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 \(x_i = X \mod m_i\). 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 [3] 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 and A. Tisserand are with IRISA laboratory CNRS, University Rennes 1 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 \(\ell\)-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 \(X \mod P\).
- \(n\) the number of moduli in the RNS base.
- \(m_i\) a \(w\)-bit modulo, \(m_i = 2^w - r_i\) and \(r_i < 2^{w/2}\) (\(m_i\) is a pseudo Mersenne [16]).
- \(B = (m_1, \ldots, m_n)\) is the RNS base, where all \(m_i\) are pairwise coprime and all \(m_i\) are odd.
- \(\overrightarrow{X}\) represents \(X\) in RNS base \(B\), i.e. \(\overrightarrow{X} = (x_1, \ldots, x_n)\)

\(M = \prod_{i=1}^n m_i\)

\(\overrightarrow{T_B} = \left(\frac{M}{m_1}, \ldots, \frac{M}{m_n}\right)\)

- 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]
- 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 [12], pairings also require costly inversion. The authors of this paper report: “[…] the remaining inversion in \( F_p \) is very expensive. Since comparison in RNS is difficult, inversion through exponentiation \( (X^{-1} \equiv X^{p-2} \mod p) \) 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 [10]. 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 \( O(\ell \times n^2) \) EMMs (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 [9] 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 \( O(\ell \times n) \) EMMs. 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 \( O(\ell \times n) \) EMMs instead of \( O(\ell \times n^2) \) for FLT.

![Speedup in Inversion Time](image)

**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 \( F_p \)), 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 \( X \) 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 \( V \) by 2 but also by 3 as much as possible using the divup function (see Sec. IV-D) and the functions \( \text{mod}3 \) and \( \text{mod}4 \) (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 $X$ into $X'$. 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 $\mod 3$ and $\mod 4$

In Algo. [11] 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 $X = (X + C_0)T_8^{-1}$, with $C_0 > P$ and $|C_0|_{12} = 0$, for instance $C_0 = 12P$. It is an adaptation of the one proposed in [10] but considering the fact that we need modulo 3 now. Conversion between $X$ and $X'$ 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–Rowe 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 $\mathbb{B}$ such that $|m_i|_{12} = 1 \text{ i.e. } |m_i|_4 = 1$ 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 $X = (\hat{x}_i)$, we use the reduction modulo 4 of the CRT (as in [10]):

$$|X|_4 = \sum_{i=1}^{n} |\hat{x}_i|_4 - |q|_4,$$

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

To compute values $\mod 3$, we use a similar formula:

$$|X|_3 = \sum_{i=1}^{n} |\hat{x}_i|_3 - |q|_3.$$

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

| Input: $A, P > 2$ with gcd$(A, P) = 1$ |
| Precomp.: $T_8, T_8^{-1}, C_i$ |
| Result: $S = [A^{-1}]_P, 0 \leq S < 2P$ |
| $u \leftarrow 0, v \leftarrow 0, \hat{U}_1 \leftarrow 1, \hat{V}_3 \leftarrow \hat{A}$ |
| $b_{\hat{v}_1} \leftarrow 0, t_{\hat{U}_3} \leftarrow 0, b_{\hat{v}_3} \leftarrow |P|_3, t_{\hat{U}_3} \leftarrow |P|_3$ |
| $b_{\hat{v}_3} \leftarrow 1, t_{\hat{U}_3} \leftarrow 0, b_{\hat{v}_3} \leftarrow \mod 4(V_3), t_{\hat{U}_3} \leftarrow \mod 3(V_3)$ |
| while $\hat{V}_3 \neq 1$ and $\hat{U}_3 \neq 1$ and $\hat{V}_3 \neq -1$ and $\hat{U}_3 \neq -1$ do |
| while $|b_{\hat{v}_3}|_4 = 0$ or $|t_{\hat{U}_3}|_4 = 0$ do |
| $r = 1$ |
| $\hat{V}_3 \leftarrow \hat{V}_3 + \hat{U}_3 - C_1, b_{\hat{v}_3} = 0, |t_{\hat{U}_3}|_4 = 0$ |
| $\hat{V}_3 \leftarrow \hat{V}_3 + \hat{U}_3 - C_1, b_{\hat{v}_3} = 0, |t_{\hat{U}_3}|_4 = 0$ |
| $\hat{V}_3 \leftarrow \hat{V}_3 + \hat{U}_3 - C_1, b_{\hat{v}_3} = 0, |t_{\hat{U}_3}|_4 = 0$ |
| $\hat{V}_3 \leftarrow \hat{V}_3 + \hat{U}_3 - C_1, b_{\hat{v}_3} = 0, |t_{\hat{U}_3}|_4 = 0$ |

However, modulo 3 reduction on $\hat{x}_i$ 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 $\mod 3$ computes Eqn. (2).
To compute $\text{mod}3$ and $\text{mod}4$, the value $q = \frac{\sum_{i=1}^{n} x_i M}{M}$ 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 unit sums up the $t$ most significant bits of each residue, here $x_i$. 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 $[-P, P]$ [49]. Using our $\hat{X}$ representation, values are now in $[C_0 - P, C_0 + P]$ for, instance $\lfloor 11P, 13P \rfloor$ for $\overrightarrow{C_0} = 12P$. 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 $\text{mod}4$ and $\text{mod}3$ computations.

The computation dedicated to $\text{mod}4$ 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 $\text{mod}3$ 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. [V] and [9]), then we can neglect them compared to RNS operations (RNS additions and multiplications).

**D. Auxiliary Functions $\text{divv0}$ and $\text{divvup}$**

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 $\text{divvup}$.

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 $f_D(X) \equiv -XP^{-1} \mod D$, in other words $X + f_D(X)P$ is a multiple of $D$ and $X + f_D(X)P \equiv X \mod D$. 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 [4]. 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 $f_D(X) \in [-\lfloor \frac{D}{2} \rfloor + 1, \lfloor \frac{D}{2} \rfloor - 1]$ enables to divide by 2 the number of stored values, but it can introduce negative intermediate values which must be managed by choosing $C_0 > (\lfloor \frac{D}{2} \rfloor - 1)P$, i.e. $C_0 > 5P$ for $D = 12$ (for instance $C_0 = 12P$) in the $\hat{X}$ representation.

To formally define the division function, called $\text{divv0}$, we denote $\text{Cox}_1 = C_0T_B^{-1}$ and $\text{P}_1 = PT_B^{-1}$, and thus $\hat{X} = XT_B^{-1} + C_1$. Then

$$\text{divv0}(\hat{X}, D, |X|_D) = \frac{\hat{X}}{D} + f_D(X)\frac{\text{P}_1}{D} + \frac{(D - 1)\text{C}_1}{D}.$$ 

In Eqn. [3] we recall that $|X|_D$ is required to compute $f_D(X)$. More, the last term $\frac{(D - 1)\text{C}_1}{D}$ is added to ensure that the result is in the $\hat{X}$ representation because

$$\frac{\hat{X}}{D} + \frac{(D - 1)\text{C}_1}{D} = \frac{\hat{X}T_B^{-1}}{D} + \frac{\text{C}_1}{D} + \frac{(D - 1)\text{C}_1}{D} = (\frac{\hat{X}}{D}).$$

To speed up the evaluation, we can precompute the possible combinations of the constants $f_D(X)\frac{\text{P}_1}{D} + \frac{(D - 1)\text{C}_1}{D}$. Then, $\text{divv0}$ 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 $\text{divvup}$, which uses mod3, mod4 and $\text{divv0}$ 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. $t_{V_4}$). After the division, we compute mod3 for control values $t_{V_3}$ and $t_{V_4}$, and mod4 for control values $t_{V_2}$ and $t_{V_4}$. Finally, in order to update $u$ and $v$, we need to evaluate the number of "suppressed" bits during the divisions by 3, we use $\sigma$ an approximation of $\log_3(3) \approx 1.5849$. In practice, using $\sigma = 1.5$ is accurate enough for our target finite fields.

To optimize the function $\text{divvup}$, 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 $3^{-1} \text{mod} 4 = 3$ and adding a small specific precomputed value. Formally, defining $b_x \in \{-1, 0, 1\}$ and $t_x \in \{-1, 0, 1\}$ such that $b_x \equiv b_X \text{ mod} 4$ and $t_x \equiv t_X \text{ mod} 3$ respectively, we define

update4($b_x, t_x$) $= \lfloor 3(b_x + (2|P|_4 - 3) t_x P) \rfloor_4$,

update3_2($t_x, b_x$) $= \lfloor 2(t_x + b_x P) \rfloor_3$ and

update3_4($t_x, b_x, b_y$) $= \lfloor (t_x + (|P|_4 - 2) b_y P) \rfloor_3$.

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

| $|X|_6$ | 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: \( \hat{V}_3, b_{V_3}, t_{V_3}, \hat{V}_1, b_{V_1}, t_{V_1}, v \)

Precomp.: \( \alpha C_1 \) and \( \beta (PT_B^{-1}) \)
with \( \alpha \in \{1, 2, 3, 5, 11, 12\} \) and \( \beta \in \{1, 5, 11, 17, 23, 29\} \) and

Output: \( \hat{V}_3, b_{V_3}, t_{V_3}, \hat{V}_1, b_{V_1}, t_{V_1}, v \)

1. case \( b_{V_3} = 0 \) and \( t_{V_3} = 0 \)
\[
\hat{V}_3 \leftarrow \text{div12} \left( \hat{V}_3, b_{V_3}, t_{V_3} \right),
\hat{V}_1 \leftarrow \text{div12} \left( \hat{V}_1, b_{V_1}, t_{V_1} \right)
\]
\[
b_{V_3} \leftarrow \text{mod4}(\hat{V}_3), t_{V_3} \leftarrow \text{mod3}(\hat{V}_3), \quad t_{V_1} \leftarrow \text{mod3}(\hat{V}_1),
\]
\[
v \leftarrow v + \sigma + 2
\]
2. case \( b_{V_3} = 2 \) and \( t_{V_3} = 0 \)
\[
\hat{V}_3 \leftarrow \text{div6} \left( \hat{V}_3, b_{V_3}, t_{V_3} \right), \quad \hat{V}_1 \leftarrow \text{div6} \left( \hat{V}_1, b_{V_1}, t_{V_1} \right)
\]
\[
b_{V_3} \leftarrow \text{mod4}(\hat{V}_3), \quad b_{V_1} \leftarrow \text{mod4}(\hat{V}_1), \quad t_{V_3} \leftarrow \text{mod3}(\hat{V}_3), \quad t_{V_1} \leftarrow \text{mod3}(\hat{V}_1),
\]
\[
v \leftarrow v + \sigma + 2
\]
3. case \( b_{V_3} = 0 \) and \( t_{V_3} \neq 0 \)
\[
\hat{V}_3 \leftarrow \text{div4} \left( \hat{V}_3, b_{V_3}, t_{V_3} \right), \quad \hat{V}_1 \leftarrow \text{div4} \left( \hat{V}_1, b_{V_1}, t_{V_1} \right),
\]
\[
t_{V_3} \leftarrow \text{update3}_4(t_{V_3}, b_{V_3}), \quad t_{V_1} \leftarrow \text{update3}_4(t_{V_1}, b_{V_1})\quad v \leftarrow v + 2,
\]
\[
b_{V_3} \leftarrow \text{mod4}(\hat{V}_3), \quad b_{V_1} \leftarrow \text{mod4}(\hat{V}_1)
\]
4. case \( b_{V_3} \neq 2 \) and \( t_{V_3} = 0 \)
\[
\hat{V}_3 \leftarrow \text{div3} \left( \hat{V}_3, t_{V_3}, \right), \quad \hat{V}_1 \leftarrow \text{div3} \left( \hat{V}_1, t_{V_1} \right)
\]
\[
b_{V_3} \leftarrow \text{update4}(b_{V_3}, t_{V_3}), \quad b_{V_1} \leftarrow \text{update4}(b_{V_1}, t_{V_1})\quad v \leftarrow v + \sigma, \quad t_{V_3} \leftarrow \text{mod3}(\hat{V}_3), \quad t_{V_1} \leftarrow \text{mod3}(\hat{V}_1)
\]
5. case \( b_{V_3} = 2 \) and \( t_{V_3} \neq 0 \)
\[
\hat{V}_3 \leftarrow \text{div2} \left( \hat{V}_3, b_{V_3}, t_{V_3} \right), \quad \hat{V}_1 \leftarrow \text{div2} \left( \hat{V}_1, b_{V_1}, t_{V_1} \right)
\]
\[
t_{V_3} \leftarrow \text{update3}_2(t_{V_3}, b_{V_3}), \quad t_{V_1} \leftarrow \text{update3}_2(t_{V_1}, b_{V_1})\quad v \leftarrow v + 1,
\]
\[
b_{V_3} \leftarrow \text{mod4}(\hat{V}_3), \quad b_{V_1} \leftarrow \text{mod4}(\hat{V}_1)
\]

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 divD leading to 2n EMVs 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 divD and 2 RNS additions. Thus each outer loop iteration costs on average 3.33n EMVs and 5.33n EMAs. For a 6-bit field, there are 0.71\( \ell \) outer loop iterations, leading to 2.37\( \ell \) EMVs and 3.79\( \ell \) EMAs on average.

For the BMI algorithm proposed in this paper, one performs one divup per inner loop iteration, leading to 2 divD and 2n EMVs 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 divD and 2 RNS additions are performed leading to an average cost of 3.5n EMVs 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 0.46\( \ell \) outer loop iterations on average. Then the total average cost of the outer loop is 1.61\( \ell \) EMVs and 2.53\( \ell \) EMAs.

Fig. 2 graphically compares the theoretical complexities for BMI and BTMI for \( \ell \) fixed (and \( n \) variable) or \( n \) variable (and \( \ell \) fixed). On average, BTMI reduces by 30% the number of EMVs 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.
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. 3 computes the modulo 3 reduction $|x_i|_3$ 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 (top left in Fig. 4) for values $\frac{1}{3}$, $\frac{2}{3}$, and $\frac{1}{17}$, 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 $|x_i|_3$ 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 $|s|_3 = \sum_{i=1}^{n} |x_i|_3$.

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 $\ell$, 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 our previous BMI architecture and our new BTMI on Virtex 5 FPGA.

<table>
<thead>
<tr>
<th>Unit</th>
<th>Sub-unit</th>
<th>BMI</th>
<th>BTMI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rower</td>
<td>mod3</td>
<td>0</td>
<td>6–11 slices</td>
</tr>
<tr>
<td>Mem. (w-bit words)</td>
<td>17</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>BRAM</td>
<td>1 or 2</td>
<td>1 or 2</td>
<td></td>
</tr>
<tr>
<td>Cox</td>
<td>mod3</td>
<td>0</td>
<td>6–11 slices</td>
</tr>
<tr>
<td>CTRL</td>
<td>FSM</td>
<td>25 states</td>
<td>30 states</td>
</tr>
<tr>
<td></td>
<td>2-bit values</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

TABLE IV
Typical global area overhead of BTMI compared to BMI implementations on Virtex 5 FPGA.

<table>
<thead>
<tr>
<th>field size $\ell$ (bits)</th>
<th>192</th>
<th>256</th>
<th>384</th>
<th>521</th>
</tr>
</thead>
<tbody>
<tr>
<td>min overhead (%)</td>
<td>2.9</td>
<td>3.1</td>
<td>2.9</td>
<td>2.1</td>
</tr>
<tr>
<td>max overhead (%)</td>
<td>3.6</td>
<td>3.4</td>
<td>3.1</td>
<td>2.5</td>
</tr>
</tbody>
</table>

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.

VII. Conclusion

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


