Binary-Ternary Plus-Minus Modular Inversion in RNS

Karim Bigou, Arnaud Tisserand

Binary-Ternary Plus-Minus Modular Inversion in RNS

Karim Bigou and Arnaud Tisserand, *Senior Member, IEEE*

***Abstract*—A fast RNS modular inversion for finite fields arithmetic has been published at CHES 2013 conference. It is based on the binary version of the plus-minus Euclidean algorithm. In the context of elliptic curve cryptography (*i.e.* 160–550 bits finite fields), it significantly speeds-up modular inversions. In this paper, we propose an improved version based on both radix 2 and radix 3. This new algorithm leads to 30 % speed-up for a maximal area overhead about 4 % on Virtex 5 FPGAs.**

***Index Terms*—Residue Number System, Modular Arithmetic, Extended Euclidean Algorithm, ECC, FPGA.**

I. Introduction

The origins of the *residue number system* (RNS) are very old (at least 2000 years ago). During the fifties, its modern use in computer arithmetic has been discussed in [33] and [19]. It uses a set of small *moduli* $(m_1, m_2, \ldots, m_n)$, called the RNS *base*, where the moduli are pairwise coprime integers. An integer X in RNS is split into n components $(x_1, x_2, \ldots, x_n)$, called the *residues* or *remainders*, with $x_i = X \bmod m_i$. Conversion from the standard representation to RNS is straightforward. Thanks to the *Chinese remainder theorem* (CRT), one can recover X in the classical representation from the residues (see for instance [34, Chap. 3]).

In RNS, computations are divided into independent *channels* with one modulo m_i per channel. Some operations are very efficient in RNS: addition, subtraction and multiplication are performed in *parallel* over the channels without carry propagation between them. Exact division can also be performed in parallel when the divisor is coprime with all the moduli.

However, sign/overflow detection, comparison, general division and modular reduction [2] operations are more complex in RNS and require a lot of precomputations. Furthermore, RNS is not directly supported by hardware description languages and computer aided design (CAD) tools (involving an important development and debug cost).

RNS has been used more recently to accelerate computations in asymmetric cryptography over *very large operands* for *RSA* (1024–4096 bits) [27], [4], [25], [30]; *elliptic curve cryptography* (ECC, 160–550 bits) [31], [26], [20]; *pairings* [14], [17]; and very recently *lattice based cryptography* [3]. Thanks to its *non-positional property*, RNS can be used to randomize the order of internal computations as a *protection* against some *side channel attacks* [5], [15], [29]. RNS has also been used as a protection against fault injection attacks with an additional channel dedicated to fault detection [15], [21].

K. Bigou[a] and A. Tisserand[b] are with IRISA laboratory CNRS[b], University Rennes 1[a] and INRIA Centre Rennes - Bretagne Atlantique in Lannion, France (contact: karim.bigou@irisa.fr or arnaud.tisserand@irisa.fr).

In RNS cryptographic hardware implementations, *modular inversion* was mainly implemented using *Fermat's little theorem* (FLT) [20], [7]. In our paper at CHES 2013 [10], we proposed a significantly faster RNS modular inversion based on the *binary extended Euclidean* algorithm and the *plus-minus* trick from [12].

In this paper, we propose a new mixed *binary-ternary* algorithm for RNS modular inversion. Adding small modulo 3 units, we can reduce the number of iterations by 1/3 compared to our binary solution from [10]. For typical ECC field sizes on a Virtex-5 FPGA, our new binary-ternary version leads to 30 % faster modular inversions with only 4 % area overhead compared to [10].

The paper is organized as follows. Section II defines our notations. Section III briefly describes state-of-the-art methods. Our new binary-ternary algorithm is detailed in Section IV and compared to state-of-the-art in Section V. Architecture and FPGA implementation are presented in Section VI. Finally, Section VII concludes the paper.

II. Notations and Definitions

Notations and definitions of some precomputations used in this paper are:

- P an ℓ-bit prime integer (for ECC $\ell \approx$ 160–550 bits).
- Capital letters, *e.g.* X, denote large integers or elements of $\mathbb{F}_P$.
- A the argument to be inverted and X, Y unspecified variables.
- $|X|_P$ denotes $X \bmod P$.
- n the number of moduli in the RNS base.
- m_i a w-bit modulo, $m_i = 2^w - r_i$ and $r_i < 2^{\lfloor w/2 \rfloor}$ (m_i is a pseudo Mersenne [16]).
- $\mathcal{B} = (m_1, \ldots, m_n)$ is the RNS base, where all m_i are pairwise coprime and all m_i are odd
- $\overrightarrow{X}$ represents X in RNS base $\mathcal{B}$, *i.e.* $\overrightarrow{X} = (x_1, \ldots, x_n)$ where $x_i = |X|_{m_i}$.
- $M = \prod_{i=1}^{n} m_i$
- $\overrightarrow{T_{\mathcal{B}}} = \left(\left| \frac{M}{m_1} \right|_{m_1}, \ldots, \left| \frac{M}{m_n} \right|_{m_n} \right)$
- FLT: Fermat's little theorem (*e.g.* [35, chap. 4.4])
- CRT: Chinese remainder theorem (*e.g.* [22, chap. 3.4])
- `EMM`: one elementary modular multiplication $(w \times w)$-bit
- `EMA`: one elementary modular addition $(w \pm w)$-bit
- MI: modular inversion
- PM: plus-minus trick from [12]
- BMI: RNS MI using binary PM algorithm from [10]
- BTMI: new RNS MI binary-ternary PM algorithm

III. State-of-the-Art

One can find comprehensive surveys on RNS in [34], [32] (mainly for signal processing applications). For RNS material related to MI in the context of asymmetric cryptography, one can refer, for instance, to [10, Sec. 3].

Efficient ECC implementations use projective coordinates to perform the scalar multiplication without intermediate MIs. But to convert back to a standard and unique representation (*e.g.* NIST [28]), at least one final MI is required. In RNS, this final MI usually represents around 10–15 % of the computation cost of a scalar multiplication on an elliptic curve (see [7] or [8, Sec. 2.1]). As stated in [14], pairings also require costly inversion. The authors of this paper report: "*[...] the remaining inversion in $\mathbb{F}_p$ is very expensive. Since comparison in RNS is difficult, inversion through exponentiation ($X^{-1} \equiv X^{p-2} \bmod p$) is used*".

In standard positional representations, MI is usually performed using the extended Euclidean algorithm or FLT. But in RNS, most of algorithms and implementations in the literature use FLT.

FLT states $|A^{P-1}|_P = 1$ with P a prime integer and A an integer coprime with P. Then, one deduces $|A^{P-2}|_P = |A^{-1}|_P$. FLT based MI performs the modular exponentiation $|A^{P-2}|_P$. For RSA, fast RNS modular exponentiations were proposed in [18] and [30]. For ECC, FLT-based RNS MI is used in [20] (which can be optimized using some tricks proposed later in [18]). The main advantage of FLT-based MI in RNS is that no specific inversion unit is required. Using a square-and-multiply algorithm, the complexity of the FLT-based RNS MI is $O(\ell \times n^2)$ `EMM`s (see [10] for details).

In non-RNS implementations, MI can be performed using the well known *extended Euclidean algorithm* [24]. An RNS version of the Euclidean algorithm has been proposed in [6] where quotients have been replaced by approximations. The complexity of this solution has been partially evaluated, and no implementation results are reported. Up to recently, the Euclidean algorithm for RNS MI was not popular since it requires comparisons which are costly operations in RNS.

Our work [10] circumvents this issue replacing comparisons by cheap modulo 4 tests and the plus-minus (PM) trick from [12]. Due to the use of modulo 4 tests, we call it the *binary PM modular inversion* (BMI). The complexity of this algorithm is only $O(\ell \times n)$ `EMM`s. On a Virtex-5 FPGA, BMI leads to 6 to 12 times faster MI than FLT-based solutions using a similar silicon area (see details in [10]). Since the publication of [10], we implemented our MI solutions with more parameters. Figure 1 reports a typical comparison of the MI based on FLT and BMI from [10] on a 384-bit field but for a larger set of parameters than our previous work (complete results for other field sizes and parameters are reported in the PhD document [8]). This figure shows the speedup obtained on FPGA and confirms that the cost of our BMI solution is only $O(\ell \times n)$ `EMM`s instead of $O(\ell \times n^2)$ for FLT.

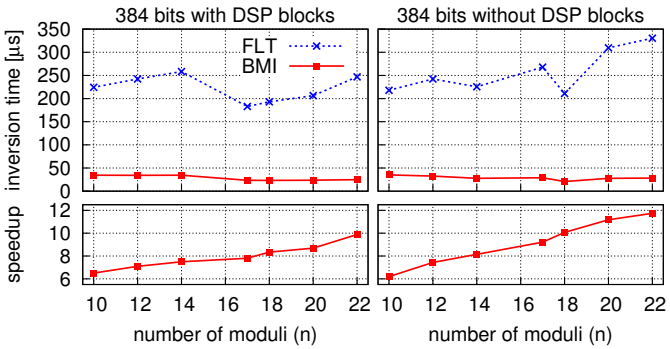

Fig. 1. Implementation results for FLT MI and BMI from [10] for 384 bits on Virtex 5 FPGA (more n parameters than [10]).

IV. Proposed Binary-Ternary Plus-Minus Algorithm

A. Main Objectives and Ideas

One modular inversion (MI) is required at the end of fast ECC scalar multiplication to convert back from projective coordinates. Similarly to the state-of-the-art solutions, we think that designing a dedicated MI unit is useless (such a unit would be idle about 85–90 % of time). In [10], we decided to slightly adapt the `Cox-Rower` architecture from state-of-the-art, originally proposed for RSA modular exponentiation in [23], [27], and optimized for ECC scalar multiplication in [20]. Hence, we mainly reused all the RNS operators used for the other operations (mainly additions, subtractions and multiplications over $\mathbb{F}_P$), adding very small computations (on 2-bit values).

In this section, we introduce a new RNS MI algorithm, called the *binary-ternary plus-minus* (BTMI) algorithm based on division and modulo operations by 3, 6 and 12 in addition to those by 2 and 4 of [10]. These modulo 3 tests enable us to significantly reduce the number of loop iterations with a very small area overhead.

B. Proposed algorithm: BTMI

Our new proposition is presented in Algo. 1. All the values are represented in an affine transformation $\widehat{X}$ of RNS, which is presented in Sec. IV-C.

As in [10], it is based on the plus-minus algorithm proposed initially for the binary standard representation in [12]. First, similarly to [10], we divide V_3 by 2 but also by 3 as much as possible using the `divup` function (see Sec. IV-D) and the functions `mod3` and `mod4` (see Sec. IV-C). Thus, after the inner loop at lines 5–6, values are neither even nor divisible by 3. Hence, we can perform a modulo 6 version of the plus-minus trick to avoid comparisons, which are hard and costly in RNS. If two integers X and Y are odd and not multiple of 3, then $X+Y$ or $X-Y$ is a multiple of 6. Values $X+Y$ and $X-Y$ are ensured to be even, we only have to test the value modulo 3 (at line 8) to choose the sum or the difference. Then we also check if the sum or the difference is a multiple of 4. It happens when $|X+Y|_4 = |X+Y|_6 = 0$ or $|X-Y|_4 = |X-Y|_6 = 0$ (with a probability of 0.5). Then, we divide by 6 (lines 16 and 25) or by 12 (lines 12 and 21). As the original plus-minus algorithm

and [10], one uses small values u and v to balance the number of divisions between U_3 and V_3 (instead of using comparisons as in the standard binary Euclidean algorithm). For instance, if V_3 is divided by 2, then one performs $v \leftarrow v+1$, $v \leftarrow v+2$ for a division by 4 and $v \leftarrow v+\sigma$ for a division by 3, where $\sigma \approx log_2(3)$ (in practice selecting $\sigma = 1.5$ is sufficient). Then, if $v > u$, one swaps U_3 and V_3 (lines 26–28). Finally, the lines 31–34 converts $\widehat{X}$ into $\overrightarrow{X}$.

Thanks to the use of higher divisors, BTMI requires less main loop iterations than BMI from [10]. But it requires more precomputations (10 RNS additional values). The cost of our new algorithm is analyzed in Sec. V, the number of EMMs is reduced by 30 % compared to our original BMI.

Adding additional modulo tests, such as 5 or 7, and the corresponding divisors is possible but with a high increase in the number of precomputed values. Moreover, the probability for an intermediate value to be a multiple of a 5 is $1/5$, for 7 it is $1/7$, etc. The same thing occurs for scalar recoding using multiple base number systems (MBNS, see for instance [13]). Then the overall gain may be reduced compared to the additional silicon and control cost.

C. Affine Transformation and Auxiliary Functions `mod3` and `mod4`

In Algo. 1, we need exact divisions and efficient modular reductions by 2, 3, 4, 6 and 12. To compute them efficiently, we propose to slightly modify the RNS representation during the execution of the modular inversion. This representation is a simple affine transformation defined by $\widehat{X} = (X + C_0)\overrightarrow{T_{\mathcal{B}}^{-1}}$, with $C_0 > P$ and $|C_0|_{12} = 0$, for instance $\overrightarrow{C_0} = \overrightarrow{12P}$. It is an adaptation of the one proposed in [10] but considering the fact that we need modulo 3 now. Conversion between $\overrightarrow{X}$ and $\widehat{X}$ only requires one RNS multiplication and one RNS addition. This conversion is performed only once at the beginning of the inversion and the opposite conversion is required at the end of the algorithm. This representation exactly behaves as standard RNS and can be directly mapped on the `Cox-Rower` architecture. It only requires a few additional bits (2 to 4 for our target finite fields).

To simplify the modulo computations (and the equations) we assume all (odd) moduli in $\mathcal{B}$ such that $|m_i|_{12} = 1$ *i.e.* $|m_i|_4 = 1$ and $|m_i|_3 = 1$. Without this small constraint, one has $|m_i|_4 = \pm 1$ and $|m_i|_3 = \pm 1$. To compute $|X|_4$ from $\widehat{X} = (\widehat{x_i})$, we use the reduction modulo 4 of the CRT (as in [10]):

$$|X|_4 = \left| \sum_{i=1}^{n} |\widehat{x_i}|_4 - |q|_4 \right|_4, \tag{1}$$

where $q = \left\lfloor \frac{\sum_{i=1}^{n} \widehat{x_i} \frac{M}{m_i}}{M} \right\rfloor$. We denote `mod4` the function which computes Eqn. (1). It evaluates the two terms $\left|\sum_{i=1}^{n} |\widehat{x_i}|_4\right|_4$ and $|q|_4$ and subtract them modulo 4.

To compute values modulo 3, we use a similar formula:

$$|X|_3 = \left| \sum_{i=1}^{n} |\widehat{x_i}|_3 - |q|_3 \right|_3. \tag{2}$$

Algorithm 1: Proposed Binary-Ternary Plus-Minus RNS Modular Inversion (BTMI).

Input: $\overrightarrow{A}, P > 2$ with $\gcd(A, P) = 1$
Precomp.: $\overrightarrow{T_{\mathcal{B}}}, \overrightarrow{T_{\mathcal{B}}^{-1}}, \overrightarrow{C_1}$
Result: $\overrightarrow{S} = |A^{-1}|_P$, $0 \leqslant S < 2P$

1 $u \leftarrow 0$, $v \leftarrow 0$, $\widehat{U_1} \leftarrow \widehat{0}$, $\widehat{V_1} \leftarrow \widehat{1}$, $\widehat{V_3} \leftarrow \widehat{A}$
2 $b_{U_1} \leftarrow 0$, $t_{U_1} \leftarrow 0$, $b_{U_3} \leftarrow |P|_4$, $t_{U_3} \leftarrow |P|_3$
3 $b_{V_1} \leftarrow 1$, $t_{V_1} \leftarrow 1$, $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$
4 **while** $\widehat{V_3} \neq \widehat{1}$ **and** $\widehat{U_3} \neq \widehat{1}$ **and** $\widehat{V_3} \neq \widehat{-1}$ **and** $\widehat{U_3} \neq \widehat{-1}$ **do**
5 **while** $|b_{V_3}|_2 = 0$ **or** $t_{V_3} = 0$ **do**
6 $\texttt{divup}\left(\widehat{V_3}, b_{V_3}, t_{V_3}, \widehat{V_1}, b_{V_1}, t_{V_1}, v\right)$
7 $\widehat{V_3^*} \leftarrow \widehat{V_3}$, $\widehat{V_1^*} \leftarrow \widehat{V_1}$
8 **if** $|t_{V_3} + t_{U_3}|_3 = 0$ **then**
9 **if** $|b_{V_3} + b_{U_3}|_4 = 0$ **then**
10 $r = 1$
11 $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C_1}, 0, 0)$
12 $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C_1}, |b_{V_1} + b_{U_1}|_4, |t_{V_1} + t_{U_1}|_3)$
13 **else**
14 $r = 0$
15 $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}(\widehat{V_3} + \widehat{U_3} - \overrightarrow{C_1}, 2, 0)$
16 $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}(\widehat{V_1} + \widehat{U_1} - \overrightarrow{C_1}, |b_{V_1} + b_{U_1}|_4, |t_{V_1} + t_{U_1}|_3)$
17 **else**
18 **if** $|b_{V_3} - b_{U_3}|_4 = 0$ **then**
19 $r = 1$
20 $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C_1}, 0, 0)$
21 $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C_1}, |b_{V_1} - b_{U_1}|_4, |t_{V_1} - t_{U_1}|_3)$
22 **else**
23 $r = 0$
24 $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}(\widehat{V_3} - \widehat{U_3} + \overrightarrow{C_1}, 2, 0)$
25 $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}(\widehat{V_1} - \widehat{U_1} + \overrightarrow{C_1}, |b_{V_1} - b_{U_1}|_4, |t_{V_1} - t_{U_1}|_3)$
26 **if** $v > u$ **then**
27 $\widehat{U_3} \leftarrow \widehat{V_3^*}$, $b_{U_3} \leftarrow b_{V_3}$, $t_{U_3} \leftarrow t_{V_3}$
28 $\widehat{U_1} \leftarrow \widehat{V_1^*}$, $b_{U_1} \leftarrow b_{V_1}$, $t_{U_1} \leftarrow t_{V_1}$, $\texttt{swap}(u, v)$
29 $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$
30 $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$, $v \leftarrow v + r + \sigma$
31 **if** $\widehat{V_3} = \widehat{1}$ **then return** $(\widehat{V_1} - \overrightarrow{C_1})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
32 **else if** $\widehat{U_3} = \widehat{1}$ **then return** $(\widehat{U_1} - \overrightarrow{C_1})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
33 **else if** $\widehat{V_3} = \widehat{-1}$ **then return** $-(\widehat{V_1} - \overrightarrow{C_1})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$
34 **else return** $-(\widehat{U_1} - \overrightarrow{C_1})\overrightarrow{T_{\mathcal{B}}} + \overrightarrow{P}$

However, modulo 3 reduction on $\widehat{x_i}$ in the channels are w-bit operators (see for instance [9]) and not only 2 LSBs as for modulo 4 because inside the channels values use the binary representation. This leads to very small extra hardware resources for each channel (see Sec. VI). The function `mod3` computes Eqn. (2).

To compute mod3 and mod4, the value $q = \left\lfloor \frac{\sum_{i=1}^{n} \widehat{x_i} \frac{M}{m_i}}{M} \right\rfloor$ must be computed. As in fast state-of-the-art ECC implementations in RNS, we use a small dedicated unit, called Cox unit to compute an approximation q' of q, see [23]. This unit is required to compute efficient modular multiplication in asymmetric cryptographic implementation in RNS. The Cox sums up the t most significant bits of each residue, here $\widehat{x_i}$. Typically, state-of-the-art solutions use $t \in [4, 6]$ (see [23] for details). The Cox approximation q' proposed in [23] only works for positive intermediate values. Using the plus-minus trick, intermediate values are in the interval $]-P, P[$. Using our $\widehat{X}$ representation, values are now in $]C_0 - P, C_0 + P[$, for instance $]11P, 13P[$ for $\overrightarrow{C_0} = \overrightarrow{12P}$. Using the results from [23], we set $t = 6$ to ensure $q' = q$ for all considered field sizes and parameters. The parameter $t = 6$ is required for the largest fields, but one can use $t = 4$ or $t = 5$ for the smaller ones. The cost of the evaluation of q is n additions of t-bit values [23]. This computation is shared for both mod4 and mod3 computations.

The computation dedicated to mod4 costs $n + 2$ additions modulo 4 (2-bit values) to evaluate Eqn. 1, which are actually negligible compared to the RNS additions and multiplications performed during the inversion algorithm. The one dedicated to mod3 costs $n + 2$ additions modulo 3. Additions modulo 3 are additions on 2-bit values and are also negligible. Reductions modulo 3 of w-bit values only require very small hardware units (see Sec. VI and [9]), then we can neglect them compared to RNS operations (RNS additions and multiplications).

D. Auxiliary Functions $\widehat{\texttt{divD}}$ and divup

In the BMI algorithm from [10], we just need to perform divisions by 2 or 4 modulo P for the BMI. For our new BTMI Algo. 1, more cases are required (divisions by 3, 6, 12 are added), thus we present how to perform all these divisions and modulo using a function called divup.

In RNS, an exact division by D can be performed through a multiplication by $|D^{-1}|_M$ when D and M are co-prime, the condition is to precompute $|D^{-1}|_M$ (in RNS the reduction by the product of moduli M is automatic). For the modular inversion algorithm, the divisions are actually performed modulo P (and not modulo M). But, when X is a multiple of D, then $X/D = |XD^{-1}|_M \equiv |XD^{-1}|_P$. We recall that we select M such that $\gcd(12, M) = 1$, P is prime; and in RNS $M > X$ to be able to represent X, thus $X/D < M$.

However, X is not always a multiple of D (where $D \in \{2, 3, 4, 6, 12\}$) in our modular inversion algorithm. To solve this issue, one can compute $(X + f_D(X)P)/D$ with $f_D(X) \equiv -XP^{-1} \bmod D$, in other words $X + f_D(X)P$ is a multiple of D and $X + f_D(X)P \equiv X \bmod P$. Actually $f_D(X)$ is computed from the value $|X|_D$ thanks to mod3 or mod4 functions, but we choose to simplify the writing of $f_D(|X|_D)$ by $f_D(X)$. As an example, we report the values of f_D for $D = 2, 3, 4, 6$ and $|P|_6 = 1$ in Table I. If we choose $f_D(X) \in [0, D-1]$, one must store $(D-2)P$ values (0 and P do not need extra storage). Choosing $f_D(X) \in [-\lceil \frac{D}{2} \rceil + 1, \lceil \frac{D}{2} \rceil - 1]$ enables to divide by 2 the number of stored values, but it can introduce negative intermediate values which must be managed by choosing $C_0 > \left(\lceil \frac{D}{2} \rceil - 1\right)P$, *i.e.* $C_0 > 5P$ for $D = 12$ (for instance $C_0 = 12P$) in the $\widehat{X}$ representation.

To formally define the division function, called $\widehat{\texttt{divD}}$, we denote $\overrightarrow{C_1} = \overrightarrow{C_0 T_{\mathcal{B}}^{-1}}$ and $\overrightarrow{P_1} = \overrightarrow{P T_{\mathcal{B}}^{-1}}$, and thus $\widehat{X} = \overrightarrow{X T_{\mathcal{B}}^{-1} + C_1}$. Then

$$\widehat{\texttt{divD}}(\widehat{X}, D, |X|_D) = \frac{\widehat{X}}{D} + f_D(X)\frac{\overrightarrow{P_1}}{D} + \frac{\overrightarrow{(D-1)C_1}}{D}. \quad (3)$$

In Eqn. 3, we recall that $|X|_D$ is required to compute $f_D(X)$. More, the last term $\frac{\overrightarrow{(D-1)C_1}}{D}$ is added to ensure that the result is in the $\widehat{X}$ representation because

$$\frac{\widehat{X}}{D} + \frac{\overrightarrow{(D-1)C_1}}{D} = \frac{\overrightarrow{X T_{\mathcal{B}}^{-1}}}{D} + \frac{\overrightarrow{C_1}}{D} + \frac{\overrightarrow{(D-1)C_1}}{D} = \widehat{(X/D)}.$$

To speed up the computations, we can precompute the possible combinations of the constants $f_D(X)\frac{\overrightarrow{P_1}}{D} + \frac{\overrightarrow{(D-1)C_1}}{D}$. Then, $\widehat{\texttt{divD}}$ costs one RNS multiplication and one addition by a constant in each channel, *i.e.* n EMMs $+n$ EMAs.

We present in Algo. 2 the function divup, which uses mod3, mod4 and $\widehat{\texttt{divD}}$ for the various D to manage all possible cases in the inversion algorithm. This function performs the division (modulo P) by the greatest divisor possible, according to the values modulo 3 (*e.g.* t_{V_3}) and 4 (*e.g.* b_{V_3}). After the division, we compute mod3 for control values t_{V_3} and t_{V_1}, and mod4 for control values b_{V_3} and b_{V_1}. Finally, in order to update u and v, we need to evaluate the number of "suppressed" bits during the divisions by 3, we use σ an approximation of $\log_2(3) \approx 1.5849$. In practice, using $\sigma = 1.5$ is accurate enough for our target finite fields.

To optimize the function divup, we can limit the number of calls to mod3 and mod4 functions. For instance, after a division by 3, it is not necessary to use the function mod4, we can directly get the new value modulo 4 from b_X multiplying by $3^{-1} \bmod 4 = 3$ and adding a small specific precomputed value. Formally, defining $\underline{b_X} \in \{-1, 0, 1, 2\}$ and $\underline{t_X} \in \{-1, 0, 1\}$ such that $\underline{b_X} \equiv b_X \bmod 4$ and $\underline{t_X} \equiv t_X \bmod 3$ respectively, we define

$$\texttt{update4}(b_X, t_X) = \left|3\ \left(b_X + (2|P|_3 - 3)\ \underline{t_X}\ P\right)\right|_4,$$

$$\texttt{update3\_2}(t_X, b_X) = \left|2\ \left(t_X + |\underline{b_X}|_2\ P\right)\right|_3 \text{ and}$$

$$\texttt{update3\_4}(t_X, b_X, b_Y) = \left|\left(t_X + (|P|_4 - 2)\ \underline{b_X}\ P\right)\right|_3.$$

These functions compute the new values modulo 3 or 4, without mod3 or mod4. In practice, these equations are simplified because P is fixed.

TABLE I
DEFINITION OF FUNCTIONS f_D FOR $D = 2, 3, 4, 6$ AND $|P|_6 = 1$.

$\lvert X\rvert_6$	0	1	2	3	4	5
$f_2(X)$	0	1	0	1	0	1
$f_3(X)$	0	-1	1	0	-1	1
$f_4(X)$	0	-1	2	1	0	-1
$f_6(X)$	0	-1	-2	3	2	1

Algorithm 2: divup Function.

Input: $\widehat{V_3}$, b_{V_3}, t_{V_3}, $\widehat{V_1}$, b_{V_1}, t_{V_1},v

Precomp.: $\alpha\overrightarrow{C_1}$ and $\beta(PT_{\mathcal{B}}^{-1})$ with $\alpha \in \{\frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{5}{6}, \frac{11}{12}\}$ and $\beta \in \{\frac{1}{12}, \frac{1}{6}, \frac{1}{4}, \frac{1}{3}, \frac{5}{12}, \frac{1}{2}, \frac{-5}{12}, \frac{-1}{3}, \frac{-1}{4}, \frac{-1}{6}, \frac{-1}{12}\}$

Output: $\widehat{V_3}$, b_{V_3}, t_{V_3}, $\widehat{V_1}$, b_{V_1}, t_{V_1},v

1 **case** $b_{V_3} = 0$ **and** $t_{V_3} = 0$

2 $\widehat{V_3} \leftarrow \widehat{\texttt{div12}}\left(\widehat{V_3}, b_{V_3}, t_{V_3}\right)$, $\widehat{V_1} \leftarrow \widehat{\texttt{div12}}\left(\widehat{V_1}, b_{V_1}, t_{V_1}\right)$ $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$, $v \leftarrow v + \sigma + 2$

3 **case** $b_{V_3} = 2$ **and** $t_{V_3} = 0$

4 $\widehat{V_3} \leftarrow \widehat{\texttt{div6}}\left(\widehat{V_3}, b_{V_3}, t_{V_3}\right)$, $\widehat{V_1} \leftarrow \widehat{\texttt{div6}}\left(\widehat{V_1}, b_{V_1}, t_{V_1}\right)$ $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$, $v \leftarrow v + \sigma + 1$

5 **case** $b_{V_3} = 0$ **and** $t_{V_3} \neq 0$

6 $\widehat{V_3} \leftarrow \widehat{\texttt{div4}}\left(\widehat{V_3}, b_{V_3}\right)$,$\widehat{V_1} \leftarrow \widehat{\texttt{div4}}\left(\widehat{V_1}, b_{V_1}\right)$, $t_{V_3} \leftarrow \texttt{update3\_4}(t_{V_3}, b_{V_3})$, $t_{V_1} \leftarrow \texttt{update3\_4}(t_{V_1}, b_{V_1})$, $v \leftarrow v + 2$, $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$

7 **case** $b_{V_3} \neq 0, 2$ **and** $t_{V_3} = 0$

8 $\widehat{V_3} \leftarrow \widehat{\texttt{div3}}\left(\widehat{V_3}, t_{V_3}\right)$, $\widehat{V_1} \leftarrow \widehat{\texttt{div3}}\left(\widehat{V_1}, t_{V_1}\right)$, $b_{V_3} \leftarrow \texttt{update4}(b_{V_3}, t_{V_3})$, $b_{V_1} \leftarrow \texttt{update4}(b_{V_1}, t_{V_1})$, $v \leftarrow v + \sigma$, $t_{V_3} \leftarrow \texttt{mod3}(\widehat{V_3})$, $t_{V_1} \leftarrow \texttt{mod3}(\widehat{V_1})$

9 **case** $b_{V_3} = 2$ **and** $t_{V_3} \neq 0$

10 $\widehat{V_3} \leftarrow \widehat{\texttt{div2}}\left(\widehat{V_3}, b_{V_3}\right)$, $\widehat{V_1} \leftarrow \widehat{\texttt{div2}}\left(\widehat{V_1}, b_{V_1}\right)$, $t_{V_3} \leftarrow \texttt{update3\_2}(t_{V_3}, b_{V_3})$, $t_{V_1} \leftarrow \texttt{update3\_2}(t_{V_1}, b_{V_1})$, $v \leftarrow v + 1$, $b_{V_3} \leftarrow \texttt{mod4}(\widehat{V_3})$, $b_{V_1} \leftarrow \texttt{mod4}(\widehat{V_1})$

V. Comparison to State-of-the-Art

In this section, we compare our BTMI algorithm to the best state-of-the-art algorithm (BMI from [10]). The comparison focuses on full RNS operations (additions/multiplications), and not on very small operations specific to mod3 reductions. These small reductions are performed in parallel using very small hardware resources, as presented in Sec. VI. Then, the cost of mod3 and mod4 are neglected in this analysis.

Each algorithm BMI and BTMI mainly performs a loop with $O(l)$ iterations. First we compare the cost of one outer loop iteration of each algorithm (the structures of both algorithms are very similar).

In BMI [10], the cost of one inner loop iteration is 2 $\widehat{\texttt{divD}}$ leading to $2n$ EMMs and $2n$ EMAs. There is on average $2/3$ inner loop iteration per outer loop iteration. After the inner loop, the plus-minus trick is performed with 2 $\widehat{\texttt{divD}}$ and 2 RNS additions. Thus each outer loop iteration costs on average $3.33n$ EMMs and $5.33n$ EMAs. For a ℓ-bit field, there are 0.71ℓ outer loop iterations, leading to $2.37n\ell$ EMMs and $3.79n\ell$ EMAs on average.

For the BTMI algorithm proposed in this paper, one performs one divup per inner loop iteration, leading to 2 $\widehat{\texttt{divD}}$ and $2n$ EMMs and $2n$ EMAs. On average there is 0.75 inner loop iteration per outer loop iteration (we tested more than 700 000 simulations). After the inner loop, 2 $\widehat{\texttt{divD}}$ and 2 RNS additions are performed leading to an average cost of $3.5n$ EMMs and $5.5n$ EMAs for each outer loop iteration. Thus a loop iteration of our algorithm is slightly more costly than BMI. However, the number of outer loop iterations is reduced: one only has 0.46ℓ outer loop iterations on average. Then the total average cost of the outer loop is $1.61n\ell$ EMMs and $2.53n\ell$ EMAs.

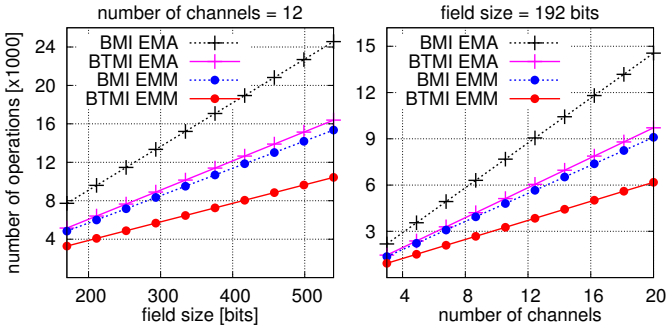

Fig. 2. BMI vs BTMI comparison of the theoretical number of EMMs and EMAs for n fixed (left) or ℓ fixed (right).

Fig. 2 graphically compares the theoretical complexities for BMI and BTMI for ℓ fixed (and n variable) or ℓ variable (and n fixed). On average, BTMI reduces by 30 % the number of EMMs and EMAs compared to BMI but it increases the number of precomputations by a factor 2.2 (but current FPGAs embed large enough BRAMs to hide this overhead) for ECC applications.

VI. Architecture and FPGA Implementation

We use the Cox–Rower architecture for modular multiplications originally proposed in [23] (in the RSA context) and optimized in several RNS papers (see for instance [20], [14] in the ECC context). As in our previous paper [10], we do not implement a dedicated modular inversion unit (it would be idle about 85–90 % of time). We prefer to slightly modify the Cox–Rower architecture to support also modular inversions for silicon efficiency purpose. Our modifications do not reduce the speed of other RNS operations during scalar multiplication.

Our new overall Cox–Rower architecture supporting BTMI is depicted in Fig. 3. This architecture is an extension of the BMI one presented in [10]. Control signals, clock signal and reset ones are just partially represented in Fig. 3. Short lines terminated by white circles (*i.e.* —○) represent control signals.

The computations on w-bit residues modulo m_i in each channel are performed in a dedicated Rower unit. In this work, we use a full parallel architecture with n Rowers for n channels similarly to state-of-the-art solutions. Our new Rower architecture is detailed in Fig. 4.

There are two main differences between the architecture for the binary version (BMI, published in [10]) and our new binary-ternary version BTMI: i) each Rower has to compute the residue modulo 3 of the x_i and transfer this 2-bit value to the Cox unit; ii) additional precomputations are required to handle modulo 3 cases.

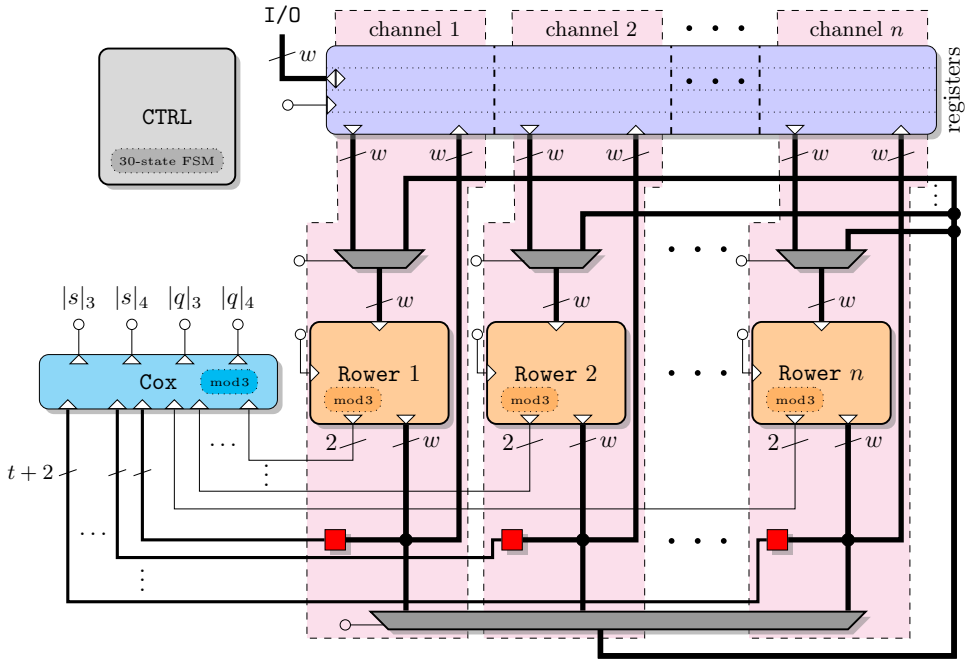

Fig. 3. Global Cox–Rower architecture adapted to our BTMI algorithm.

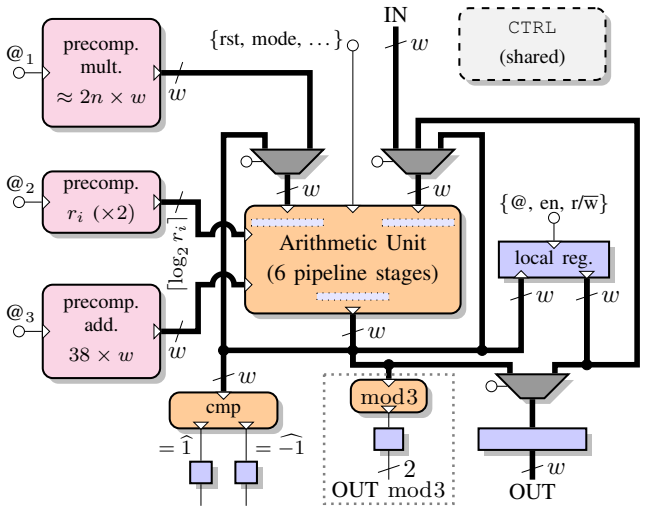

Fig. 4. Rower architecture for BTMI.

Each Rower contains one arithmetic unit, a few local registers and small ROMs (read only memories) for precomputed values. In each Rower, a small unit, in the dotted region in Fig. 4, computes the modulo 3 reduction $|\widehat{x_i}|_3$ of the output residue of the arithmetic unit. Modulo 3 reduction is a very small combinatorial unit. Tab. II presents its implementation results on Xilinx Virtex 5 LX220 FPGA for various sizes w. This very small additional unit is not on the critical path of the Rower, then it does not reduce the overall architecture frequency. The new additive precomputations required for divup are stored in the local "precomp. add." memory (bottom left memory in Fig. 4). In BTMI, 38 words of w bits are required while only 17 were used in BMI. The new multiplicative precomputations are stored in the "precomp. mult." memory

TABLE II
IMPLEMENTATION RESULTS OF w-BIT MODULO 3 UNITS ON VIRTEX 5.

w (bits)	17	20	22	24	29	33	36
Area (slices)	6	6	6	6	10	10	11
Freq. (MHz)	275	230	221	218	162	161	170

(top left in Fig. 4) for values $\frac{1}{3}$, $\frac{1}{6}$, and $\frac{1}{12}$, leading to $2n+7$ w-bit words per Rower instead of $2n+4$ in [10]. In our target FPGAs, those additional precomputations still fit the BRAMs (36Kb each in Virtex 5 FPGAs), then there is no area overhead at this level.

In Cox–Rower architecture in Fig. 3, the small (red) squares just select the t MSBs and 2 LSBs of the w-bit output residue of each Rower (which is just routing) to compute the global q value (see Sec. IV-C and [10]). In the BTMI architecture, new 2-bit wires transfer $|\widehat{x_i}|_3$ from each Rower to the Cox unit. All these 2-bit values are summed up and reduced modulo 3 in the Cox to produce the value $|s|_3 = \left|\sum_{i=1}^{n} |\widehat{x_i}|_3\right|_3$.

Tab. III summarizes the new hardware resources for BTMI compared to BMI from [10] in Rowers, Cox and global Cox–Rower control. The finite state machine in the global Cox–Rower control is increased by few states, and there are now 8 control values of 2 bits (corresponding to b_{V_i}, b_{U_i}, t_{V_i} and t_{U_i} with $i \in \{1, 3\}$) due to modulo 3 units.

We estimated the area overhead and speedup for our new BTMI architecture compared to BMI one for several sets of parameters (field size ℓ, number of channels n and channel width w). For area overhead, Tab. IV reports typical obtained values, the worst case is at most 4 %. For speedup, Fig. 5

TABLE III
SUMMARY OF CHANGES BETWEEN OF OUR PREVIOUS BMI ARCHITECTURE AND OUR NEW BTMI ON VIRTEX 5 FPGA.

Unit	Sub-unit	BMI	BTMI
Rower	mod3	0	6–11 slices
	Mem. (w-bit words)	17	38
	BRAM	1 or 2	1 or 2
Cox	mod3	0	6–11 slices
CTRL	FSM	25 states	30 states
	2-bit values	4	8

TABLE IV
TYPICAL GLOBAL AREA OVERHEAD OF BTMI COMPARED TO BMI IMPLEMENTATIONS ON VIRTEX 5 FPGA.

field size ℓ (bits)	192	256	384	521
min overhead (%)	2.9	3.1	2.9	2.1
max overhead (%)	3.6	3.4	3.1	2.5

compares the execution time of a full modular inversion on Virtex 5 FPGA for FLT, BMI and BTMI architectures on various field sizes. FLT architecture is far slower than Euclidean based architectures with binary BMI from [10] and our new binary-ternary BTMI. BTMI is 30 % faster than BMI since the number of operations is reduced by 30 % and the clock frequency is maintained.

For the validation of both BTMI algorithm and architecture, we used the same strategy than [10] for BMI. Proving at high level the correctness of Euclidean based algorithms, as our BMI and BTMI, is not very difficult since $V_1 A \equiv V_3 \bmod P$ and $U_1 A \equiv U_3 \bmod P$ are always true (see for instance [24, Sec 4.5.2]). But formally proving the average number of loop iterations is highly difficult (see for instance [1]). We used intensive simulations to estimate the actual number of loop iterations. Those simulations were performed using a computer algebra system (Maple 15) for all algorithms over 700 000 random operands for P-192, P-256, P-384 and P-521 primes from [28].

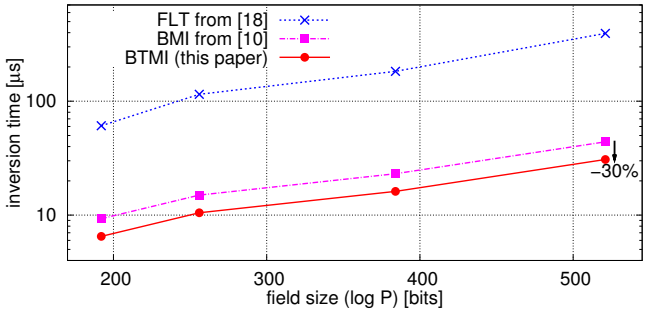

Fig. 5. Comparison of RNS MI timings on Virtex-5 FPGA for various field sizes and algorithms (FLT and BMI from [10] and the proposed BTMI.)

VII. CONCLUSION

A new fast RNS modular inversion algorithm has been proposed. This algorithm is based on the binary extended Euclidean algorithm, as the state-of-the-art one proposed in [10]. In the new algorithm, the number of iterations in the main loop is reduced compared to [10], reducing the number of operations by 30 %, thanks to cheap modulo 3 tests and divisions by $\{2, 3, 4, 6, 12\}$. The added hardware resources for modulo 3 computations are very small and do not reduce the frequency. The area overhead is only 2–4 % (depending on the target finite field size) with a 30 % speed-up compared to the architecture from [10] on Virtex 5 FPGAs.

We plan to fully implement our new algorithm on a new architecture, with configurable RNS operators which take benefits from the very recent results of [11] for ECC in RNS.

ACKNOWLEDGMENT

This work has been supported in part by a *DGA–INRIA* PhD grant and by the PAVOIS project (ANR 12 BS02 002 01). We also thank the anonymous Reviewers for their valuable comments.

REFERENCES

[1] A. Akhavi and B. Vallee. Average bit-complexity of euclidean algorithms. In *Proc. 27th International Colloquium Automata, Languages and Programming (ICALP)*, volume 1853 of *LNCS*, pages 373–387. Springer, July 2000.

[2] J.-C. Bajard, L.-S. Didier, and P. Kornerup. An RNS Montgomery modular multiplication algorithm. *IEEE Transactions on Computers*, 47(7):766–776, July 1998.

[3] J.-C. Bajard, J. Eynard, N. Merkiche, and T. Plantard. RNS arithmetic approach in lattice-based cryptography. accelerating the "rounding-off" core procedure. In *Proc. 22nd International Symposium on Computer Arithmetic (ARITH)*, pages 113–120. IEEE, June 2015.

[4] J.-C. Bajard and L. Imbert. A full RNS implementation of RSA. *IEEE Transactions on Computers*, 53(6):769–774, June 2004.

[5] J.-C. Bajard, L. Imbert, P.-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Proc. Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *LNCS*, pages 62–75. Springer, 2004.

[6] J.-C. Bajard, N. Meloni, and T. Plantard. Study of modular inversion in RNS. In *Proc. Advanced Signal Processing Algorithms, Architectures, and Implementations XV*, volume 5910, pages 247–255. SPIE, July 2005.

[7] J.-C. Bajard and N. Merkiche. Double level Montgomery Cox-Rower architecture, new bounds. In *Proc. 13th Smart Card Research and Advanced Application Conference (CARDIS)*, volume 8968 of *LNCS*, pages 139–153. Springer, November 2014.

[8] K. Bigou. *Étude théorique et implantation matérielle d'unités de calcul en représentation modulaire des nombres pour la cryptographie sur courbes elliptiques*. Phd thesis, University Rennes 1, November 2014.

[9] K. Bigou, T. Chabrier, and A. Tisserand. Opérateur matériel de tests de divisibilité par des petites constantes sur de très grands entiers. In *Proc. 15ème Symposium en Architectures nouvelles de machines (SympA)*, January 2013. In French.

[10] K. Bigou and A. Tisserand. Improving modular inversion in RNS using the plus-minus method. In *Proc. 15th Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 8086 of *LNCS*, pages 233–249. Springer, August 2013.

[11] K. Bigou and A. Tisserand. Single base modular multiplication for efficient hardware RNS implementations of ECC. In *Proc. 17th International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 9293 of *LNCS*, pages 123–140. Springer, September 2015.

[12] R. P. Brent and H. T. Kung. Systolic VLSI arrays for polynomial GCD computation. *IEEE Transactions on Computers*, C-33(8):731–736, August 1984.

[13] T. Chabrier and A. Tisserand. On-the-fly multi-base recoding for ECC scalar multiplication without pre-computations. In *Proc. 21st Symposium on Computer Arithmetic (ARITH)*, pages 219–228. IEEE, April 2013.

[14] R. C. C. Cheung, S. Duquesne, J. Fan, N. Guillermin, I. Verbauwhede, and G. X. Yao. FPGA implementation of pairings using residue number system and lazy reduction. In *Proc. 13th Cryptographic Hardware and Embedded Systems (CHES)*, volume 6917 of *LNCS*, pages 421–441. Springer, September 2011.

[15] M. Ciet, M. Neve, E. Peeters, and J.-J. Quisquater. Parallel FPGA implementation of RSA with residue number systems – can side-channel threats be avoided? –. In *Proc. 46th Midwest Symposium on Circuits and Systems (MWSCAS)*, volume 2, pages 806–810. IEEE, December 2003.

[16] R. E. Crandall. Method and apparatus for public key exchange in a cryptographic system. US Patent 5159632 A, October 1992.

[17] S. Duquesne. RNS arithmetic in $\mathbb{F}_p^k$ and application to fast pairing computation. *Journal of Mathematical Cryptology*, 5:51–88, June 2011.

[18] F. Gandino, F. Lamberti, G. Paravati, J.-C. Bajard, and P. Montuschi. An algorithmic and architectural study on Montgomery exponentiation in RNS. *IEEE Transactions on Computers*, 61(8):1071–1083, August 2012.

[19] H. L. Garner. The residue number system. *IRE Transactions on Electronic Computers*, EC-8(2):140–147, June 1959.

[20] N. Guillermin. A high speed coprocessor for elliptic curve scalar multiplications over $\mathbb{F}_p$. In *Proc. 12th Cryptographic Hardware and Embedded Systems (CHES)*, volume 6225 of *LNCS*, pages 48–64. Springer, August 2010.

[21] N. Guillermin. A coprocessor for secure and high speed modular arithmetic. Technical Report 354, Cryptology ePrint Archive, 2011.

[22] K. Ireland and M. Rosen. *A Classical Introduction to Modern Number Theory*, volume 84 of *Graduate Texts in Mathematics*. Springer, 2nd edition, 1990.

[23] S. Kawamura, M. Koike, F. Sano, and A. Shimbo. Cox-Rower architecture for fast parallel Montgomery multiplication. In *Proc. 19th International Conference on the Theory and Application of Cryptographic (EUROCRYPT)*, volume 1807 of *LNCS*, pages 523–538. Springer, May 2000.

[24] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 3rd edition, 1997.

[25] Z. Lim and B. J. Phillips. An RNS-enhanced microprocessor implementation of public key cryptography. In *Proc. 41th Asilomar Conference on Signals, Systems and Computers*, pages 1430–1434. IEEE, November 2007.

[26] Z. Lim, B. J. Phillips, and M. Liebelt. Elliptic curve digital signature algorithm over GF(p) on a residue number system enabled microprocessor. In *Proc. IEEE Region 10 Conference (TENCON)*, pages 1–6, January 2009.

[27] H. Nozaki, M. Motoyama, A. Shimbo, and S. Kawamura. Implementation of RSA algorithm based on RNS Montgomery multiplication. In *Proc. 3rd Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, volume 2162 of *LNCS*, pages 364–376. Springer, May 2001.

[28] National Institute of Standards and Technology (NIST). FIPS 186-2, digital signature standard (DSS), 2000.

[29] G. Perin, L. Imbert, L. Torres, and P. Maurine. Electromagnetic analysis on RSA algorithm based on RNS. In *Proc. 16th Euromicro Conference on Digital System Design (DSD)*, pages 345–352. IEEE, September 2013.

[30] D. Schinianakis and T. Stouraitis. Multifunction residue architectures for cryptography. *IEEE Transactions on Circuits and Systems I*, 61(4):1156–1169, April 2014.

[31] D. M. Schinianakis, A. P. Fournaris, H. E. Michail, A. P. Kakarountas, and T. Stouraitis. An RNS implementation of an $\mathbb{F}_p$ elliptic curve point multiplier. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 56(6):1202–1213, June 2009.

[32] M. Soderstrand, W. K. Jenkins, G. Jullien, and F. Taylor, editors. *Residue Number System Arithmetic - Modern Applications in Digital Signal Processing*. IEEE, 1986.

[33] A. Svoboda and M. Valach. Operátorové obvody (operator circuits in czech). *Stroje na Zpracování Informací (Information Processing Machines)*, 3:247–296, 1955.

[34] N. S. Szabo and R. I. Tanaka. *Residue arithmetic and its applications to computer technology*. McGraw-Hill, 1967.

[35] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.