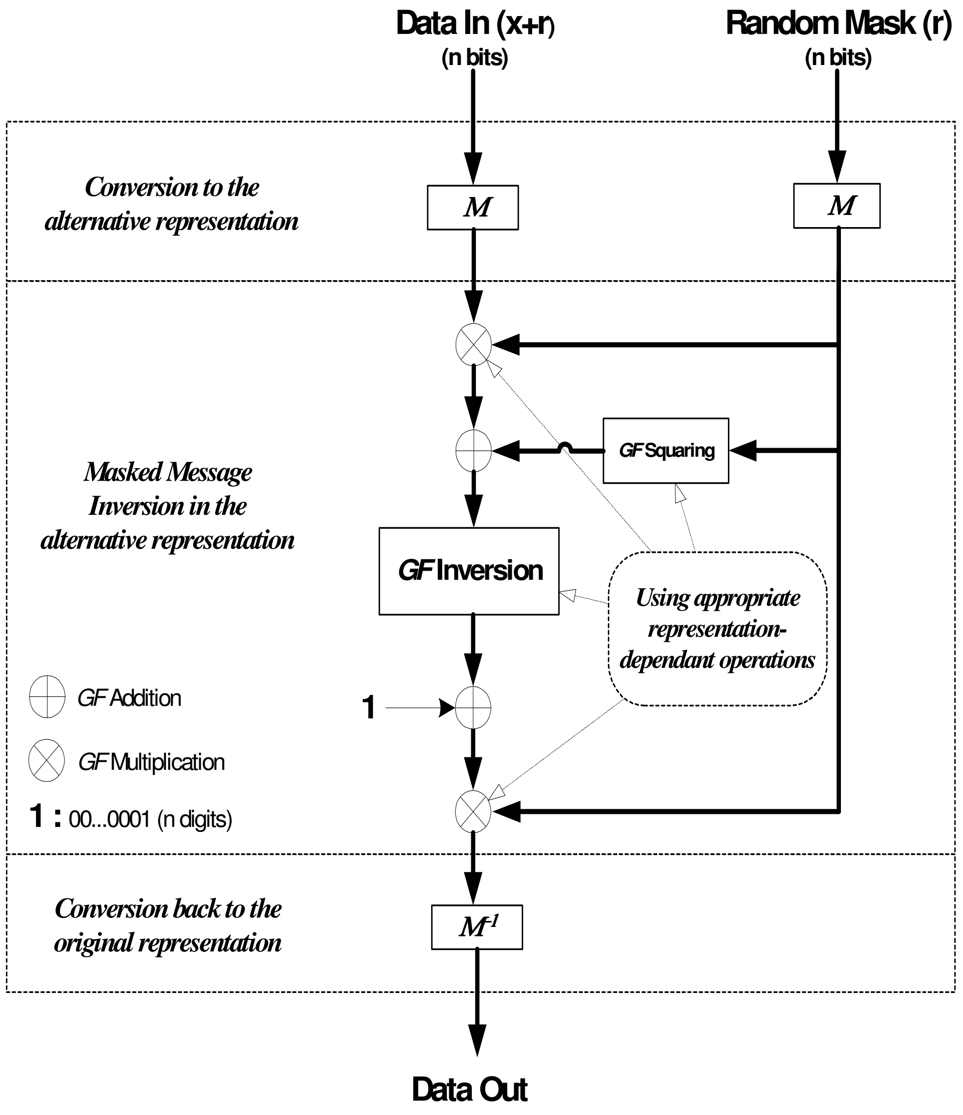
Data In (x+r)
(n bits)
Random Mask (r)
(n bits)
Conversion to the alternative representation
M
M
Masked Message Inversion in the alternative representation
GF Squaring
GF Inversion
Using appropriate representation-dependant operations
GF Addition
GF Multiplication
1 : 00...0001 (n digits)
1
Conversion back to the original representation
M-1
Data Out

Figure X.1: A masked inversion circuit, using mixed field representations. The input (x+r) and the output (x^{-1}+r) are given in one field representation, whereas the inversion is computed in an alternative field representation. The details of the inversion, multiplication and squaring depend on the particular choice of the alternative representation.

X.3. MASKING THE AES ROUND

This section describes the details of an efficient implementation of a masked AES round, using mixed field representations of $GF(2^8)$. It is therefore assumed hereafter that n=8. The resulting circuit is illustrated in Figure X.2. It is displayed after the following explanations.

X.3.1. Masking the AES S-Box

Here, unlike the simpler case for plain inversion, our output is not $x^{-1}+r$, and as we describe below, careful mask updating is required. The AES S-Box computes the transformation $x \mapsto \boldsymbol{A}x^{-1}+b$ for encryption (and key schedule), and $x \mapsto (\boldsymbol{A}^{-1}x + \boldsymbol{A}^{-1}b)^{-1}$ for decryption ($\boldsymbol{A} \in Z_2^{8\times8}, b \in Z_2^{8}$ are given by the standard [7]. The operations are in $GF(2^8)$). For efficiency, we merge the affine transformations with the transformations that account for representation conversion. The resulting modified S-Box operation is $x \mapsto \boldsymbol{AM}^{-1}T(\boldsymbol{M}x)+b$ for encryption, and $x \mapsto \boldsymbol{M}^{-1}T(\boldsymbol{MA}^{-1}x+\boldsymbol{MA}^{-1}b)$ for decryption, where T denotes the inverse operator in the new representation, and $\boldsymbol{M} \in Z_2^{8\times8}$ is the conversion matrix.

At the first round, a randomly generated byte $r \neq 0$ is added to the data, x, and the S-Box input, $x+r$, is passed through a linear (for encryption) or an affine (for decryption) transformation. For brevity, we denote both transformations by $x \mapsto \boldsymbol{U}x+v$ ($\boldsymbol{U} \in Z_2^{8\times8}$, $v \in Z_2^{8}$). Applying it to the masked data yields $\boldsymbol{U}(x+r)+v=\boldsymbol{U}x+\boldsymbol{U}r+v$. Since $\boldsymbol{U}$ is a regular matrix, and $r \neq 0$, it follows that $\boldsymbol{U}r$ is also a valid mask, so we may securely process $\boldsymbol{U}x+\boldsymbol{U}r+v$ through the inversion circuit, and update the mask to $\boldsymbol{U}r$.

After the inversion in the new representation (the T operation), the inverted value is again passed through an affine (for encryption) or linear (for decryption) transformation. We denote both transformations by $x \mapsto \boldsymbol{U}'x+v'$ ($\boldsymbol{U}' \in Z_2^{8\times8}$, $v' \in Z_2^{8}$). In parallel, the updated mask (now $\boldsymbol{U}r$) is updated again, to $\boldsymbol{U}'\boldsymbol{U}r$. Consequently, the twice updated mask equals $\boldsymbol{A}r$ for encryption and $\boldsymbol{A}^{-1}r$ for decryption, and it must be part of the output of the circuit, in order to be used in the subsequent rounds.

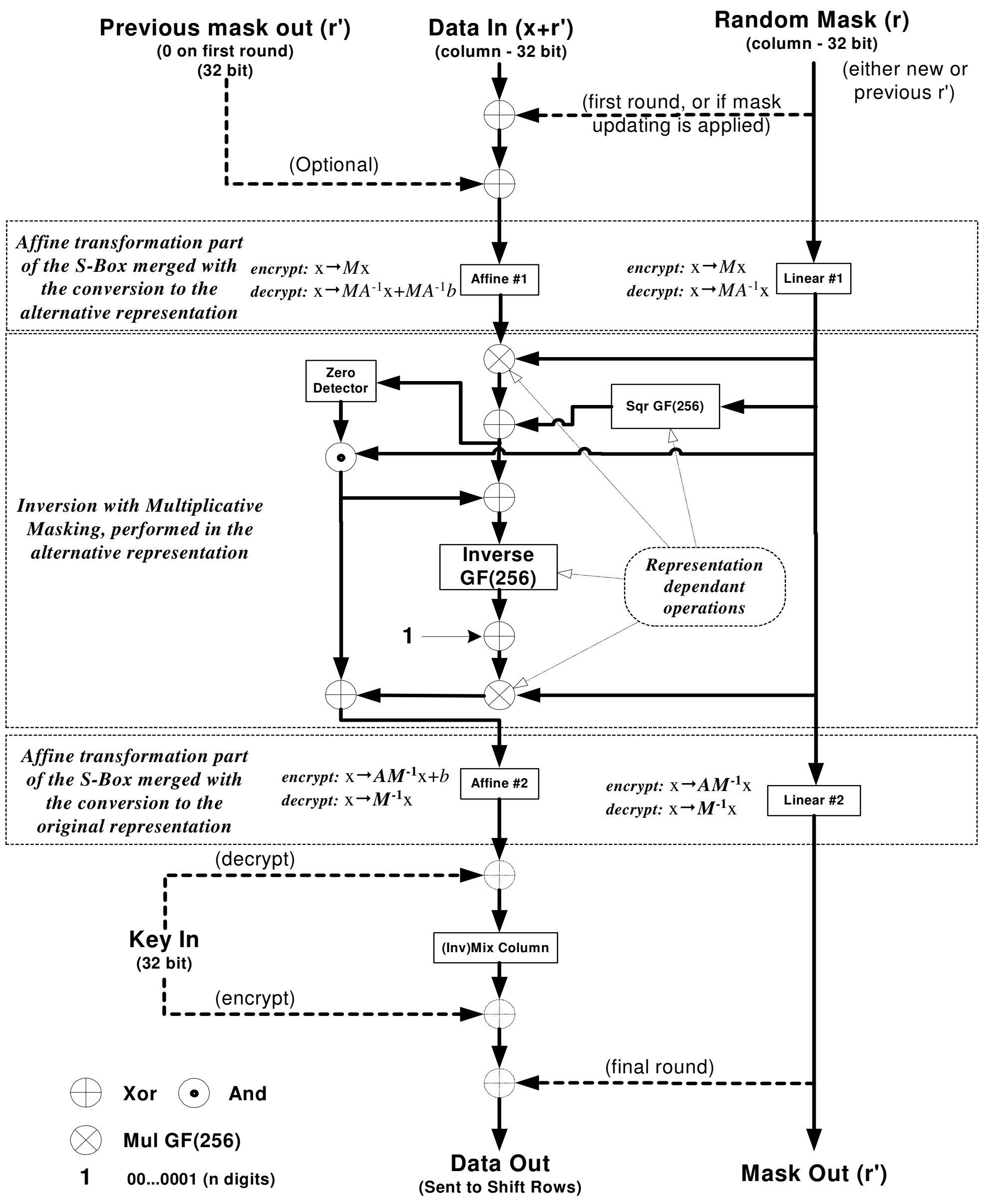

Figure X.2: Circuit design for an AES round (for one column), where the random masking, and the mixed field representation techniques are applied.

X.3.2. New mask for every round

Extra security may be achieved by using a different mask for every round. This option is illustrated in Figure X.2, where a new mask is generated, and the previous one is cleaned out (in a proper order, so as not to expose a protected intermediate result). This optional feature requires a sufficiently fast supply of random bits, and additional storage of 8 bits for the mask.

X3.3. Feeding to the Mix Column and Shift Rows transformations without mask updates

After the S-Box, the (masked) data is either sent to the Mix/InvMix Column module, or added (in $GF(2^8)$) to the round key (the order depends on the encryption/decryption mode). The composition of these operations is an affine transformation whose linear part is the Mix/InvMix Column (regular) matrix denoted $\boldsymbol{W} \in GF(2^8)^{4\times 4}$.

The following problem now occurs. Unlike the S-Box that operates on each byte separately, the Mix/InvMix Column transformations operate on a column (4 bytes). If the mask of the entire column is $\hat{r} \in GF(2^8)^4 \neq 0$, then we also have $\boldsymbol{W}\hat{r} \in GF(2^8)^4 \neq 0$. However, the problem is that some of the bytes of the column $\boldsymbol{W}\hat{r}$ may be zero, and this would destroy the multiplicative masking process of the next round. To overcome this obstacle, we suggest using the same mask for each of the four bytes of the column. If we start with such a mask, this property is also preserved throughout the mask updates. Furthermore, inspection of the Mix Column matrix reveals the following property:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} r \\ r \\ r \\ r \end{pmatrix} = \begin{pmatrix} r \\ r \\ r \\ r \end{pmatrix}$$

(with a similar property for the InvMix Column matrix). This implies that no mask update is required at all, for the Mix/InvMix Column phase.

To avoid mask updating after the Shift Row phase as well, we must use the same mask for all the bytes of the entire block.

Since we keep track of the updated mask throughout the rounds, as part of the circuit output, it is easy to remove the mask after the final round by simply adding it to the output.

X.3.4. Handling a zero data byte without compromising the security

The simplified multiplicative masking in [9] is much cheaper to implement than the original method proposed in [1], because it involves fewer operations. However, in [3] it was argued that the simplified version is less secure than the original one because it does not really mask a zero data byte. Indeed, a zero input to the S-Box is masked by multiplying by *r*, and thus remains zero. Note that the input to the S-Box in the first round is the plaintext, xored with a corresponding part of the key. Thus, an attacker can set this input to be zero by choosing the same plaintext byte as the key that is being guessed, and this may facilitate a DPA attack.

We propose a simple remedy, via a zero detector (*ZD* for short) module, as shown in Figure X.2. *ZD* operates before the inversion, and its input is the "questionable" byte *xr*. Its output *ZD(xr)* is 1 if *xr=0* and 0 otherwise (a zero byte is detected by taking the logical *NOT* of its bits, and then their logical *AND*). This output is expanded to eight identical bits, denoted *ZD8(xr)*, and then *Q(xr)=ZD8(xr) AND r* is computed. Now, *Q(xr)* becomes the input to the inversion circuit. Clearly, for a nonzero *xr, Q(xr)*=*xr*, so the result of the masked inversion is not affected in this case. On the other hand, if *xr=0* we have $Q(xr) = r \neq 0$, so the inversion circuit operates on some nonzero input, and the fact that *xr=0* is therefore not exposed. In this case, the inversion's output is r^{-1}, and the rest of the process produces $r^{-1} \xrightarrow[+1]{} r^{-1} + 1 \xrightarrow[\times r]{} 1 + r$ instead of *r* (= 0^{-1}+*r*), as required (note that 0^{-1} is defined to be *0*, as in [7]). This is corrected by adding the latter output to the output of *ZD8*, which appropriately equals 0 or 1 (see Figure X.3).

For a discussion on a procedure generating nonzero random bytes, see Appendix C.

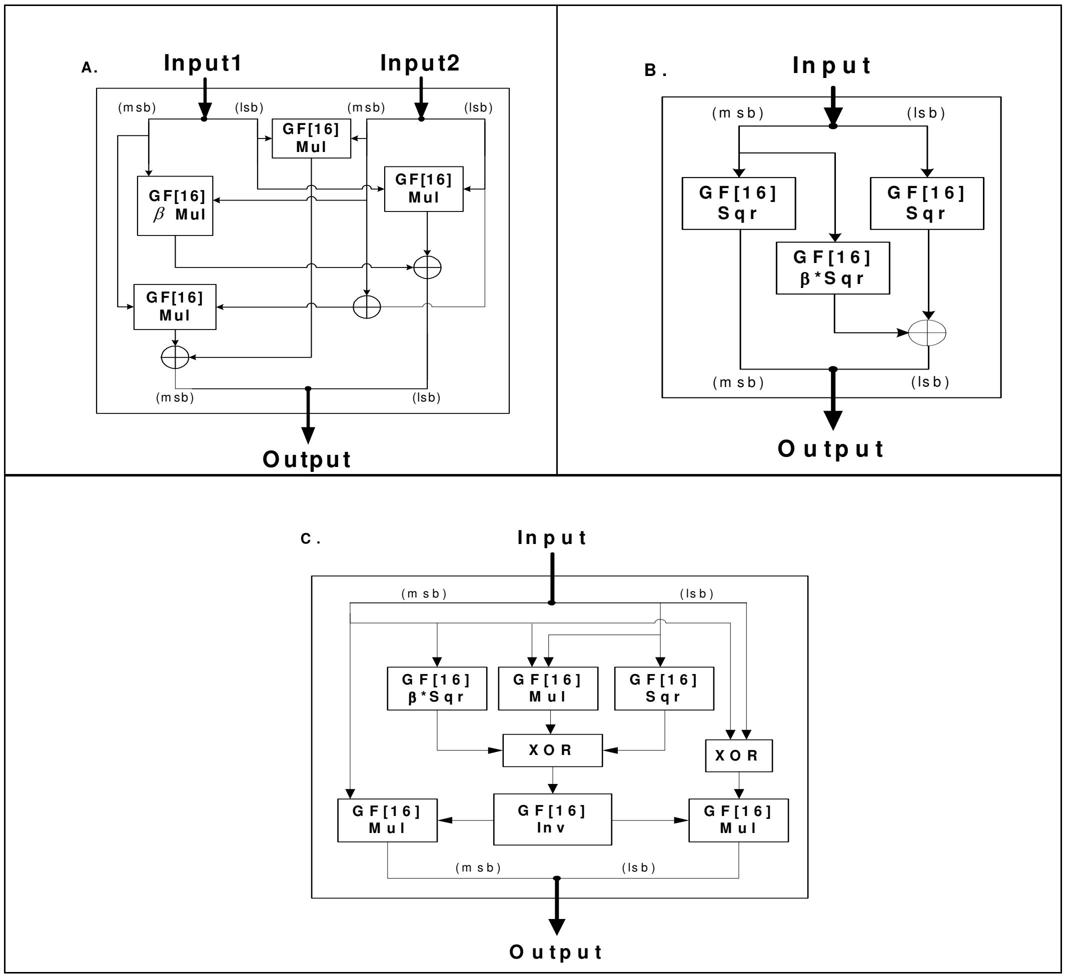

Figure X.3: Three circuits for implementing operations GF(2^8), when the field is represented as the extension of GF(2^4). A. Multiplication (requires three multiplications, three additions, and one multiplication of two elements by β, all in GF(2^4)) B. Squaring (requires one addition, two squares, and one β square). C. Inversion (requires three multiplications, one square, one β square, and one (table) inverse).

X.4. EXAMPLE AND RESULTS

The example discussed in this section uses the representation of $GF(2^8)$ as the extension of $GF(2^4)$, i.e., $GF(2^4)[x]/X^2+X+\beta$ for some polynomial representation of $GF(2^4)$, in which $X^2+X+\beta$ is irreducible. Each 8-bit element of $GF(2^8)$ is a pair of 4-bit nibbles $[a,b]$, $a,b \in GF(2^4)$, and is viewed as the linear polynomial $aX+b$. With this representation, each of the operations multiplication, squaring, inversion in $GF(2^8)$, is replaced by a sequence of several operations in $GF(2^4)$. For $a,b,c,d \in GF(2^4)$ we have: $(aX+b)(cX+d)=(a(c+d)+bc)X+(ac\beta+bd)$, $(aX+b)^2=(a^2)X+(a^2\beta+b^2)$, $(aX+b)^{-1}=a(a^2\beta+b^2+ab)^{-1}X+(a+b)(a^2\beta+b^2+ab)^{-1}$, where in these expressions, multiplications, inversions and additions, are operations in $GF(2^4)$. The three corresponding circuit designs are shown in Figure X.3.

There are three polynomial representations of $GF(2^4)$. For each representations of $GF(2^4)$, there are eight values of β for which $X^2+X+\beta$ is irreducible. For each β, there are eight conversion matrices. Altogether, there are 192 such field representations of $GF(2^8)$, and their details are provided in Appendix B.

All of the 192 field representations were tested, in order to find an optimal one (in terms of area). These circuits were synthesized using DC Shell 2001.08-SP1 (DC Expert) from Synopsys. The target library was TSMC0.18micron (Artisan SAGE-X). We tested S-Box designs that include both encryption and decryption modes (except for the S-boxes used during the key-schedule, where only encryption mode is needed). The results were compared with the standard design (lookup table, and a multiplicative mask in $GF(2^8)$). The synthesis was performed for different time propagation delays constraints, which enable running frequencies between 66.7 to 111 MHZ. These synthesis results indicate that a significant reduction in area is achieved by the proposed design: more than 45% compared with the straightforward approach. Furthermore, our inspection reveals that some choices among the 192 variations result in significantly smaller area. The representation where $GF(2^4)$ is defined by the reduction polynomial $X^4 + X^3 + X^2 + X + 1$, with the choice choose $\beta=8$ was found to be one of the most favorable option. The conversion that corresponds to this optimal choice matrix ***M*** is [01 0c 50 ed 42 35 67 92] (here, each of the eight hexadecimal numbers represents, in binary form, a column of ***M***).

The bit level expression for squaring in this specific representation of $GF(2^4)$ is $[a_3,a_2,a_1,a_0]^2 = [a_2,\ a_1+a_2,\ a_2+a_3,\ a_0+a_2]$, and the bit level expression for multiplication is $[a_3,a_2,a_1,a_0]*[b_3,b_2,b_1,b_0]=[a_2b_1+a_3b_0+a_3b_1+a_1b_2+a_1b_3+a_2b_2+a_0b_3, a_3b_1+a_2b_2+a_1b_1+a_2b_0+a_0b_2+a_1b_3,\ a_0b_1+a_1b_3+a_1b_0+a_3b_1+a_3b_3+a_2b_2,\ a_3b_2+a_1b_3+a_0b_0+a_2b_3+a_2b_2+a_3b_1]$.

Table X.1 summarizes the cost of the resulting circuit.

Operation	Number of copies
Multiplications 8x8 Matrix with a vector	4 (2 for decryption and 2 for encryption)
Byte multiplexer (selecting encrypt/decrypt)	2
Byte addition over GF(2) (xor)	12
Inverse circuit in $GF(2^4)$	1
Multiplication circuits in $GF(2^4)$	6
Squaring circuit in $GF(2^4)$	3
β*Square circuit in $GF(2^4)$	2
β*Multiplication circuit in $GF(2^4)$	1
Zero Detector	1

Table X.1: Summary of elements and circuits required for the masked S-box.

X.5. CONCLUSION

We described here a design for an efficient and secure hardware implementation of a masked AES round. This masking technique turns out to be more attractive for low resources environments. Experimental results indicate that the improvement, compared with the straightforward implementation, is significant. The masked S-Box design with mixed representations is approximately 45% more efficient, in terms of area, than the standard implementation. Further manual optimization can improve the results of the synthesis that was performed in automatic mode.

Other S-Box implementations use GF(2^8) representations which are different from the one used here. One example is the implementation, optimized for low power AES design, that uses the recursive representation of GF(2^8), $GF\left(\left(\left(2^2\right)^2\right)^2\right)$, as proposed in [6]. The masking technique described here, can be used with any field representation of GF(2^8).

X.6. REFERENCES

[1] M. Akkar, and C. Giraud, "An implementation of DES and AES, secure against some attacks", *CHES 2001, Lecture Notes in Computer Science* 2162, 2001, pp. 309-318.

[2] K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Specification of Camellia – a 128-bit Block Cipher", http://info.isl.ntt.co.jp/camellia/.

[3] J. D. Golic, and C. Tymen, "Multiplicative Masking and Power Analysis of AES", *CHES 2002, Lecture Notes in Computer Science* 2523, 2002, pp. 198-212.

[4] J. Daemen, and V. Rijmen, *The design of Rijndael: AES – The Advanced Encryption Standard*, Springer-Verlag Berlin Heidelberg, 2002. (Section 4.3.2).

[5] H. Lee, "Zodiac: Block Cipher Proposal", http://www.safedigm.com/productpds/download/Safedigm_Zodiac.pdf.

[6] S. Morioka, and A. Satoh, "An optimized S-Box circuit architecture for low power AES design", *CHES 2002, Lecture Notes in Computer Science* 2523, 2003, pp. 172-186.

[7] AES. http://csrc.nist.gov/CryptoToolkit/aes/.

[8] V. Rijmen, "Efficient implementation of the Rijndael S-box", http://www.esat.kuleuven.ac.be/~rijmen/rijndael/sbox.pdf.

[9] E. Trichina, D. De Seta, and L. Germani, "Simplified Adaptive Multiplicative Masking for AES and its secure Implementation", *CHES 2002, Lecture Notes in Computer Science* 2523, 2002, pp. 187-197.

X. APPENDIX A: MULTIPLICATION AND SQUARING IN GF(2^8) MODULO $X^8+X^4+X^3+X+1$

The simplified mask design is expensive, mainly because of the two required multiplication circuits. To appreciate why multiplication is costly, we provide here the bit level expressions for multiplication and squaring in GF(2^8) modulo $X^8+X^4+X^3+X+1$ (with AND, and XOR (+) operations, operating on a pair of bits):

$[a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0]\ [b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0] =$
$b_2a_6+b_3a_5+a_2b_6+a_1b_7+a_0b_0+a_6b_7+a_4b_4+a_3b_5+a_6b_6+b_1a_7+a_7b_6+a_5b_7+(a_5b_4+a_1b_0+a_5b_7+b_3a_6+b_2a_7+$
$b_3a_5+b_1a_7+a_7b_7+a_4b_5+a_4b_4+a_1b_7+a_2b_6+a_0b_1+a_3b_5+a_6b_6+a_3b_6+b_2a_6+a_2b_7+(a_5b_4+a_6b_7+a_7b_6+b_3a_6+$
$a_2b_0+a_3b_6+a_6b_4+a_3b_7+b_3a_7+a_4b_5+a_2b_7+a_4b_6+a_1b_1+b_2a_7+a_0b_2+a_5b_5+(a_1b_7+a_3b_0+a_1b_2+a_0b_3+a_2b_1+($
$a_1b_7+a_1b_3+b_0a_4+a_0b_4+a_3b_1+a_2b_2+(b_1a_4+a_5b_4+a_0b_5+a_3b_7+a_2b_7+a_3b_6+a_4b_6+a_4b_5+b_5a_7+b_3a_6+b_0a_5+$
$a_2b_3+a_6b_6+b_3a_7+a_6b_4+b_2a_7+a_5b_5+a_3b_2+a_5b_7+a_1b_4+(a_6b_4+a_3b_3+a_4b_7+a_3b_7+a_6b_7+b_2a_4+a_6b_5+a_0b_6+$
$a_2b_4+a_5b_6+a_4b_6+a_1b_5+a_7b_4+b_0a_6+a_7b_6+b_3a_7+a_5b_5+b_1a_5+(a_4b_7+a_1b_6+b_1a_6+a_6b_6+a_5b_7+a_7b_7+b_3a_4+$
$a_0b_7+b_5a_7+a_2b_5+b_0a_7+a_3b_4+a_5b_6+a_7b_4+a_6b_5+b_2a_5+b_5a_7X)X)X)X+b_3a_5+a_4b_4+a_3b_5+b_1a_7+b_5a_7+a_5$
$b_4+b_3a_6+b_2a_6+a_7b_4+a_2b_7+a_4b_5+a_4b_7+b_2a_7+a_6b_5+a_3b_6+a_5b_6+a_7b_7+a_2b_6)X+b_3a_5+a_4b_4+a_3b_5+b_1a_7+a$
${}_6b_7+a_7b_6+a_5b_7+a_6b_6+a_3b_7+b_3a_7+b_2a_6+a_6b_4+a_7b_4+a_4b_7+a_6b_5+a_4b_6+a_5b_5+a_5b_6+a_7b_7+a_2b_6)X)X)X$

$[a_7\ a_6\ a_5\ a_4\ a_3\ a_2\ a_1\ a_0]^2 =$
$a_4+a_0+a_6+a_5a_7+(a_7+a_4+a_5a_7+a_6+(a_5+a_1+(a_5a_7+a_4+a_6+a_7+a_5+(a_4+a_7+a_2+a_5a_7+(a_6+a_5+(a_5+a_3+(a_6+$
$a_7+xa_5a_7)X)X)X)X)X)X)$

Multiplication requires 149 XOR and 150 AND operations, which is approximately 1.5 times the corresponding number required with the optimal representation.

APPENDIX B: GF(2^8) REPRESENTATIONS BY GF(2^4) FIELD EXTENSION

To find and count the number of possible representations $GF(2^8)$, as the extension of $GF(2^4)$, we use the following algebraic properties:

1. There are three polynomial representations of $GF(2^4)$ (over $GF(2)$). These are obtained by using the three irreducible reduction polynomials $1+x+x^4$, $1+x^3+x^4$, $1+x+x^2+x^3+x^4$.
2. There are exactly 120 irreducible quadratic polynomials (over $GF(2^4)$) of the form $x^2 + \alpha x + \beta$ (where α and β are in $GF(2^4)$).

It follows that the field $GF(2^8)$ can be represented as the field extensions of $GF(2^4)$ in 360 ways. For our study, we are interested only in polynomials $x^2 + \alpha x + \beta$ where $\alpha = 1$ (because this simplifies the inversion circuit). There are exactly eight such polynomials (listed below). Considering only these eight (out of 120) polynomials, the number of relevant extensions reduces from to 24.

We now note that for each one of the 24 extensions, we need to compute the appropriate conversion to and from the AES standard representation, in order to construct an equivalent S-box. The following two lists provide the details of the 192 GF(2^8) representations and conversion matrices, which were tested.

List I. Bit level operations for GF(2^4)

For each reduction polynomial, the list gives the GF(2^4) inversion table (i.e., the inverses of the 16 elements (in ascending order), written in hexadecimal form), the squaring circuit, and the multiplication circuit, in the corresponding GF(2^4) representation.

1. Reduction polynomial: $x^4 + x + 1$
Inversion : [0, 1, 9, e, d, b, 7, 6, f, 2, c, 5, a, 4, 3, 8]
Squaring : $[a_3,a_2,a_1,a_0]^2 = [a_3,a_1+a_3,a_2,a_0+a_2]$
Multiplication : $[a_3,a_2,a_1,a_0] * [b_3,b_2,b_1,b_0] =$
$[a_1b_2+a_3b_3+a_3b_0+a_2b_1+a_0b_3, a_2b_3+a_0b_2+a_3b_3+a_2b_0+a_1b_1+b_2a_3,$
$a_1b_3+b_2a_3+a_0b_1+a_2b_2+a_2b_3+a_1b_0+a_3b_1, a_0b_0+a_1b_3+a_2b_2+a_3b_1]$

2. Reduction polynomial: $x^4 + x^3 + 1$
Inversion : [0, 1, c, 8, 6, f, 4, e, 3, d, b, a, 2, 9, 7, 5]
Squaring : $[a_3,a_2,a_1,a_0]^2 = [a_2+a_3,a_1+a_3,a_3,a_0+a_2+a_3]$
Multiplication : $[a_3,a_2,a_1,a_0] * [b_3,b_2,b_1,b_0] =$
$[a_0b_3+a_1b_3+a_3b_2+a_2b_3+a_3b_1+a_2b_1+a_1b_2+a_3b_3+a_3b_0+a_2b_2, a_0b_2+a_3b_3+a_1b_1+a_2b_0,$
$a_0b_1+a_3b_2+a_3b_3+a_1b_0+a_2b_3, a_1b_3+a_0b_0+a_2b_3+a_3b_2+a_2b_2+a_3b_1+a_3b_3]$

3. Reduction polynomial: $x^4 + x^3 + x^2 + x + 1$
Inversion : [0, 1, f, a, 8, 6, 5, 9, 4, 7, 3, e, d, c, b, 2]
Squaring : $[a_3,a_2,a_1,a_0]^2 = [a_2+a_1+a_2,a_2+a_3,a_0+a_2]$
Multiplication : $[a_3,a_2,a_1,a_0] * [b_3,b_2,b_1,b_0] =$
$[a_2b_1+a_3b_0+a_3b_1+a_1b_2+a_1b_3+a_2b_2+a_0b_3, a_3b_1+a_2b_2+a_1b_1+a_2b_0+a_0b_2+a_1b_3,$
$a_0b_1+a_1b_3+a_1b_0+a_3b_1+a_3b_3+a_2b_2, a_3b_2+a_1b_3+a_0b_0+a_2b_3+a_2b_2+a_3b_1]$

List II. The 192 Conversion Matrices

For each reduction polynomial, and extension polynomial, the list gives the eight conversion matrices ***M***. Each matrix is represented as eight hexadecimal numbers (two digits each). Every such number represents, in binary form, the appropriate column of ***M***.

1. Reduction polynomial: $x^4 + x + 1$

(a) Extension Polynomial: $x^2 + x + 8$
01 e1 5c 0c af 1b e3 85, 01 e1 5c 0c ae fa bf 89, 01 5c e0 50 a2 02 b8 db, 01 5c e0 50 a3 5e 58 8b, 01 e0 5d b0 f2 04 ad 6f, 01 e0 5d b0 f3 e4 f0 df, 01 5d e1 ed 42 10 a7 92,
01 5d e1 ed 43 4d 46 7f
(b) Extension Polynomial: $x^2 + x+ 9$
01 e1 5c 0c 12 4b 0f d8, 01 e1 5c 0c 13 aa 53 d4, 01 5c e0 50 1e b2 b5 3a, 01 5c e0 50 1f ee 55 6a, 01 e0 5d b0 4e 09 a1 83, 01 e0 5d b0 4f e9 fc 33, 01 5d e1 ed fe 1c 16 72,
01 5d e1 ed ff 41 f7 9f
(c) Extension Polynomial: $x^2 + x + 10$
01 e1 5c 0c 43 46 0e 39, 01 e1 5c 0c 42 a7 52 35, 01 5c e0 50 ae bf 54 36, 01 5c e0 50 af e3 b4 66, 01 e0 5d b0 a3 58 fd d3, 01 e0 5d b0 a2 b8 a0 63, 01 5d e1 ed f2 ad f6 c2,
01 5d e1 ed f3 f0 17 2f
(d) Extension Polynomial: $x^2 + x + 11$
01 e1 5c 0c fe 16 e2 64, 01 e1 5c 0c ff f7 be 68, 01 5c e0 50 12 0f 59 d7, 01 5c e0 50 13 53 b9 87, 01 e0 5d b0 1f 55 f1 3f, 01 e0 5d b0 1e b5 ac 8f, 01 5d e1 ed 4e a1 47 22,

01 5d e1 ed 4f fc a6 cf

(e) Extension Polynomial: $x^2 + x + 12$

01 e1 5c 0c a2 1a 02 d9, 01 e1 5c 0c a3 fb 5e d5, 01 5c e0 50 f3 03 e4 3b, 01 5c e0 50 f2 5f 04 6b, 01 e0 5d b0 43 05 4d 32, 01 e0 5d b0 42 e5 10 82, 01 5d e1 ed ae 11 fa 73,
01 5d e1 ed af 4c 1b 9e

(f) Extension Polynomial: $x^2 + x + 13$

01 e1 5c 0c 1f 4a ee 84, 01 e1 5c 0c 1e ab b2 88, 01 5c e0 50 4f b3 e9 da, 01 5c e0 50 4e ef 09 8a, 01 e0 5d b0 ff 08 41 de, 01 e0 5d b0 fe e8 1c 6e, 01 5d e1 ed 12 1d 4b 93,
01 5d e1 ed 13 40 aa 7e

(g) Extension Polynomial: $x^2 + x + 14$

01 e1 5c 0c 4e 47 ef 65, 01 e1 5c 0c 4f a6 b3 69, 01 5c e0 50 ff be 08 d6, 01 5c e0 50 fe e2 e8 86, 01 e0 5d b0 12 59 1d 8e, 01 e0 5d b0 13 b9 40 3e, 01 5d e1 ed 1e ac ab 23,
01 5d e1 ed 1f f1 4a ce

(h) Extension Polynomial: $x^2 + x + 15$

01 e1 5c 0c f3 17 03 38, 01 e1 5c 0c f2 f6 5f 34, 01 5c e0 50 43 0e 05 37, 01 5c e0 50 42 52 e5 67, 01 e0 5d b0 ae 54 11 62, 01 e0 5d b0 af b4 4c d2, 01 5d e1 ed a2 a0 1a c3,
01 5d e1 ed a3 fd fb 2e

2. Reduction polynomial: $x^4 + x^3 + 1$

(a) Extension Polynomial: $x^2 + x + 2$

01 b1 ec 0c 4f 7c 80 69, 01 b1 ec 0c 4e cd 6c 65, 01 ec 0d 50 ff 60 97 d6, 01 ec 0d 50 fe 8c 9a 86, 01 0d 51 b0 13 c7 94 3e, 01 0d 51 b0 12 ca c5 8e, 01 51 b1 ed 1e 24 91 23,
01 51 b1 ed 1f 75 20 ce

(b) Extension Polynomial: $x^2 + x+ 3$

01 b1 ec 0c f3 2c dc 38, 01 b1 ec 0c f2 9d 30 34, 01 ec 0d 50 43 3c 7a 37, 01 ec 0d 50 42 d0 77 67, 01 0d 51 b0 ae 27 98 62, 01 0d 51 b0 af 2a c9 d2, 01 51 b1 ed a3 28 70 2e,
01 51 b1 ed a2 79 c1 c3

(c) Extension Polynomial: $x^2 + x + 4$

01 b1 ec 0c ff 21 60 68, 01 b1 ec 0c fe 90 8c 64, 01 ec 0d 50 13 6d c7 87, 01 ec 0d 50 12 81 ca d7, 01 0d 51 b0 1e 96 24 8f, 01 0d 51 b0 1f 9b 75 3f, 01 51 b1 ed 4f 95 7c cf,
01 51 b1 ed 4e c4 cd 22

(d) Extension Polynomial: $x^2 + x + 5$

01 b1 ec 0c 43 71 3c 39, 01 b1 ec 0c 42 c0 d0 35, 01 ec 0d 50 af 31 2a 66, 01 ec 0d 50 ae dd 27 36, 01 0d 51 b0 a3 76 28 d3, 01 0d 51 b0 a2 7b 79 63, 01 51 b1 ed f2 99 9d c2,
01 51 b1 ed f3 c8 2c 2f

(e) Extension Polynomial: $x^2 + x + 8$

01 b1 ec 0c af 7d 31 85, 01 b1 ec 0c ae cc dd 89, 01 ec 0d 50 a2 61 7b db, 01 ec 0d 50 a3 8d 76 8b, 01 0d 51 b0 f2 c6 99 6f, 01 0d 51 b0 f3 cb c8 df, 01 51 b1 ed 42 25 c0 92,
01 51 b1 ed 43 74 71 7f

(f) Extension Polynomial: $x^2 + x + 9$

01 b1 ec 0c 13 2d 6d d4, 01 b1 ec 0c 12 9c 81 d8, 01 ec 0d 50 1e 3d 96 3a, 01 ec 0d 50 1f d1 9b 6a, 01 0d 51 b0 4f 26 95 33, 01 0d 51 b0 4e 2b c4 83, 01 51 b1 ed ff 29 21 9f,
01 51 b1 ed fe 78 90 72

(g) Extension Polynomial: $x^2 + x + 14$

01 b1 ec 0c 1f 20 d1 84, 01 b1 ec 0c 1e 91 3d 88, 01 ec 0d 50 4e 6c 2b 8a, 01 ec 0d 50 4f 80 26 da, 01 0d 51 b0 ff 97 29 de, 01 0d 51 b0 fe 9a 78 6e, 01 51 b1 ed 13 94 2d 7e, 01 51 b1 ed 12 c5 9c 93

(h) Extension Polynomial: $x^2 + x + 15$

01 b1 ec 0c a3 70 8d d5, 01 b1 ec 0c a2 c1 61 d9, 01 ec 0d 50 f2 30 c6 6b, 01 ec 0d 50 f3 dc cb 3b, 01 0d 51 b0 42 77 25 82, 01 0d 51 b0 43 7a 74 32, 01 51 b1 ed ae 98 cc 73, 01 51 b1 ed af c9 7d 9e

3. Reduction polynomial: $x^4 + x^3 + x^2 + x + 1$

(a) Extension Polynomial: $x^2 + x + 2$

01 50 b0 0c a3 8b d3 d5, 01 50 b0 0c a2 db 63 d9, 01 b0 ed 50 f2 6f c2 6b, 01 b0 ed 50 f3 df 2f 3b, 01 ed 0c b0 43 7f 39 32, 01 ed 0c b0 42 92 35 82, 01 0c 50 ed af 85 66 9e, 01 0c 50 ed ae 89 36 73

(b) Extension Polynomial: $x^2 + x + 3$

01 50 b0 0c 1e 3a 8f 88, 01 50 b0 0c 1f 6a 3f 84, 01 b0 ed 50 4f 33 cf da, 01 b0 ed 50 4e 83 22 8a, 01 ed 0c b0 fe 72 64 6e, 01 ed 0c b0 ff 9f 68 de, 01 0c 50 ed 13 d4 87 7e, 01 0c 50 ed 12 d8 d7 93

(c) Extension Polynomial: $x^2 + x + 4$

01 50 b0 0c f3 3b df 38, 01 50 b0 0c f2 6b 6f 34, 01 b0 ed 50 43 32 7f 37, 01 b0 ed 50 42 82 92 67, 01 ed 0c b0 ae 73 89 62, 01 ed 0c b0 af 9e 85 d2, 01 0c 50 ed a3 d5 8b 2e, 01 0c 50 ed a2 d9 db c3

(d) Extension Polynomial: $x^2 + x + 5$

01 50 b0 0c 4e 8a 83 65, 01 50 b0 0c 4f da 33 69, 01 b0 ed 50 fe 6e 72 86, 01 b0 ed 50 ff de 9f d6, 01 ed 0c b0 13 7e d4 3e, 01 ed 0c b0 12 93 d8 8e, 01 0c 50 ed 1f 84 6a ce, 01 0c 50 ed 1e 88 3a 23

(e) Extension Polynomial: $x^2 + x + 8$

01 50 b0 0c ae 36 62 89, 01 50 b0 0c af 66 d2 85, 01 b0 ed 50 a2 63 c3 db, 01 b0 ed 50 a3 d3 2e 8b, 01 ed 0c b0 f3 2f 38 df, 01 ed 0c b0 f2 c2 34 6f, 01 0c 50 ed 42 35 67 92, 01 0c 50 ed 43 39 37 7f

(f) Extension Polynomial: $x^2 + x + 9$

01 50 b0 0c 13 87 3e d4, 01 50 b0 0c 12 d7 8e d8, 01 b0 ed 50 1f 3f ce 6a, 01 b0 ed 50 1e 8f 23 3a, 01 ed 0c b0 4e 22 65 83, 01 ed 0c b0 4f cf 69 33, 01 0c 50 ed fe 64 86 72, 01 0c 50 ed ff 68 d6 9f

(g) Extension Polynomial: $x^2 + x + 14$

01 50 b0 0c fe 86 6e 64, 01 50 b0 0c ff d6 de 68, 01 b0 ed 50 13 3e 7e 87, 01 b0 ed 50 12 8e 93 d7, 01 ed 0c b0 1e 23 88 8f, 01 ed 0c b0 1f ce 84 3f, 01 0c 50 ed 4e 65 8a 22, 01 0c 50 ed 4f 69 da cf

(h) Extension Polynomial: $x^2 + x + 15$

01 50 b0 0c 43 37 32 39, 01 50 b0 0c 42 67 82 35, 01 b0 ed 50 ae 62 73 36, 01 b0 ed 50 af d2 9e 66, 01 ed 0c b0 a3 2e d5 d3, 01 ed 0c b0 a2 c3 d9 63, 01 0c 50 ed f2 34 6b c2, 01 0c 50 ed f3 38 3b 2f

APPENDIX C: NON-ZERO MASK GENERATION

We show here two possible circuits for a nonzero random mask generator, which is required for implementing the masking technique.

Smooth non-deterministic generator

Random bits are generated by a hardware random bit generator, and are routed to an *n* bits register. If the content of this register is not entirely zero, it is written to another register, which holds the random mask. The expected number of attempts, required in order to generate a valid mask in this way, is $2^n/(2^n-1)$. This circuit guarantees nonzero masks which are evenly distributed among the 2^n-1 possible nonzero masks, thus offering the maximal possible entropy. For *n*=8, this entropy is $\log_2(255) \approx 7.99435$.

If at some stage, an invalid (i.e., zero) random mask is generated, the mask cannot be refreshed, and the previous mask is reused (of course, the value of this previous mask is unknown to the attacker). This occurs with the probability $1/n$. Therefore, the conditional entropy of a mask, given the previous mask is

$$H(m_t \mid m_{t-1}) = -P(m_t = m_{t-1}) \log P(m_t = m_{t-1}) - \sum_{x \neq m_{t-1}} P(m_t = x) \log P(m_t = x) =$$
$$-\left(1/2^{n-1} \log 1/2^{n-1} + \left(1 - 1/2^{n-1}\right) \log 1/2^n \right)$$

For *n*=8 this equals $1023/128 \approx 7.99218$.

Let us now consider a bit-per-clock random bits generator, an 8 S-Box design and a 16 round AES. The probability of not updating the mask between two consecutive blocks is negligible $(\frac{1}{256})^8 = 2^{-64}$, since this is the probability that the generated random value is zero 8 consecutive times.

Deterministic non-smooth generator

A property of the nonzero mask generating circuit discussed above, which could possibly be considered as a drawback, is that the time required for generating the mask is not constant. Therefore, we propose here an alternative design, which ensures the generation of a non-zero random mask in a constant time.

We assume here that n, the length of the mask, is a power of two. The hardware random bits generator generates $\log n + n - 1$ bits. The first $\log n$ bits form a number *x*, $0 \le x < n$. The other *n*-1 from a number denoted by *y*. Now, a mask of length *n* is generated, where bit number *x* is set to 1, and the other *n*-1 bits assume the values of the bits of *y*.

For example, suppose that *n*=8, and the random bits generator generated the 10 bits 1011100101. Three of them form a number $x = 101_2 = 5$, and the other 7 are $y = 1100101$. Now, the byte $2^5 = 00100000$ is generated (bit number 5 is turned on), and its other 7 zero bits are replaced with the bits of *y*, finally obtaining the nonzero mask 11100101.This process assures that

1. A nonzero mask is generated.
2. Every possible nonzero mask can be generated.
3. Masks with the same Hamming weight have the same probability.
4. High entropy is obtained: $-\sum_{k=1}^{n}\binom{n}{k}\frac{k}{2^{n-1}n}\log\left(\frac{k}{2^{n-1}n}\right)$.

With 2^8 this amounts to $-\sum_{k=1}^{8}\binom{8}{k}{}^{k}\!/_{1024}\log_2\left({}^{k}\!/_{1024}\right)\approx 7.90244$.

Note that in practice, a simpler implementation can be achieved at the cost of drawing one extra random bit. Here, y takes n random bits (instead of n-1), and we perform a logical AND operation to the x-th bit of y. In C notation, the mask is y & (1 << x).