Efficient AES Threshold Implementation on FPGA

UGRC-II final report

Vishal Mohanty (Guided by: Chester Reberio)

Department of Computer Science and Engineering
Indian Institute of Technology, Madras
E-mail: cs15b039@smail.iitm.ac.in

Abstract

AES S-box is vulnerable to various kinds of side-channel attacks, revealing the secret key used for the encryption. In order to prevent the S-boxes from revealing any information, a number of measures such as masking and threshold implementations have been proposed. In this project, we explore the available possible security measures incorporated into the composite field implementations, or as referred to as the *tower field* implementation and find out the most efficient S-box when laid out in FPGA board.

Keywords and phrases. AES, S-box, Composite Fields, Threshold Implementation, Masking

1 Introduction

Low power devices requiring security such as RFID chips rely on compact AES encryption schemes to ensure no leakage of information. While AES in itself is quite secure and doesn't directly reveal anything about the secret key used for encryption, there are other ways of determining the key. These are called as side-channel attacks because they rely on some side-product of the encryption in order to create the exploit. Some popular side-channel attacks are Simple, Differential and Correlation Power Analysis.

1.1 AES S-box operations

The following steps are involved in every step of encryption within the S-box.

1. *SubBytes*: This involves an inversion in $GF(2^8)$ field (for all elements except 0 [1]) and affine transformation on the inverse.
$$Y = A \times X^{-1} + B \tag{1.1}$$
where X is the input to the S-box and A and B are transformation matrices.

2. *ShiftRows*: The first row is not shifted, second row shifted by one byte to the left, third by two bytes and fourth by three bytes.

3. *MixColumns*: The columns are interpreted as coefficients of a 4 term polynomial over $GF(2^4)$. Each column is multiplied modulo $x^4 + 1$ with a fixed polynomial $a(x) = 03x^3 + 01x^2 + 01x + 02$.

[1] Note that the inversion of 0 in any field is returned as 0.

4. *AddRoundKey*: The output from the previous step is xor-ed with the round key for that particular round.

1.2 Composite Field Implementations

Composite Field implementations also known as tower field implementations is a technique to project a vector in the higher dimension to component of vectors in the lower dimension. For eg. a vector in the 8-dimensional space can be considered as an 8-tuple of vectors in the one dimensional space. This is sometimes exploited to perform efficient computations in the lower dimensional space which would otherwise be cumbersome in the original higher dimensional space.

In the AES S-box, the input to the *SubBytes* is an element in the $GF(2^8)$ field and we need to compute inverse of this element and then apply an affine transformation on it. If instead we could project the $GF(2^8)$ vector into component vectors in the lower dimension and perform computations in the lower field and use that to get the inverse in the higher field, it would improve the efficiency of the encryption. This is what is typically done.

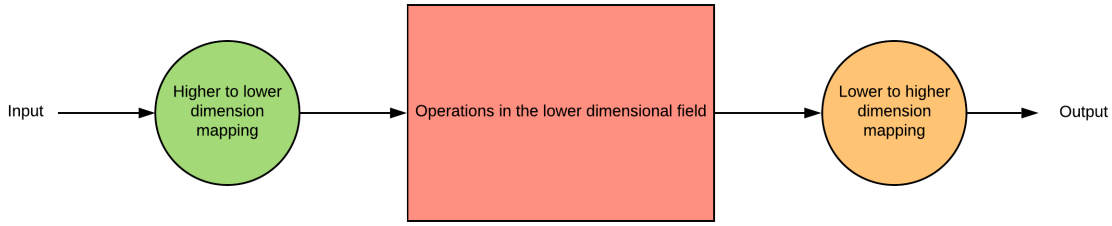

Figure 1: Composite field implementation

Canright and Batina [2] have given the following tower-field implementation of the AES S-box.

- Vector in $GF(2^8)$ field $\equiv$ vector of dimension 2 in $GF(2^4)$ field.
- Vector in $GF(2^4)$ field $\equiv$ vector of dimension 2 in $GF(2^2)$ field.
- Vector in $GF(2^2)$ field $\equiv$ vector of dimension 2 in $GF(2)$ field.

Inversion in $GF(2^8)/GF(2^4)$ field using a normal basis $[Y^{16}, Y]$ where Y and Y^{16} are roots of the equation $X^2 + X + N$ and $N \in GF(2^4)$ is the norm, is computed in three steps as shown by [2].

$$A = A_1 Y^{16} + A_0 Y \tag{1.2}$$

$$B = N \times (A_1 + A_0)^2 + A_1 \times A_0 \tag{1.3}$$

$$A^{-1} = (A_0 \times B^{-1}) Y^{16} + (A_1 \times B^{-1}) Y \tag{1.4}$$

where the multiplications and additions are in the $GF(2^4)$ field. As can be seen, a single inversion in the $GF(2^8)$ field entails 3 multiplications, 1 squaring and 1 inversion in the $GF(2^4)$

field. Surprisingly all of this is more efficient than the original inversion in the higher order Galois field in terms of area but slower.

By similar means, the inversion in the $GF(2^4)$ field can be reduced to multiplications and inversion in $GF(2^2)$ field. It so happens that the inversion in the $GF(2^2)$ field is the same as squaring, equivalent to a bit swap.

$$c = c_1 w^2 + c_0 w \quad \text{is the } GF(2^2) \text{ element} \tag{1.5}$$

$$c^{-1} = c_0 w^2 + c_1 w \tag{1.6}$$

where w and w^2 are roots of the equation $x^2 + x + 1 = 0$. Therefore the non-linear functions of a composite field S-box are only the multiplications. If we convert the $GF(2^4)$ multiplication to $GF(2^2)$ multiplications, then the only non-linear operation is multiplication in $GF(2^2)$.

2 Threshold Implementations

Threshold Implementations (TI) use shares for ensuring security of the S-box. The shares are defined as follows. Let us say that the input to the S-box (X) is an element of $GF(2^8)$ and has an 8-bit representation, then we can split X across say n shares ($X_i (1 \leqslant i \leqslant n)$), each being an element in $GF(2^8)$ which sum up to X.

$$\sum_{i=1}^{n} X_i = X \tag{2.1}$$

Let us say that the S-box applies a function F on the input to convert it to the output Y as follows.

$$F(X) = Y \tag{2.2}$$

Now, even the output can be shared across m shares ($Y_i (1 \leqslant i \leqslant m)$)which sum up to give Y.

$$\sum_{i=1}^{m} Y_i = Y \quad \text{where } Y_i = F_i(X_1, X_2, ..., X_n) \text{ for } 1 \leqslant i \leqslant m \tag{2.3}$$

Typically the number of output shares is the same as the number of input shares, i.e $m = n$, but it is not necessary. There are certain important properties that the shares need to satisfy in order for the sharing to be successful. These properties ensure the security of the S-boxes against the various side channel attacks while still producing the correct output.

1. ***Correctness***: Let $x, y, z, ...$ be the variables which are part of the computation and let $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, ... be the sharings of the input variables. Denote the function acting on the input variables as $F(x, y, z, ...)$ and the component functions as F_i. Then the *correctness* property states that the sum of the individual component functions should equal the expected output.

$$F(x, y, z, ...) = \sum_i F_i(\mathbf{x}, \mathbf{y}, \mathbf{z}, ...) \tag{2.4}$$

2. ***Incompleteness***: Every component function F_i should be independent of at least one of the shares of the input variables $\mathbf{x}, \mathbf{y}, \mathbf{z}, ...$ Typically F_i is designed to be independent of x_i and so on.

3. ***Uniformity***: Let the dimension of the input variables be m and the output be a. Let the number of sharings of the input variables be denoted by s_x and the number of output sharings be s_y such that $\sum_{i}^{s_y} F_i = a$. Then the following should hold true for a *uniform* sharing.

$$|(x_1, x_2, ..., x_{s_x}) \in Sh(x) | F(x_1, x_2, ..., x_{s_x}) = (y_1, y_2, ..., y_{s_y})| = \frac{2^{m(s_x-1)}}{2^{n(s_y-1)}} \tag{2.5}$$

2.1 Security from the properties

The correctness and incompleteness properties ensure that the S-box doesn't leak any information about the input or the output of the S-box. The correctness property must obviously hold for the S-box sharing to even be valid. The incompleteness property assumes that each of the operations within the S-box can be tapped and hence reveals the computations within. But the incomplete sharing within each of the component functions ensures that none of them reveal the input or the output in totality. This works fine as long as the cross-connection between the component functions is negligible.

The uniform masking has an interesting role to play here in that it ensures that the expected value of any leakage signal of an implementation of the sharing F, be it instantaneous or summed over an arbitrary period of time, is constant.

2.2 Constructing shares

2.2.1 Direct Sharing

Direct sharing is a technique that can be used to generate shares that satisfy correctness and incompleteness property. The uniformity property is not always guaranteed. We need to manually check for uniformity. If it is not satisfied, there are techniques to incorporate that such as adding correction terms which we discuss later. The direct sharing technique is described as follows.

- Each input variable is taken as sum of equal number of shares. The number of shares can be determined by finding out the class of permutation that the function belongs to as shown by Bilgin et al. [1] [2].
- Assign the linear terms containing index j to F_{j-1}.
- Assign the quadratic terms containing indices j and $j+1$ to F_{j-1}.
- Assign the quadratic terms containing index j to F_{j-1}.

2.2.2 Correction Terms

Since direct sharing not always results in an uniform sharing Nikova et al. [3] proposed the use of correction terms. These terms when added in pairs to different component functions (or output shares) cancel out each other and hence restore correctness. Now in order to ensure

[2] One important point to note here is that the number of shares needed to ensure the first two properties are satisfied is independent of the field in which we perform the computations. For eg. Multiplication in $GF(2^8)$ field and multiplication in $GF(2^4)$ field are both quadratic permutations and hence can be done using the same number of shares. In fact [1] have proved that if we know the sharing for a representative of the class of permutation, we can derive a sharing for all the permutations belonging to the same class.

incompleteness we just need to be careful so as to not include any i^{th} variable in the i^{th} component function. For eg. in a sharing using 3 shares, we can use x_i or $x_i y_i$ as the correction terms but not $x_i y_j (i \neq j)$ because otherwise incompleteness would be broken. The correction terms are added so as to restore uniformity to the sharing.

2.2.3 Virtual variables

Sometimes in order to reduce the number of shares required for a function, we can introduce virtual variables. It is an additional input to the function which doesn't affect the output. For example, the multiplication of two variables (x, y) requires a minimum of 4 shares to satisfy all properties. Instead, we can use another variable z and reduce the number of computations required to compute the multiplication [1] by reducing the number of shares to 3.

$$
\begin{aligned}
F_1 &= x_2y_2 + x_2y_3 + x_3y_2 + z \\
F_2 &= x_3y_3 + x_1y_3 + x_3y_1 + x_1z + y_1z \\
F_3 &= x_1y_1 + x_1y_2 + x_2y_1 + x_1z + y_1z + z
\end{aligned}
$$

Another interesting point about this sharing is that it uses just a single share for the introduced variable while the inputs use 3 shares. This involves total of 7 elements usage while the 4 sharing would use $4 \times 2 = 8$ elements.

3 AES S-Box Composite/Threshold Implementation

Here we describe a Composite Implementation of the AES S-box along with an analysis of its optimal Threshold Implementation. Consider the following implementation by [1] in which the inversion function in $GF(2^8)$ is computed in the lower $GF(2^4)$ Galois field. It involves 3 $GF(2^4)$ multiplications, 1 $GF(2^4)$ squaring and an inversion in $GF(2^4)$.

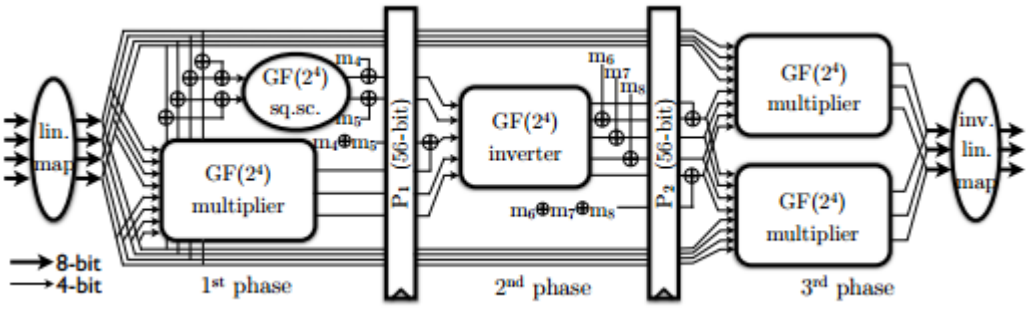

Figure 2: $GF(2^4)$ Threshold Implementation from [1]

As can be seen, they have used 4 shares for computing each of the multiplications and 5 shares for doing the inversion.

Note that we can further reduce the inversion in $GF(2^4)$ to the computations in lower field of $GF(2^2)$. Inversion in the $GF(2^2)$ is simply a squaring 1.5.

Consider computing the inverse in $GF(2^4)$ as 3 multiplications, a squaring and an inversion in $GF(2^2)$ which is essentially a squaring. Since we know that multiplication utilizes the same shares

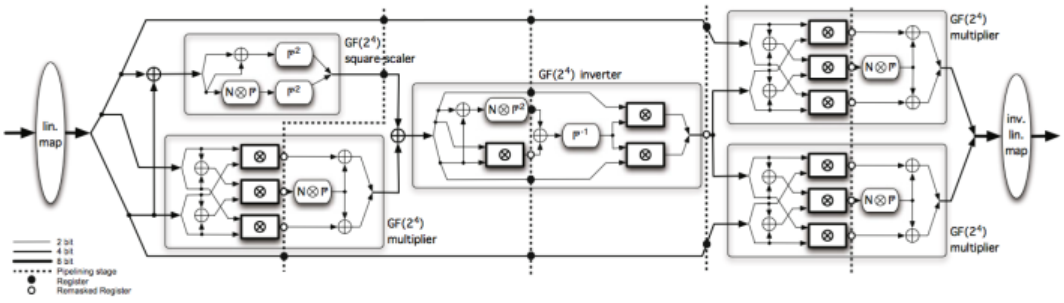

Figure 3: $GF(2^2)$ Implementation from [1]

in any field, we would need an additional $3 \times (5) = 15$ $GF(2^2)$ multiplications and $3 \times 13 = 39$ $GF(2^2)$ additions for the multiplications and 2 squares which are linear.

4 AES functions

In the AES S-box there are only two functions that we need to take care of. One is the $GF(2^4)$ multiplication and the $GF(2^4)$ inverter. Now the multiplication in $GF(2^4)$ using 4 input shares and 3 output shares can be done as follows. It uses 4 multiplications and 4 additions.

$$f_1 = (x_2 + x_3 + x_4)(y_2 + y_3) + y_4 \tag{4.1}$$
$$f_2 = ((x_1 + x_3)(y_1 + y_4)) + x_1 y_3 + x_4 \tag{4.2}$$
$$f_3 = ((x_2 + x_4)(y_1 + y_4)) + x_1 y_2 + x_4 + y_4 \tag{4.3}$$

The output of the multiplication is given by.

$$f = f_1 + f_2 + f_3$$

Most efficient inversion in terms of performance in $GF(2^4)$ [3] uses 5 input and 5 output shares. There is an alternative proposed which uses 4 input and 4 output shares but it performs the inversion in 3 stages (see section 3.2 in [1]). While this would be efficient in terms of area compromises the performance. The idea is to represent an element in the higher order field in the lower order so that we can efficiently invert them. The following is the composite field decomposition shown by [3].

Let $GF(2^2)$ be represented as $GF(2)[t]/(t^2 + t + 1)$. Inversion in $GF(2^2)$ then corresponds to

$$(at + b)^{-1} = at + (a + b) \tag{4.4}$$

Let $GF(2^4)$ be represented by $GF(2^2)[s] = (s^2 + s + \alpha)$. Inversion in $GF(2^4)$ then becomes.

$$(as + b)^{-1} = a(a^2\alpha + ab + b^2)^{-1}s + (a + b)(a^2\alpha + ab + b^2)^{-1} \tag{4.5}$$

If we represent that in terms of the elements of $GF(2)$, the equation can be written as follows.

$$((xt + y)s + (zt + v))^{-1} = (ft + g)s + (ht + k) \tag{4.6}$$

where f, g, h, and k are Boolean functions defined as follows:

$$f = x + y + xv + xyz \tag{4.7}$$
$$g = y + xv + yz + xyz + xyv \tag{4.8}$$
$$h = y + xv + yv + xzv \tag{4.9}$$
$$k = z + v + xz + yz + yv + xzv + yzv \tag{4.10}$$

5 Sharing the AES composite field functions

As discussed in the previous section, we would like to share the inputs and the outputs in order to achieve confidentiality. The sharing for the $GF(2^4)$ multiplication is given by equation 4.1. For the inversion we just present only the sharing for f. The rest can be found in the appendix section of [3]. To start with we need to do a mapping between the variables that we use in our verilog code with the variables used in [3].

5.1 Computing the maps and inverse maps

As can be seen from equation 4.6, x, y, z, v are elements in $GF(2)$ field, i.e they can take only values 0 or 1. Even variables f, g, h, k from equations 4.7 are from the same field. For mapping from our chosen $GF(2^4)$ field to this composite field, we chose the generators as follows. For illustration we have chosen our field as $x^4 + x + 1$ and the generator x in it. For the composite field, s happens to be a generator.

We have used the following algorithm to perform the mapping.

Algorithm 1 Computing map and inverse map

Input: $GF(2^4)$ field in which we want to perform the computation. Composite field as depicted in 4.6. Generators x and s in the respective fields.

Output: $map[]$ mapping from the $GF(2^4)$ elements to the lower field and $inv\_map[]$ storing the inverse mapping.

1: $map[0] = 0$ and $inv\_map[0] = 0$
2: $map[1] = 1$ and $inv\_map[1] = 1$
3: $cur_x \leftarrow 1$ and $cur_s \leftarrow 1$
4: **for** $i = 1$ to 14 **do**
5: $\quad cur_x \leftarrow cur_x \times x$
6: $\quad cur_s \leftarrow cur_s \times s$
7: $\quad$ Assign $map[cur_x] = cur_s$
8: $\quad$ Assign $inv\_map[cur_s] = cur_x$
9: **Output** $map[]$ and $inv\_map[]$

Representing (x, y, z, v) in terms of a 4 bit number and even the input as a 4 bit number (a, b, c, d), we get the following truth table.

In order to construct the verilog code for performing the mapping, we use the Karnaugh map to find the shortest linear transformation from (a, b, c, d) to (x, y, z, v) and vice versa. We show the Karnaugh maps for the computing the mappings as follows. Note that a represents 1 and $\tilde{a}$ represents 0 and similarly for the other input variables.

Table 1: Truth table for mapping $GF(2^4)$ elements to the double composite field

a	**b**	**c**	**d**	**x**	**y**	**z**	**v**
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	1	0	0
0	1	0	0	0	1	1	0
1	0	0	0	1	1	1	0
0	0	1	1	0	1	0	1
0	1	1	0	0	0	1	0
1	1	0	0	1	0	0	0
1	0	1	1	1	0	1	1
0	1	0	1	0	1	1	1
1	0	1	0	1	0	1	0
0	1	1	1	0	0	1	1
1	1	1	0	1	1	0	0
1	1	1	1	1	1	0	1
1	1	0	1	1	0	0	1
1	0	0	1	1	1	1	1

Table 2: Karnaugh map for x

x	$\tilde{a}\tilde{b}$	$\tilde{a}b$	ab	$a\tilde{b}$
$\tilde{c}\tilde{d}$	0	0	1	1
$\tilde{c}d$	0	0	1	1
cd	0	0	1	1
$c\tilde{d}$	0	0	1	1

Table 3: Karnaugh map for y

y	$\tilde{a}\tilde{b}$	$\tilde{a}b$	ab	$a\tilde{b}$
$\tilde{c}\tilde{d}$	0	1	0	1
$\tilde{c}d$	0	1	0	1
cd	1	0	1	0
$c\tilde{d}$	1	0	1	0

Table 4: Karnaugh map for z

z	$\tilde{a}\tilde{b}$	$\tilde{a}b$	ab	$a\tilde{b}$
$\tilde{c}\tilde{d}$	0	1	0	1
$\tilde{c}d$	0	1	0	1
cd	0	1	0	1
$c\tilde{d}$	0	1	0	1

Table 5: Karnaugh map for v

v	$\tilde{a}\tilde{b}$	$\tilde{a}b$	ab	$a\tilde{b}$
$\tilde{c}\tilde{d}$	0	0	0	0
$\tilde{c}d$	1	1	1	1
cd	1	1	1	1
$c\tilde{d}$	0	0	0	0

As can be seen clearly, the output variables can be represented in terms of the input variables as follows.

$$v = d; \tag{5.1}$$
$$x = a; \tag{5.2}$$
$$y = a \oplus b \oplus c; \tag{5.3}$$
$$z = a \oplus b; \tag{5.4}$$

And similarly the inverse map can be obtained as follows.

$$a = x; \tag{5.5}$$
$$b = x \oplus z; \tag{5.6}$$
$$c = y \oplus z; \tag{5.7}$$
$$d = v; \tag{5.8}$$

5.2 Shared inversion in new composite field

Given that the input shares are x_i for $1 \leqslant i \leqslant 5$ and similarly for y, z and v. For computing the shares for x, we choose x_i at random for $1 \leqslant i \leqslant 4$ and x_5 is determined from the xor of x and the others x_i's.

The following is the code that computes the shares of f using the input shares. These f_i's are then xor-ed to compute f.

$f1$ $x2 \oplus y2 \oplus (x2 \oplus x3 \oplus x4 \oplus x5)(v2 \oplus v3 \oplus v4 \oplus v5) \oplus (x2 \oplus x3 \oplus x4 \oplus x5)(y2 \oplus y3 \oplus y4 \oplus y5)(z2 \oplus z3 \oplus z4 \oplus z5)$

$f2$ $x3 \oplus y3 \oplus x1(v3 \oplus v4 \oplus v5) \oplus v1(x3 \oplus x4 \oplus x5) \oplus x1v1 \oplus x1(y3 \oplus y4 \oplus y5)(z3 \oplus z4 \oplus z5) \oplus y1(x3 \oplus x4 \oplus x5)(z3 \oplus z4 \oplus z5) \oplus z1(x3 \oplus x4 \oplus x5)(y3 \oplus y4 \oplus y5) \oplus x1y1(z3 \oplus z4 \oplus z5) \oplus x1z1(y3 \oplus y4 \oplus y5) \oplus y1z1(x3 \oplus x4 \oplus x5) \oplus x1y1z1$

$f3$ $x4 \oplus y4 \oplus x2v1 \oplus x1v2 \oplus x1y1z2 \oplus x1y2z1 \oplus x2y1z1 \oplus x1y2z2 \oplus x2y1z2 \oplus x2y2z1 \oplus x1y2z4 \oplus x2y1z4 \oplus x1y4z2 \oplus x2y4z1 \oplus x4y1z2 \oplus x4y2z1 \oplus x1y2z5 \oplus x2y1z5 \oplus x1y5z2 \oplus x2y5z1 \oplus x5y1z2 \oplus x5y2z1$

$f4$ $x5 \oplus y5 \oplus x1y2z3 \oplus x1y3z2 \oplus x2y1z3 \oplus x2y3z1 \oplus x3y1z2 \oplus x3y2z1$

$f5$ $x1 \oplus y1$

$$f = f1 \oplus f2 \oplus f3 \oplus f4 \oplus f5 \tag{5.9}$$

6 Results

We have used Xilinx Vivado 2017.2 to perform our simulations, synthesis and implementations. We are measuring the number of LUTs used to perform the $GF(2^8)$ inversion in the S-Box. We observe that we are able to do the inversion with very slight increase in the number of LUTs for the implementation. The synthesis takes about 10 more LUTs than the basic inversion but the implementation is nearly the same.

We have summarized the results in the following table.

	Synthesis	**Implementation**
No sharing	41	41
Only Multiplications sharing	49	41
Only Inversion sharing	43	42
Complete sharing	50	42

7 Conclusion

The inclusion of the threshold implementation along with the composite field increases the number of LUTs used while making our S-Box secure against side-channel attacks. In our report, we have shown the various side-channel attacks that we need to counter against. We also discussed the existing threshold implementations and implemented one that we found suitable, namely the one by [3]. For multiplication we used the one described in [1]. Our results have shown that the added threshold support does increase the number of LUTs used on an FPGA. This is the cost we pay for added security.

I would like to thank Aditya for supporting me with my project and helping me complete it in time.

8 Future Work

In our simulations, we have used a specific $GF(2^4)$ field $(x^4 + x + 1)$. We need to run the simulations on the rest of the composite fields available. This will give the most efficient implementation of all which can then be deployed on low-power RFID devices to maintain performance while strengthening security.

The next step forward would be to evaluate the shared implementation against the known side-channel attacks. Also comparing the most efficient sharing given by our runs with the one used by Canright coupled with threshold measures would give us a sense of how efficient our implementation is compared to the existing methods.

References

[1] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. "Threshold Implementations of All 3×3 and 4×4 S-Boxes". In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 76–91. ISBN: 978-3-642-33027-8.

[2] D. Canright and Lejla Batina. "A Very Compact "Perfectly Masked" S-Box for AES". In: *Applied Cryptography and Network Security*. Ed. by Steven M. Bellovin, Rosario Gennaro, Angelos Keromytis, and Moti Yung. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 446–459. ISBN: 978-3-540-68914-0.

[3] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *Information and Communications Security*. Ed. by Peng Ning, Sihan Qing, and Ninghui Li. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 529–545. ISBN: 978-3-540-49497-3.