# Fast and Secure Finite Field Multipliers

Danuta Pamula, Arnaud Tisserand

# Fast and Secure Finite Field Multipliers

Danuta Pamula
Silesian University of Technology
Institute of Electronics
Gliwice, Poland
Email: dpamula@polsl.pl

Arnaud Tisserand
CNRS, IRISA, Univ. Rennes 1, INRIA
Lannion, France
Email: arnaud.tisserand@irisa.fr

***Abstract*—The paper presents details on fast and secure $GF(2^m)$ multipliers dedicated to elliptic curve cryptography applications. Presented design approach aims at high efficiency and security against side channel attacks of a hardware multiplier. The security concern in the design process of a $GF(2^m)$ multiplier is quite a novel concept. Basing on the results obtained in course of conducted research it is argued that, as well as efficiency of the multiplier impacts the efficiency of the cryptoprocessor, the security level of the multiplier impacts the security level of the whole cryptoprocessor. Thus the goal is to find a tradeoff, to compromise efficiency, in terms of speed and area, and security of the multiplier. We intend to secure the multiplier by masking the operation, either by uniformization or by randomization of the power consumption of the device during its work. The design methodology is half automated. The analyzed field sizes are the standard ones, which ensure that a cryptographic system is mathematically safe. The described architecture is based on principles of Mastrovito multiplication method. It is very flexible and enables to improve the resistance against side channel attacks without degrading the multiplier efficiency.**

***Index Terms*—Finite-field multiplication, side-channel attack, arithmetic protection, elliptic-curve cryptography**

## I. Introduction

Nowadays due to infeasibility of breaking mathematically secure cryptographic algorithm [10], in order to break cryptographic systems, attackers develop more and more efficient physical attacks (ways of tampering the systems) [2], [20]. Noninvasive physical attacks are relatively cheap and effective. By observation of power consumption or changes in electromagnetic field of the device, the attacker is able to discover a cryptographic key. The necessity of protecting devices against those types of attacks – the side channel attacks (SCA) [21] is of high importance. Many countermeasures were already developed and research on this topic is still of top interest [4], [5], [7], [14], [26]. With developement of new countermeasures against SCA, new ways of tampering cryptographic devices appear [27]. Cryptanalysts use more and more effective statistical and learning methods in order to correlate recorded emissions with the secret data, on which the devices operates [2], [20].

In our research we consider elliptic curve cryptography (ECC) systems [13]. Generally, the ECC system has three layers: protocol layer (ECC protocol operations), curve layer (operations on points on the curve - primitive elliptic curve operations), field layer (finite-field arithmetic, operations on point coordinates). There are many protections developed for two upper layers [7], but not at the finite field one. We argue that protecting also the lowest layer will rend the system more secure.

## II. Finite field multiplication and ECC

Finite-field arithmetic units are the most critical units of the ECC systems. There are different types of fields used, we consider binary finite fields extensions ($GF(2^m)$) of sizes recommended by National Institute for Standards and Technology (NIST) [10].

In $GF(2^m)$ two basic operations are defined: addition and multiplication [17]. The elements of $GF(2^m)$ can be represented by different bases, we consider $GF(2^m)$ elements represented by polynomial basis of the form $\{1, x, x^2, ..., x^{m-2}, x^{m-1}\}$. Thus addition is a very simple operation, as it is a XOR operation. On the other hand, multiplication is regarded as a costly operation. The multiplication requires performing two steps: large vector multiplication $a(x)b(x)$ and reduction of the result modulo irreducible polynomial $f(x)$ generating the field. There exist many algorithms for performing binary field multiplication [1], [3], [6], [16], [22]. They are either dedicated for specific applications or developed in order to increase the arithmetic unit efficiency in terms of speed, area or power consumption. We have decided to analyse existing $GF(2^m)$ algorithms regarding not only hardware efficiency but also advantages to security. After vast analysis of the most popular $GF(2^m)$ algorithms [23], we have determined the algorithm, which seems to be the most suitable for our purposes. As the most suitable the Mastrovito multiplication algorithm [19] was chosen. We intend to secure arithmetic units against power analysis side channel attacks (PA SCA) [18]. Power analysis attacks rely on analysis of power consumed during operation of the device. The attacker tries to correlate recorded emissions with the data, on which the device operates. Our goal is to uniformise or randomise the power consumption of a multiplier unit in such a way that it is impossible to recognize the operation or establish its time boundaries. We intend to do this via specific architecture of the multiplier and algorithmic modifications. We do not consider circuit level countermeasures against SCA due to the assumption that the solution should be platform independent.

### A. Mastrovito algorithm overview

In classic $GF(2^m)$ multiplication method one has to perform separately multiplication and reduction. In Mastrovito method [19] those two steps are combined. Let $A$ be a matrix representing operand $a$ and $b$ be a vector representing operand $b$. Operands $a$ and $b$ are elements of $GF(2^m)$. Then

$$c = ab \bmod f = Ab \bmod f = (A^L + A^H R)b = Mb,$$

where $A^L, A^H$ matrices are, respectively, lower and upper parts of matrix $A$ (which represents operand $a$) and $R$ is so called reduction matrix. Matrix $M$ is called the Mastrovito matrix.

## III. Multiplier architecture design idea

The most problematic aspect of Mastrovito algorithm is the manipulation of matrix $M$. Let us consider a field of size $m = 233$, matrix $M$ is then of size $233 \times 233$, which is about 54289 bits, thus storing it requires huge amount of resources. Furthermore, due to its dependency not only on $f(x)$, which is fixed for given field, but also on operand $a$, matrix $M$ needs to be recomputed each time the multiplier is used. There exist few methods for computation of matrix $M$ coefficients. Regarding properties of considered fields we have decided to use such an equation: $M = (A^L + A^H R)$. In our case matrix $R$ (reduction matrix) has a small number of nonzero coefficients and matrices $A^H$, $A^L$ are triangle matrices.

Our idea is to partition matrix $M$ into $16 \times 16$-bit submatrices and to design submultipliers, which will on-the-fly calculate coefficients of selected submatrix of $M$ and multiply the submatrix by the proper 16-bit word of vector $b$. Submultiplier units compute partial vectors $c_{i,j}$, where $i, j = 0, 1, \ldots, \frac{m}{16}$. Final result, vector $c$, is a concatenation of $c_{i,j}$ words: $c = c_0 \ \& \ c_1 \ \& \ c_2 \ \& \ \cdots \ \& \ c_{(\frac{m}{16}-1)} \ \& \ c_{\frac{m}{16}}$, where:

$$\begin{aligned}
c_0 &= c_{0,0} \oplus c_{0,1} \oplus \cdots \oplus c_{0,\frac{m}{16}} = \\
&= M_{(0,0)}b_0 \oplus M_{(0,1)}b_1 \oplus \cdots \oplus M_{(0,\frac{m}{16})}b_{\frac{m}{16}} \\
c_1 &= c_{1,0} \oplus c_{1,1} \oplus \cdots \oplus c_{1,\frac{m}{16}} = \\
&= M_{(1,0)}b_0 \oplus M_{(1,1)}b_1 \oplus \cdots \oplus M_{(1,\frac{m}{16})}b_{\frac{m}{16}} \\
&\vdots \\
c_{\frac{m}{16}} &= c_{\frac{m}{16},0} \oplus c_{\frac{m}{16},1} \oplus \cdots \oplus c_{\frac{m}{16},\frac{m}{16}} = \\
&= M_{(\frac{m}{16},0)}b_0 \oplus M_{(\frac{m}{16},1)}b_1 \oplus \cdots \oplus M_{(\frac{m}{16},\frac{m}{16})}b_{\frac{m}{16}};
\end{aligned}$$

$\oplus$ designates addition in $GF(2)$, that is `xor` and expression $M_{(i,j)}b_j$ denotes multiplication of submatrix of $M$ by a word of $b$. In the following paragraphs we will present details on the design process of the $GF(2^{283})$ multiplier. In the process we have to derive equations for each submultiplier unit. At first, in order to create each submatrix submultiplier we have to derive equations for computation of coefficients of each submatrix $M_{i,j}$. Then, we have to identify submatrices of similar structures and design as many different submultipliers as necessary. It occurs that some submatrices of $M$ are constructed in a similar way (are built of similar parts of vectors $a, f$) and require similar manipulation in order to obtain partial solution $c_{i,j}$.

As an example we present equations derived for submatrices $M_{i,j}$ for $i = 0$ and $j = 0, 1, \ldots, 17$:

$$\begin{aligned}
M_{0,0} &= A_0^H R_0 \oplus A_1^H R_1 \oplus A_0^L \\
M_{0,1} &= A_1^H R_0 \oplus A_2^H R_1 \\
&\vdots \\
M_{0,15} &= A_{15}^H R_0 \oplus A_{16}^H R_1 \\
M_{0,16} &= A_{16}^H R_0 \oplus A_{17}^H R_1 \oplus A_0^H R_4 \\
M_{0,17} &= A_{17}^H R_0 \oplus A_0^H R_5
\end{aligned}$$

Matrix $A$ and its decomposition into submatrices $A^H$ and $A^L$ for $m = 283$, is presented below, where $A_i^L$, $A_i^H$, $i = 0, 1, \ldots, 17$ are $16 \times 16$-bit blocks of matrices $A^L, A^H$ :

$$\left[\begin{array}{cccccc}
A_0^L & 0 & 0 & \ldots & 0 & 0 \\
A_1^L & A_0^L & 0 & \ldots & 0 & 0 \\
\vdots & \vdots & \ddots & \ldots & \ddots & \vdots \\
A_{16}^L & A_{15}^L & A_{14}^L & \ldots & A_0^L & 0 \\
A_{17}^L & A_{16}^L & A_{15}^L & \ldots & A_1^L & A_0^L \\
\hline
A_0^H & A_1^H & A_2^H & \ldots & A_{16}^H & A_{17}^H \\
0 & A_0^H & A_1^H & \ldots & A_{15}^H & A_{16}^H \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\
0 & 0 & 0 & \ldots & A_0^H & A_1^H \\
0 & 0 & 0 & \ldots & 0 & A_0^H
\end{array}\right] \Rightarrow \left[\begin{array}{c} A^L \\ \hline A^H \end{array}\right]$$

Matrix $R$ partitioning for $m = 283$ is as follows:

| **R** | 0 | 1 | 2 | 3 | $\cdots$ | 14 | 15 | 16 | 17 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | $R_0$ | 0 | 0 | 0 | $\cdots$ | 0 | 0 | $R_4$ | $R_5$ |
| 1 | $R_1$ | $R_0$ | 0 | 0 | $\cdots$ | 0 | 0 | 0 | 0 |
| 2 | 0 | $R_1$ | $R_0$ | 0 | $\cdots$ | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | $R_1$ | $R_0$ | $\cdots$ | 0 | 0 | 0 | 0 |
| $\vdots$ | 0 | 0 | 0 | $R_1$ | $\ddots$ | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 | $\ddots$ | $R_0$ | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 | | $R_1$ | $R_0$ | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 | $\cdots$ | 0 | $R_1$ | $R_0$ | 0 |
| 17 | 0 | 0 | 0 | 0 | $\cdots$ | 0 | 0 | $R_1$ | $R_0$ |

where $R_0, R_1, R_4, R_5$ are $16 \times 16$-bit submatrices of matrix $R$, fields marked with zeroes designate submatrices with all coefficients equal to 0.

One can observe that the following types of submultipliers are needed: $A_k^H R_0$, $A_k^H R_1$, $A_k^H R_4$, $A_k^H R_5$, where $k = 0, 1, \ldots, 17$. In next step the equations for those submultipliers are derived and optimized. It will be shown how it is done on the example of $A_k R_0$ submultiplier.

Lets simplify the case and assume, just to illustrate the procedure, that $M_{0,1} = A_1^H R_0$. In order to derive equations one has to know how $A_1^H$ and $R_0$ matrices are constructed.

Derived equations are as follows:

$$
\begin{aligned}
M_{0,0}(\ 0) =& a(30 \text{ downto } 15) \oplus a(25 \text{ downto } 10) \oplus a(23 \text{ downto } 8) \oplus \\
& a(18 \text{ downto } 3); \\
& \vdots \\
M_{0,0}(\ 3) =& a(27 \text{ downto } 12) \oplus a(22 \text{ downto } 7) \oplus a(20 \text{ downto } 5) \oplus \\
& a(15 \text{ downto } 0); \\
M_{0,0}(\ 4) =& a(26 \text{ downto } 11) \oplus a(21 \text{ downto } 6) \oplus a(19 \text{ downto } 4); \\
& \vdots \\
M_{0,0}(\ 8) =& a(22 \text{ downto } 7) \oplus a(17 \text{ downto } 2) \oplus a(15 \text{ downto } 0); \\
M_{0,0}(\ 9) =& a(21 \text{ downto } 6) \oplus a(16 \text{ downto } 1); \\
M_{0,0}(10) =& a(20 \text{ downto } 5) \oplus a(15 \text{ downto } 0); \\
M_{0,0}(11) =& a(19 \text{ downto } 4); \\
& \vdots \\
M_{0,0}(15) =& a(15 \text{ downto } 0);
\end{aligned}
$$

where $M_{0,0}(i)$, $i = 0, 1, \ldots, 15$, is the $i$-th column of submatrix $M_{0,0}$. The last step is to create architectures for submultipliers computing partial product $c_{i,j}$. Partial product $c_{i,j}$ is the result of multiplication of submatrix $M_{i,j}$ and $b_j$ (16-bit word of vector $b$), where $i, j = 0, 1, \ldots, 17$ for $m = 283$. Partial product $c_{i,j}$ for 16-bit wide row $i = 0$ is computed as follows:

$$
\begin{aligned}
c_{0,0} &= M_{0,0}(0)b(\ 0) \oplus M_{0,0}(1)b(\ 1) \\
&\quad \oplus \cdots \oplus M_{0,0}(15)b(15); \\
c_{0,1} &= M_{0,1}(0)b(16) \oplus M_{0,1}(1)b(17) \\
&\quad \oplus \cdots \oplus M_{0,1}(15)b(31); \\
&\vdots \\
c_{0,16} &= M_{0,16}(0)b(256) \oplus M_{0,16}(1)b(257) \\
&\quad \oplus \cdots \oplus M_{0,0}(15)b(271); \\
c_{0,17} &= M_{0,17}(0)b(272) \oplus M_{0,17}(1)b(273) \\
&\quad \oplus \cdots \oplus M_{0,17}(15)b(287);
\end{aligned}
$$

Then $c_0$ (16-bit word of $c$) is as follows:

$$c_0 = c_{0,0} \oplus c_{0,1} \oplus c_{0,2} \oplus \cdots \oplus c_{0,15} \oplus c_{0,16} \oplus c_{0,17}.$$

Final result $c$ is computed, as shown at the beginning of the section, as a concatenation of partial products $c_i$.

In this work we describe multipliers for field of sizes $m = 233, 283, 409$. The method can be applied for other field sizes. It is half automated. Necessary equations are derived using a software application. Having the equations, one may design the submultiplier units. The method was at first applied for $m = 233$ and then extended for other fields.

On Figure 1 the architecture of the multiplier is presented, where blocks $M0, M1, \ldots, M5$ are submultiplier units. The work of all submultipliers is scheduled by a finite-state machine (FSM). The submultiplier units can be used in arbitrary order, due to the fact that submatrices and partial results $c_{i,j}$ are independent of each other. There exist many variations of submultiplications schedule.

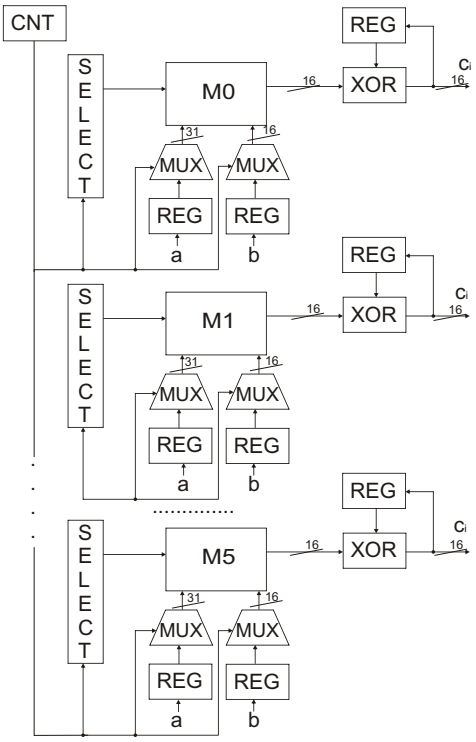


Fig. 1. Architecture of the multiplier

We have automated the method for deriving equations needed to design submultipliers, however the optimization needs to be done by the designer. Performed optimization impacts efficiency and security, thus it is the designer who decides, which feature is more important. The method for "inserting" countermeasures against SCA is also half automated.

## IV. Security analysis

Our first concern was the multiplier's efficiency in terms of speed and area. After many tests we have finally come up with an FSM schedule and structure of the multiplier (number of submultiplier instances used, number of FSM states, etc), which yielded satisfying hardware efficiency. Table I presents implementation results of the three analysed multipliers on Virtex-6 xc6vlx240t. The table presents results reported by Xilinx ISE 12.2 environment. Number of clock cycles given in the table is the total number of cycles required by a multiplier to initialize all registers, fetch data and provide the result, they were recorded basing on performed circuit simulation. The total number of clock cycles required just for a finite field multiplication is 56 clock cycles, in case of the second version of multiplier for $m = 409$ it is 202 cycles. In Table II the exemplary existing solutions are presented, basing on $AT$ factor ($area \times operation\ time$), we may compare them with ours. Due to the fact that authors of other solutions define area in many different manners, the most popular unit is slice, in order to be able to do a comparison we have estimated

TABLE II
IMPLEMENTATION RESULTS OF VARIOUS EXISTING SOLUTIONS

| | Multiplier (type, $m$) | device | reported area | slices (Xilinx) | clock frequency / critical path delay | mult. time [$\mu$s] | AT [slice $\times$ $\mu$s] |
|---|---|---|---|---|---|---|---|
| [9] | Folded m=233 | Xilinx FPGA probably xc2v-6000 | 933 LUT 699 FF | 466 | 165.8 MHz | 2.8 | 1311.2 |
| | Pipelined Architecture m=233 | | 216923 LUT 108578 FF RAM blocks | 108461 | 167.2 MHz | 0.012 | 1297.4 |
| [12] | classical m=233 | xc2v-6000 -ff1517-4 FPGA | 37296 LUT 37552 FF | 18648 | 13 ns (77 MHz) | 3.03 | 56484.8 |
| | Karatsuba m=233 | | 11746 LUT 13941 FF | 5873 | 11.07 ns (90.33 MHz) | 2.6 | 15148.3 |
| | Massey Omura m=233 | | 36857 LUT 8543 FF | 18428 | 15.91 ns (62.85 MHz) | 3.7 | 68315 |
| | Sunar,Koc m=233 | | 45435 LUT 41942 FF | 22717 | 10.73 ns (93.20 MHz) | 2.5 | 56795.8 |
| [25] | m=191 | XCV2600E | 8721 | 8721 | N/A | 82.4 | 718610.4 |
| [8] | m=210 | xcv-300e-6 | 334 | 334 | N/A | 2.21 | 738.1 |
| | m=233 | xc2v-6000-4 | 415 | 415 | N/A | 2.42 | 1004.3 |
| | m=239 | xcv-300e-6 | 385 | 385 | N/A | 2.47 | 950.9 |
| [15] | m=210 | xcv1000e | 343 [4] 420 [7] | 343 420 | 17.1 MHz | 12.3 | 4218.9 5166 |
| [11] | m=239 | xcv-300e-6 | 347(359) | 347 | 75 MHz | 3.1 | 1075.7 |
| [11] | m=120 | v1000fg680-6 | 603 | 603 | 88.47 MHz | 1.36 | 817.9 |
| | m=240 | | 1211 | 1211 | 57.99 MHz | 4.14 | 5011.9 |
| | | | 1203 | 1203 | 41.41 MHz | 5.8 | 6972.2 |
| | m=480 | | 2426 | 2426 | 59.03 MHz | 8.1 | 19726.9 |
| | | | 2403 | 2403 | 18.52 MHz | 25.9 | 62280.8 |
| | m=720 | | 3641 | 3641 | 56.83 MHz | 12.7 | 46129.2 |
| | | | 3603 | 3603 | 12.64 MHz | 56.9 | 205234.2 |
| | m=1080 | | 5458 5403 | 5458 5403 | 54.61 MHz 8.4 MHz | 19.8 128.6 | 107940.7 694671.4 |
| [28] | m=163 g=1 | Virtex-5 V5LX110-3 | 504 LUT 500 FF | 252 | 1.78 ns | 0.29 | 73.08 |
| | m=163 g=11 | | 1179 LUT 495 FF | 589 | 2.36 ns | 0.035 | 20.62 |
| | m=163 g=82 | | 7176 LUT 495 FF | 3588 | 3.54 ns | 0.007 | 25.12 |
| | m=571 g=1 | | 1731 LUT 1727 FF | 865 | 1.85 ns | 1.056 | 913.44 |
| | m=571 g=24 | | 8051 LUT 1720 FF | 4025 | 2.78 ns | 0.067 | 269.68 |
| | m=571 g=32 | | 10350 LUT 1720 FF | 5175 | 2.98 ns | 0.054 | 279.45 |

TABLE I
IMPLEMENTATION RESULTS OF OUR BASIC NOT-SECURED MULTIPLIERS FOR $m = 233, 283, 409$

| Algorithm | area [slices] | $f$ [MHz] | clock cycles | AT |
|---|---|---|---|---|
| Mastrovito 233 | 940 | 297 | 75 | 237.37 |
| Mastrovito 283 | 1112 | 187 | 75 | 445.99 |
| Mastrovito 409 a | 1648 | 175 | 75 | 706.28 |
| Mastrovito 409 b | 1149 | 187 | 221 | 1357.9 |

number of equivalent slices for each multiplier. We have assumed the following approximations for given devices: 1 slice = 2 LUT; except for Virtex-5 where 1 slice = 4 LUT. It is difficult to compare existing solutions. It is not always said if the results are for the entire device i.e. multiplier with the control of inputs and outputs or just for the core of arithmetic operator. One can implement described multipliers, however usually not all implementation details are given and explained. Moreover implementation results depend on software environment used and depend greatly on the manner of coding. The other problem is that in the newest solutions authors tend to implement their multipliers on ASIC devices. We find that it is impossible to reliably compare solutions implemented on FPGA devices with solutions implemented on ASIC devices. Thus the Table II may lack some recent solutions.

In order to introduce countermeasures against SCA we had to analyse the activity of the multiplier and define its weaknesses. By analysing activity we mean simulation and analysis of instantaneous power consumption of the multiplier [29]. We have performed hardware simulations and later current measurements, in order to verify if our assumptions regarding hardware simulation of the device behaviour were correct. In the paper we will mainly present results of hardware simulations due to the fact that they fairly model the device behaviour [24]. All presented traces are exemplary traces of random multiplications. Traces are not averaged in order to demonstrate their characteristics properly. Averaged traces are

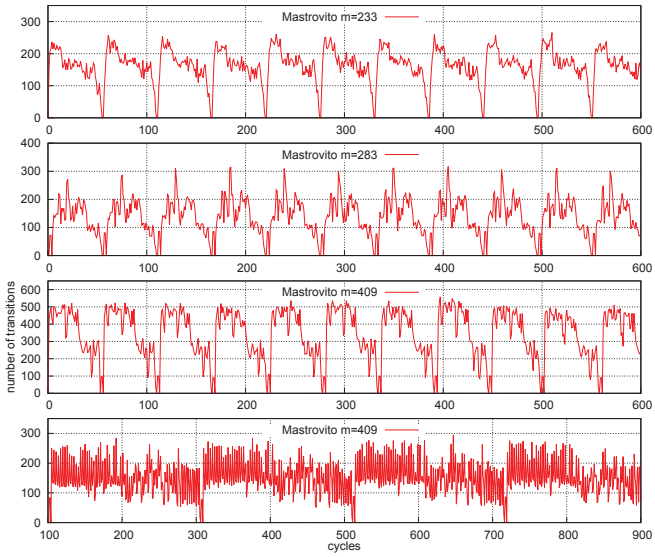


Fig. 2. Activity simulation results for random $GF(2^m)$ multiplications for basic multipliers.

more uniform.

### A. Activity simulation

The activity of the multiplier was simulated with ChipScopePro tool. We had inserted the activity monitors [24], which count number of signals changing state $(1 \to 0, 0 \to 1)$ in one clock cycle. The activity for each clock cycle is recorded, plotted with use of ChipScopePro tool and analysed. On Figure 2 we present the activity recorded for series of $GF(2^m)$ multiplications of random operands performed using the three presented multipliers. ChipScopePro tool enables to record the activity during the device work. For $m = 409$ we present two versions of original multiplier, in the first one multiplication lasts 56 cycles as in case of others, in the second one 202 cycles. The shape of the activity trace is very specific (step-wave shape) and allows by simple observation to define operation time boundaries. We argue that this may allow to recognise some upper layer operations (operations on points of the curve $2P$ or $P + Q$).

The hardware simulation with ChipScopePro tool gives results very close to those obtained during current measurements. Software simulation (e.g. using ModelSim) does not adequately simulate all conditions, which may occur during device operation.

### B. Architecture advantages to security

The specific behaviour of the multiplier is due to its specific architecture. The multiplier is built of various submultipliers operating on vectors of different sizes. The basic schedule of the submultiplier unit operation assumes starting all units at once. Thus, if some units are utilised more than the others the number of units utilised in one state gradually decreases e.g. at first we are using 7 units, later 5 and at the end only 2. Concluding, at the beginning the activity is higher and then decreases with time.

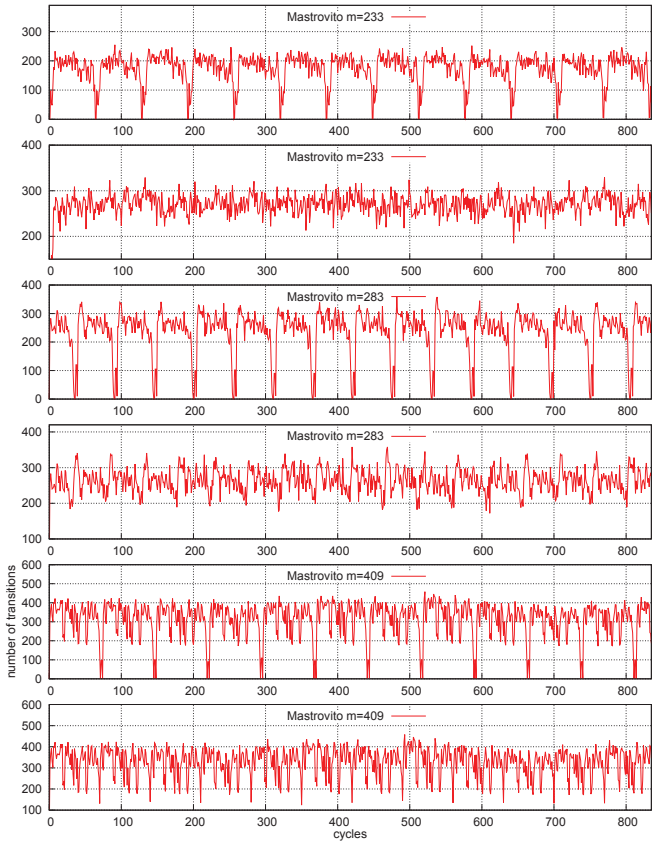


Fig. 3. Uniformization examples

However, the schedule of subunits operation is very flexible. Operations on submatrices as well as submultipliers are independent of each other. Such architecture's flexibilty enables to adapt the units to our needs.

Regarding security we have defined two architecture/schedule modification goals: the uniformization of the multipliers' activity and the randomization of the multipliers' activity. We have simulated many versions of the multipliers and we present here exemplary ones. Presented architecture is very flexible and can be adapted to different requirements.

## V. Modifications examples

Modification of submultiplier units operation schedule allows activity trace uniformization and randomization. We have performed various modifications of the schedule and obtained promising results for both type of activity masking. Figures 3, 4 present the most interesting, according to us, results obtained for activity uniformization and randomization for $m = 233, 283, 409$. On some activity traces there are visible drops to zero, which maybe informative as they may allow to establish operation time boundaries. Those drops appear mostly due to features of the algorithm controlling the multiplier activity tests. In order to distinguish each operation we had designed the control in such a way, that there is a clear

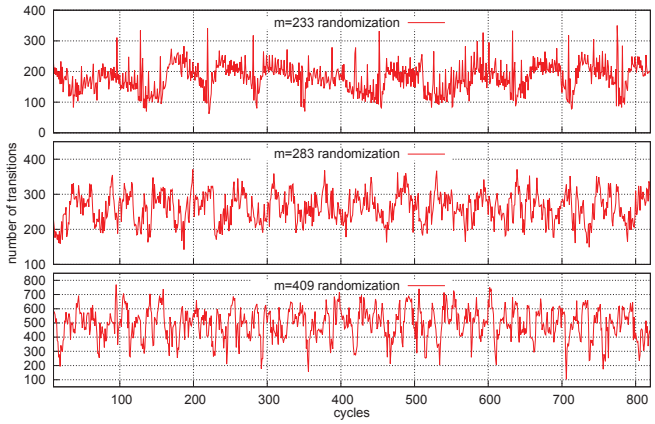


Fig. 4. Randomization examples

pause between each operation. Later we have redesigned the control in order to omit unnecessary pauses.

In case of uniformization, we had tried to uniformize number of bits/signals that change state in every clock cycle. It was done in half automated manner. A programme generates exemplary schedule, basing on the number of bits on which each submultiplier operates, and then it has to be verified in hardware. Due to the fact that submultipliers operate on vectors of various sizes it is not an easy task, however, as presented on second from the top plot on Fig. 3 it is possible to obtain satisfying results. The example of uniformization procedure for $m = 233$ is presented in Table III. Left part presents original schedule and right part a new one. In the Table, it is presented which multipliers are used in a certain state (*multipliers used* column) and the total number of bits/signals changing in that state is given (*bits* column). One can observe that standard deviation of maximal number of signals changing state in a given clock cycle in uniformized schedule is 7 while in original schedule it was 71.

The case is different for randomization. In order to randomize multiplication activity, we had tried many approaches. The best results gave the solution in which every submultiplier has its separate, independent of other controlling FSM, which starts scheduling submultiplications in a different moment of time. The starting sequence, the order in which submultipliers are turned on, is variable. The crucial point was to design a method, which will allow generating random/variable starting sequence. The goal was to make the starting sequence random but not to introduce too many additional units. The best solution was to use a single LFSR register or a combination of LFSR registers and control them with certain bits of input operands $a, b$ .

The efficiency of proposed protections was one of the most important goals in our research however while analysing the countermeasures we have kept in mind that protections cannot reduce the efficiency in terms of speed and area of the multiplier. In Table IV we present implementation results of our Mastrovito multipliers. As *uni1*, *uni2* we denote first and second uniformisation example for a given field size (see top, bottom plot accordingly for each field size on Figure 3) and *rand1* denotes randomisation example for a given field size presented on Figure 4. Analysing implementation results

TABLE IV
IMPLEMENTATION RESULTS OF OUR EXEMPLARY SECURED MULTIPLIERS FOR $m = 233, 283, 409$ COMPARED WITH THEIR NOT-SECURED VERSIONS

| Algorithm | area [slices] | $f$ [MHz] | cycles | AT [slice $\times$ us] |
|---|---|---|---|---|
| **Mastrovito 233** | **940** | **297** | **75** | **237.37** |
| uni1 | 865 | 414 | 75 | 156.7 |
| uni2 | 972 | 225 | 48 | 207.36 |
| rand2 | 975 | 319 | avg.80 | 244.51 |
| **Mastrovito 283** | **1112** | **187** | **75** | **445.99** |
| uni1 | 1085 | 223 | 75 | 364.91 |
| uni2 | 1043 | 219 | 75 | 357.19 |
| rand2 | 1113 | 220 | avg.78 | 394.6 |
| **Mastrovito 409** | **1648** | **175** | **75** | **706.28** |
| uni1 | 1532 | 175 | 75 | 656.57 |
| uni2 | 1537 | 217 | 75 | 531.22 |
| rand2 | 1586 | 217 | avg.74 | 540 |

presented in Table IV one may observe that for greater field sizes the area grows proportionally. Basing on the results obtained for $m = 233$ we have decided to use in all multipliers in primary version a controlling FSM with 56 states. However, in order to reduce area one may increase number of states of controlling FSM and decrease number of instances of submultipliers. Moreover in such architecture as proposed, protections do not alter significantly the multiplier efficiency.

## VI. CONCLUSION

In the paper we have presented the idea for the architecture of hardware $GF(2^m)$ multiplier design based on Mastrovito algorithm principles. The architecture is very flexible. It is efficient and allows introducing countermeasures against SCA.

Due to similarity of structure of multiplier for field sizes defined as secure, $m = 233, 283, 409$ basing on already obtained results we may say that it is possible to find efficient and secure against side channel power analysis solutions. The specific architecture allows introducing SCA countermeasures without adding excessive overhead. As far as we are concerned the design of the finite-field arithmetic units with security concern is a new idea, which may contribute to the security of ECC systems.

Presented hardware multipliers efficiency is satisfying, moreover, first results concerning their security seem also satisfying. With such arithmetic units it should be harder to recognise higher level elliptic curve operations such as doubling or addition of points of the curve.

As the multipliers are intended to be a part of a cryptographic system we plan to conduct intensive protection evaluations using attacks. The attacks should verify how finite field operation protections influence the security of the system. The presented countermeasures are not autonomous countermeasures thus they should be verified in conjunction with other ECC system countermeasures.

TABLE III
UNIFORMIZATION EXAMPLE FOR $m = 233$

| state | multipliers used | | | | | | | bits |
|---|---|---|---|---|---|---|---|---|
| **0** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **0a** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **0b** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **0c** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **1** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **1a** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **1b** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **1c** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **2** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **2a** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **2b** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **2c** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **3** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **3a** | m0_1 | m1 | m5 | m4 | m0_2 | mr | mc | *384* |
| **3b** | m0_1 | m1 | m5 | m4 | m0_2 | mr | | *336* |
| **3c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **4** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **4a** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **4b** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **4c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **5** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **5a** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **5b** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **5c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **6** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **6a** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **6b** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **6c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **7** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **7a** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **7b** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **7c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **8** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **8a** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **8b** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **8c** | m0_1 | m1 | m5 | m4 | m0_2 | | | *296* |
| **9** | m0_1 | m1 | m5 | m4 | | | | *249* |
| **9a** | m0_1 | m1 | m5 | m4 | | | | *249* |
| **9b** | m0_1 | m1 | m5 | m4 | | | | *249* |
| **9c** | m0_1 | m1 | m5 | m4 | | | | *249* |
| **10** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **10a** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **10b** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **10c** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **11** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **11a** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **11b** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **11c** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **12** | m0_1 | m1 | m5 | m9 | | | | *234* |
| **12a** | m0_1 | m1 | | m9 | | | | *172* |
| **12b** | m0_1 | m1 | | m9 | | | | *172* |
| **12c** | m0_1 | m1 | | m9 | | | | *172* |
| **13** | m0_1 | m1 | | m9 | | | | *172* |
| **13a** | m0_1 | m1 | | m9 | | | | *172* |
| **13b** | m0_1 | m1 | | m9 | | | | *172* |
| **13c** | m0_1 | m1 | | | | | | *110* |

standard deviation : 71, range =274

$\Longrightarrow$

| state | multipliers used | | | | | | | | bits |
|---|---|---|---|---|---|---|---|---|---|
| **0** | m0_1 | m1 | | | m0_2 | m5 | | m9 | *281* |
| **0a** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **0b** | m0_1 | m1 | mr | mc | | | m4 | | *275* |
| **0c** | m0_1 | m1 | mr | | | m5 | m4 | m9 | *289* |
| **1** | m0_1 | m1 | | | m0_2 | m5 | | | *281* |
| **1a** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **1b** | m0_1 | m1 | | | m0_2 | m5 | | | *281* |
| **1c** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **2** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **2a** | m0_1 | m1 | | mc | m0_2 | | m4 | | *282* |
| **2b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **2c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **3** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **3a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **3b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **3c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **4** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **4a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **4b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **4c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **5** | m0_1 | m1 | mr | mc | m0_2 | m5 | | | *307* |
| **5a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **5b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **5c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **6** | m0_1 | m1 | | mc | m0_2 | | m4 | | *282* |
| **6a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **6b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **6c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **7** | m0_1 | m1 | mr | mc | m0_2 | | | m9 | *307* |
| **7a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **7b** | m0_1 | m1 | | | m0_2 | m5 | | m9 | *281* |
| **7c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **8** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **8a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **8b** | m0_1 | m1 | | | m0_2 | m5 | | m9 | *281* |
| **8c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **9** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **9a** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **9b** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **9c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **10** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **10a** | m0_1 | m1 | | | m0_2 | m5 | | m9 | *281* |
| **10b** | m0_1 | m1 | | | m0_2 | m5 | | m9 | *281* |
| **10c** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **11** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **11a** | m0_1 | m1 | | | m0_2 | m5 | m4 | | *296* |
| **11b** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **11c** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **12** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **12a** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **12b** | m0_1 | m1 | mr | | | m5 | m4 | | *289* |
| **12c** | m0_1 | m1 | | mc | | m5 | | m9 | *282* |
| **13** | m0_1 | m1 | | mc | m0_2 | | m4 | | *282* |
| **13a** | m0_1 | m1 | | mc | m0_2 | | m4 | | *282* |
| **13b** | m0_1 | m1 | mr | mc | m0_2 | m5 | | | *307* |
| **13c** | m0_1 | m1 | | mc | m0_2 | | m4 | | *282* |

standard deviation : 7.6, range = 32

No. of bits used by each multiplier: m0_1 = 47; m1=63; mr=40; mc=48; m0_2=47; m5=62; m4=77; m9=62

## ACKNOWLEDGMENT

This work has been supported in part by the PAVOIS project (ANR 12 BS02 002 01).

## REFERENCES

[1] C. W. Chiou, J.-M. Lin, C.-Y. Lee, and C.-T. Ma. Novel Mastrovito Multiplier over $GF(2^m)$ Using Trinomial. In *Proc. 5th International Conference on Genetic and Evolutionary Computing*, ICGEC '11, pages 237–242, Washington, DC, USA, 2011. IEEE Computer Society.

[2] C. Clavier, J.-S. Coron, and N. Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. *CHES 2000, LNCS 1956. Springer*, pages 252–263.

[3] J.-P. Deschamps, J. L. Imana, and G. D. Sutter. *Hardware Implementation of Finite-Field Arithmetic*. McGraw-Hill, 2009.

[4] C. Edwards. Secure-system designers strive to stem data leaks. *Communications of the ACM*, 58(4):18–20, April 2015.

[5] H. Eldib and C. Wang. Synthesis of masking countermeasures against side channel attacks. In *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 114–130. Springer International Publishing, 2014.

[6] H. Fan and M. A. Hasan. A new approach to subquadratic space complexity parallel multipliers for extended binary fields. *IEEE Transactions on Computers*, 56(2):224–233, Feb. 2007.

[7] J. Fan, X. Guo, E. D. Mulder, P. Schaumont, B. Preneel, and I. Verbauwhede. State-of-the-art of Secure ECC Implementations: A Survey on Known Side-channel Attacks and Countermeasures. In *HOST*, pages 76–87, 2010.

[8] E. Ferrer, D. Bollman, and O. Moreno. A fast finite field multiplier. In *Proc. 3rd International Conference on Reconfigurable Computing: Architectures, Tools and Applications*, ARC'07, pages 238–246. Springer, 2007.

[9] A. P. Fournaris and O. Koufopavlou. $GF(2^k)$ multipliers based on Montgomery Multiplication Algorithm. In *Proceedings of the 2004 International Symposium on Circuits and Systems (ISCAS).*, volume 2, pages II–849–52 Vol.2, May 2004.

[10] P. Gallagher. FIPS PUB 186-3 Federal Information Processing Standards Publication Digital Signature Standard (DSS), 2009.

[11] M. Garcia-Martinez, R. Posada-Gomez, G. Morales-Luna, and F. Rodriguez-Henriquez. FPGA implementation of an efficient multiplier over finite fields $GF(2^m)$. In *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pages 21–26, Sept 2005.

[12] C. Grabbe, M. Bednara, J. Teich, J. von zur Gathen, and J. Shokrollahi. FPGA designs of parallel high performance $GF(2^{233})$ multipliers. In *Proc. International Symposium on Circuits and Systems (ISCAS)*, volume 2, pages 268–271, May 2003.

[13] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[14] H. Kim, N. Bruce, H.-J. Lee, Y. Choi, and D. Choi. Side Channel Attacks on Cryptographic Module: EM and PA Attacks Accuracy Analysis. In *Information Science and Applications*, volume 339 of *Lecture Notes in Electrical Engineering*, pages 509–516. Springer Berlin Heidelberg, 2015.

[15] P. Kitsos, G.Theodoridis, and O. Koufopavlou. An efficient reconfigurable multiplier architecture for Galois field $GF(2^m)$. *Microelectronics Journal*, 34(10):975 – 980, 2003.

[16] C. K. Koc and T. Acar. Montgomery multiplication in GF($2^k$). *Designs, Codes and Cryptography*, 14(1):57–69, Apr. 1998.

[17] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, 2nd edition, 1994.

[18] S. Mangard, E. Oswald, and T. Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards*. Springer, 2007.

[19] E. Mastrovito. *VLSI Architectures for Computation in Galois Fields*. PhD thesis, Department of Electrical Engineering, Linkoping University, Sweden, 1991.

[20] T. S. Messerges. Using Second-Order Power Analysis to Attack DPA Resistant Software. *CHES, LNCS 1956. Springer*, pages 238–251, 2000.

[21] S. Micali and L. Reyzin. Physically observable cryptography. *Theory of Cryptography, First Theory of Cryptography Conference (TCC), LNCS 2951. Springer*, pages 278–296, 2004.

[22] A. H. Namin, W. Huapeng, and M. Ahmadi. A high-speed word level finite field multiplier in $F_{2^m}$ using redundant representation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(10):1546–1550, Oct. 2009.

[23] D. Pamula. *Arithmetic operators on $GF(2^m)$ for cryptographic applications: performance - power consumption security tradeoffs*. PhD thesis, Silesian University of Technology; University of Rennes 1, Poland, France, 2012.

[24] D. Pamula and A. Tisserand. GF($2^m$) finite-field multipliers with reduced activity variations. In *Proc. 4th International Workshop on the Arithmetic of Finite Fields*, volume 7369 of *LNCS*, pages 152–167, Bochum, Germany, July 2012. Springer.

[25] F. Rodríguez-Henríquez, N. A. Saqib, and A. Díaz-Pérez. A fast parallel implementation of elliptic curve point multiplication over GF($2^{\mathrm{m}}$). *Microprocessors and Microsystems*, 28(5-6):329–339, 2004.

[26] P. Rohatgi. Defend encryption systems against side-channel attacks. EDN network, March 2015.

[27] S. Skorobogatov. Side-channel attacks: new directions and horizons. In *ECRYPT2 School on Design and Security of Cryptographic Algorithms and Devices*, Albena near Varna, Bulgaria, May 2011.

[28] G. Sutter, J. Deschamps, and J. Imana. Efficient Elliptic Curve Point Multiplication Using Digit-Serial Binary Field Operations. *IEEE Transactions on Industrial Electronics*, 60(1):217–225, Jan 2013.

[29] A. Tisserand. Fast and accurate activity evaluation in multipliers. In *Proc. 42nd Asilomar Conference on Signals, Systems and Computers*, pages 757–761. IEEE, Oct. 2008.