Cryptographic Primitives Optimization Based on the Concepts of the Residue Number System and Finite Ring Neural Network

Andrei Tchernykh[1,2,3(✉)], Mikhail Babenko[2,4], Bernardo Pulido-Gaytan[1], Egor Shiryaev[4], Elena Golimblevskaia[4], Arutyun Avetisyan[2], Nguyen Viet Hung[5], and Jorge M. Cortés-Mendoza[3]

[1] CICESE Research Center, Ensenada, BC, Mexico
chernykh@cicese.mx, lpulido@cicese.edu.mx
[2] Ivannikov Institute for System Programming, Moscow, Russian Federation
mgbabenko@ncfu.ru, arut@ispras.ru
[3] South Ural State University, Chelyabinsk, Russia
kortesmendosak@susu.ru
[4] North-Caucasus Federal University, Stavropol, Russian Federation
eshiriaev@ncfu.ru
[5] Le Quy Don Technical University, Hanoi, Vietnam
hungnv@lqdtu.edu.vn

Abstract. Data encryption has become a vital mechanism for data protection. One of the main challenges and an important target for optimization is the encryption/decryption speed. In this paper, we propose techniques for speeding up the software performance of several important cryptographic primitives based on the Residue Number System (RNS) and Finite Ring Neural Network (FRNN). RNS&FRNN reduces the computational complexity of operations with arbitrary-length integers such as addition, subtraction, multiplication, division by constant, Euclid division, and sign detection. To validate practical significance, we compare LLVM library implementations with state-of-the-art, high-performance, portable C++ NTL library implementations. The experimental analysis shows the superiority of the proposed optimization approach compared to the available approaches. For the NIST FIPS 186-5 digital signature algorithm, the proposed solution is 85% faster, even though the sign detection has low efficiency.

Keywords: Residue number system · Finite ring neural network · Encryption · High-performance · Cryptographic primitives

1 Introduction

Security becomes commonplace in all modern computing areas and affects many fields, including casual people communication, Internet of Things (IoT), analytics, self-learning systems, cloud computing, etc. Advanced cryptographic algorithms provide key mechanisms for data confidentiality, integrity, authentication, non-repudiation, etc.

B. Dorronsoro et al. (Eds.): OLA 2021, CCIS 1443, pp. 241–253, 2021.
https://doi.org/10.1007/978-3-030-85672-4_18

The cryptographic primitives are usually complex in terms of computational overhead and memory usage. They are designed based on mathematical theory, elliptic curves, Neural Networks (NNs), etc.

The high performance of cryptographic algorithms is important for numerous reasons. The principal one is the computational cost in terms of execution time. They can be executed by conventional computers, accelerated computing servers, and specialized hardware devices. In many cases, they are implemented as software components.

Many approaches are used to optimize encryption operations. Neuromorphic computing is concerned with emulating the neural structure and operation of the human brain. The main goals are to create a device that can extract better features, learn, recognize, classify, acquire new information, and even make a logical inference.

For instance, a single-chip prototype of the BrainScaleS 2, Intel Labs designed Loihi, and IBM's TrueNorth neuromorphic systems provide a proof-of-concept of a spiking neural network application to learn neurons and synapses [13, 14]. They include a hundred thousand neurons, each of which can communicate with thousands of others.

A Residue Number System (RNS) can achieve both fast computation and low power consumption. It is parallel, adaptable, and fault-tolerant, meaning it can produce results after components are failed [9, 10]. These properties allow for the successful development of cybersecurity systems [11–18].

RNS is a number system that represents integers by the remainders of division by several pairwise coprimes, called moduli. The arithmetic is called multi-modular arithmetic. It is widely used for computation with arbitrary length integers, for instance, in cryptography. It provides faster computation than with the usual numeral systems, even when converting between numeral systems is taken into account. By decomposing a large integer into a set of smaller integers, a large calculation is performed as a series of smaller calculations that can be performed independently and in parallel. The number of parallel elementary processes equals the number of RNS moduli.

In this paper, we propose a new optimization method RNS&FRNN of operations with arbitrary-length integers based on RNS and Finite Ring Neural Network (FRNN).

This paper is organized as follows. Section 2 describes the main concept of modular arithmetical operations. Section 3 introduces modular logical operations. Section 4 presents the scaling of RNS numbers by RNS base extension and introduces RNS&FRNN optimization method. Section 5 focuses on the experimental analysis. The conclusions and future work are discussed in the last Sect. 6.

2 Modular Arithmetical Operations

2.1 Addition, Subtraction, Multiplication, and Division

In the RNS, arithmetic operations are performed on each residue, according to the following general formula:

$$X \circ Y \overset{RNS}{\rightarrow} \left(|x_1 \circ y_1|_{p_1}, |x_2 \circ y_2|_{p_2}, \ldots, |x_n \circ y_n|_{pn} \right), \tag{1}$$

where $\{p_1, p_2, \ldots, p_n\}$ is a moduli set of pairwise coprime numbers. "$\circ$" denotes the operation of addition, subtraction, or multiplication.

Integer numbers X and Y are defined in RNS as tuples $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$, where x_i represents the remainder of the division of X by pi, defined by $x_i = |X|_{p_i}$.

However, an additional restriction is imposed on the multiplication operation, which follows from the Chinese Remainder Theorem (CRT): $X \cdot Y < P$, where $P = \prod_{i=1}^{n} p_i$.

Integer division can be performed by various methods [2–4]. The most reliable algorithm is based on the scaling method. In this case, a dividend is an arbitrary number in the range $[0, P)$, and a divisor is any factor of $P = p_1 \cdot p_2 \cdot \ldots \cdot p_n$.

This division is similar to dividing by numbers belonging to a certain limited set, which is faster than dividing by an arbitrary divisor (2).

$$X = \left\lfloor \frac{X}{p_1} \right\rfloor \cdot p_1 + x_1, \tag{2}$$

where X is the dividend, and p_1 is the divisor.

The dividend is represented by the residues $X \overset{RNS}{\rightarrow} (x_1, x_2, \ldots, x_n)$, and the divisor is one of the moduli p_i. x_i is the residue of the division. In the first step of scaling, it is necessary to subtract the residue from the dividend (3):

$$X' \overset{RNS}{\rightarrow} \left(x_1', x_2', \ldots, x_n'\right) = \left(\left|x_1 - |x_i|_{p_1}\right|_{p_1}, \left|x_1 - |x_i|_{p_2}\right|_{p_2}, \ldots, \left|x_n - |x_i|_{p_n}\right|_{p_n}\right). \tag{3}$$

In the second step, the division of X' by p_i is carried out directly by (4):

$$\left\lfloor \frac{X}{p_1} \right\rfloor \overset{RNS}{\rightarrow} \left(-, \left|x_1' \left|p_1^{-1}\right|_{p_2}\right|_{p_2}, \ldots, \left|x_n' \left|p_1^{-1}\right|_{p_n}\right|_{p_n}\right), \tag{4}$$

where $\left|p_1^{-1}\right|_{p_i}$ is the multiplicative inversion of p_i.

At the end of the second stage, the residue x_i modulo p_i remains unknown, which can be found using the base extension (Sect. 4).

2.2 Euclidean Division

Euclidean division is carried out using the approximate division method. The essence of the approximate method for calculating the positional characteristic to compare and restore the positional notation of the numbers in RNS. It is based on the relative values of the numbers to the full range determined by CRT.

We have:

$$X = \left| \sum_{i=1}^{n} \frac{P}{p_i} \left| P_i^{-1} \right|_{p_i} x_i \right|_P, \tag{5}$$

where $P = \prod_{i=1}^{n} p_i$, p_i is the RNS moduli, $\left|P_i^{-1}\right|_{p_i}$ is the multiplicative inversion of P_i relative to p_i, $P_i = \frac{P}{p_i}$.

To overcome this difficulty, it is necessary to compare the current iteration values with the previous ones in the RNS. It allows to correct determining a larger or smaller number. The overflow of the dynamic range in the RNS can be used to make the decision "more – less".

In the first iteration, the dividend is compared with the divisor, and at the other iterations, the doubled values of the divisors $q_i Y < q_{i+1} Y$ are compared. In each new iteration, the current value is compared with the previous one.

The number of iterations required depends on the divisible and divisor values. Successive application of this operation leads to the formation of a sequence of integers $Yq_1 < ... < Yq_n > Yq_{n+1}$.

Let the case $Yq_n > Yq_{n+1}$ be fixed at $n + 1$ iterations, which corresponds to an overflow of the RNS range, i.e., $Yq_{n+1} > P$ and $X < Yq_{n+1}$. This completes the process of generating interpolation of the quotient by a binary series or by a set of constants in the RNS.

The process of approximating the quotient can be carried out by comparing only doubled neighboring approximate divisors. An important issue when implementing the function $F()$ is the accuracy of the coefficients.

It should also be noted that the number of characters in the fractional part should be twice as much as the number of characters in the RNS range. The modular numbers' division based on the approximate method of comparing numbers consists of the following steps (see Algorithm 1).

In this case, when the divisor has the minimum value and the dividend has the maximum, the threshold Δ_i is more than zero. It reduces the number of iterations when dividing a large divisible and a small divisor.

Algorithm 1. Euclidean division in RNS.

Input: $X \overset{RNS}{\longrightarrow} (x_1, x_2, \ldots, x_n)$, $Y \overset{RNS}{\longrightarrow} (y_1, y_2, \ldots, y_n)$, $F(X)$.

Output: $\omega = \left\lfloor \frac{X}{Y} \right\rfloor$, $\gamma = |X|_Y$.

Step 1. We calculate the approximate values of the divisible $F(X)$ and the divisor $F(Y)$ and compare them. If $F(X) < F(Y)$, then the division process ends and the quotient $\left\lfloor \frac{X}{Y} \right\rfloor = 0$. If $F(X) = F(Y)$, then the division process ends, and the quotient is equal to unity. If $F(X) > F(Y)$, then a higher degree 2^k is searched for by approximating the quotient with a binary code.

Step 2. We select the constant 2^k (the highest power of the series), multiply it by the divisor $F_1(X) = X2^k$ and introduce it into the comparison scheme. The constants $2^j \bmod p_i$, where $i = \overline{1,n}$, $1 \le j \le \log_2 P$ are previously stored in the memory.

Step 3. We find $\Delta_i = F(X) - F_1(Y)$. If in the sign digit Δ_i is "1", then the corresponding degree of the series is discarded, if it is "0", then in the adder of the quotient we add the value of a member of the series with this degree, that is 2^k.

Step 4. We find $F_1(Y)$, and check the term of the series with a degree 2^{k-1}.

Step 5. We find $\Delta_2 = \Delta_1 - F_1(Y)$ and perform the actions in accordance with paragraph 4.

Step 6. Similarly, we check all the remaining members of the series of the pre-zero degree. The resulting residue $\Delta_i = \Delta_{i-1} - F_{i-1}(Y) \approx 0$.

$x_1 \div x_n$ with coefficients $g_1 \div g_n$ calculates the rank of the number r_A. Then the modular network modulo p_j calculates the value $\sum_{i=1}^{n-1} a_i |B_i|_{p_{n+1}}$. In the second stage, $x_j = |X|_{p_j}$ is calculated using the computational model (11).

Each set of moduli of the modular code is characterized by an orthogonal basis, due to which, for the base extension, it is necessary to recalculate the basis $B_i^{'}$, $i = \overline{1, n+1}$. To recalculate them, the input data are: orthogonal basis B_i, $i = \overline{1, n}$, the moduli $p_1, p_2, \ldots, p_n$ and the values of the extended modulo p_j. Since $P_i^{'} = P'/p_i$ and P_i are coprime, we can calculate the orthogonal basis of the extended system as follows

$$B_i^{'} \equiv \frac{P'}{p_i} \cdot \left| P_i^{'-1} \right|_{p_i} \tag{13}$$

To calculate it on a NN basis, it is necessary to calculate two constants: $\left|\frac{1}{P_j}\right|_{pi}$. and $P_i^{'} = \frac{P'}{p_i}$. Thus, the NN architecture can be presented as following (Fig. 2).

The proposed algorithm has lower computational complexity compared to the known methods. However, the method involves multiplying pre-calculated constants. These constants are usually known in advance.

5 Experimental Results

We perform experimental analysis on CPU 2.7 GHz Intel Core i5, RAM 8 GB 1867 MHz DDR3, macOS High Sierra version 10.13.6 operating system. We use NTL, a high-performance, portable C++ library version 11.4.3, and LLVM's OpenMP runtime library version 10.0.0. RNS moduli are generated as a sequence of decreasing consecutive coprime numbers starting from $p_1 = 32,749, \ldots, p_{285} = 29,789$, and $L = \lceil \log_2 P \rceil$. One million random values of X and Y are generated using RandomBnd() function, an NTL routine for generating pseudo-random numbers. Execution time T of arithmetic and logical operations are measured in microseconds (μs). The number of threads is four. The results are presented in Table 1.

First, we measure the relative performance of each operation independently. The speedup of RNS&FRNN is between 9,954 and 25,888 for the addition, 12,348 and 31,385 for the subtraction, 13,193.9 and 318,203 for multiplication, 15,353.5 and 140,290 for division by constant, and 17,815.5 and 40,359.7 for Euclid division, varying n and L. RNS sign detection performance is between 4.5 and 15 times lower.

Now, let us compare the performance of NIST FIPS 186–5 digital signature algorithm with two implementations. It is based on the operation of multiplying the point of an elliptic curve over $GF(q)$ by a scalar, the most time-consuming operation, where q is a prime number.

Different approaches for computing the elliptic scalar multiplication are introduced. Well-known Montgomery approach is based on the binary method, where scalar multiplication is defined to be the elliptic point resulting from adding value to itself several times. It performs addition and doubling in each iteration.

Let us evaluate the mathematical expectation of the number of additions and doubling.

Table 1. Execution time of operations on NTL 11.4.3 (binary) and RNS&FRNN (RNS) (μs).

(a) Addition, Subtraction, and Multiplication

n	L	Addition			Subtraction			Multiplication		
		Binary	RNS	Binary/RNS	Binary	RNS	Binary/RNS	Binary	RNS	Binary/RNS
15	225	99,548	10	**9,954.8**	105,076	8	13,134.5	118,745	9	**13,193.9**
30	450	110,852	10	11,085.2	126,619	8	15,827.4	194,619	9	21,624.3
45	675	103,665	8	12,958.1	111,137	8	13,892.1	198,589	10	19,858.9
60	900	108,377	10	10,837.7	116,266	8	14,533.3	322,731	8	40,341.4
75	1,124	113,044	8	14,130.5	115,830	9	12,870	392,779	10	39,277.9
90	1,349	114,060	8	14,257.5	120,409	8	15,051.1	510,666	8	63,833.3
105	1,573	116,498	9	12,944.2	123,482	10	**12,348.2**	604,474	9	67,163.8
120	1,797	168,430	9	18,714.4	180,615	10	18,061.5	727,589	9	80,843.2
135	2,021	167,513	8	20,939.1	179,552	8	22,444	827,077	8	103,384.6
150	2,245	172,927	8	21,615.9	185,494	9	206,10.4	973,639	10	97,363.9
165	2,469	172,716	9	19,190.7	218,787	8	27,348.4	1,140,607	9	126,734.1
180	2,693	180,369	9	20,041.0	231,800	8	28,975	1,328,500	8	166,062.5
195	2,917	186,132	9	20,681.3	199,568	10	19,956.8	1,397,494	9	155,277.1
210	3,140	186,433	9	20,714.8	211,051	8	26,381.4	1,602,832	8	200,354.0
225	3,364	187,804	9	20,867.1	209,095	9	23,232.8	1,757,143	9	195,238.1
240	3,587	201,887	8	25,235.9	221,684	9	24,631.6	1,936,657	8	242,082.1
255	3,810	201,556	8	25,194.5	243,480	10	24,348	2,117,587	8	264,698.4
270	4,033	233,000	9	**25,888.9**	241,572	8	30,196.5	2,208,706	9	245,411.8
285	4,256	215,689	10	21,568.9	282,472	9	**31,385.8**	2,545,628	8	**318,203.5**

(b) Division by constant, Euclid division, and Sign detection

n	L	Division by constant			Euclid division			Sign detection		
		Binary	RNS	Binary/RNS	Binary	RNS	Binary/RNS	Binary	RNS	Binary/RNS
15	225	122,828	8	**15,353.5**	171,928	8	21,491.0	1	9	0.11
30	450	168,685	9	18,742.8	182,879	9	20,319.9	1	8	0.13
45	675	145,610	9	16,178.9	178,155	10	**17,815.5**	1	9	0.11
60	900	174,282	8	21,785.3	201,592	10	20,159.2	1	9	0.11
75	1,124	198,819	8	24,852.4	183,151	9	20,350.1	1	9	0.11
90	1,349	220,280	9	24,475.6	191,398	8	23,924.8	1	9	0.11
105	1,573	244,787	9	27,198.6	194,943	8	24,367.9	1	9	0.11
120	1,797	319,813	8	39,976.6	251,513	8	31,439.1	1	8	0.13
135	2,021	334,435	9	37,159.4	252,916	9	28,101.8	1	15	**0.07**
150	2,245	362,685	8	45,335.6	266,925	9	29,658.3	1	10	0.10
165	2,469	407,955	9	45,328.3	262,714	8	32,839.3	1	9	0.11
180	2,693	439,295	10	43,929.5	282,383	8	35,297.9	1	9	0.11
195	2,917	451,525	9	50,169.4	287,426	8	35,928.3	1	9	0.11
210	3,140	461,168	9	51,240.9	283,955	9	31,550.6	2	10	0.20
225	3,364	486,675	10	48,667.5	285,086	8	35,635.8	1	8	0.13
240	3,587	504,493	9	56,054.8	332,445	10	33,244.5	1	9	0.11
255	3,810	537,938	10	53,793.8	331,538	9	36,837.6	1	10	0.10
270	4,033	1,262,615	9	**140,290.6**	363,237	9	**40,359.7**	1	10	0.10
285	4,256	553,609	9	61,512.1	355,031	10	35,503.1	2	9	**0.22**

Doubling can be expressed as:

$$\frac{1}{2^{\lceil \log_2 q \rceil}} \sum_{i}^{\log_2 q - 1} i \cdot 2^i = \frac{\left(\lceil \log_2 q \rceil - 2\right) 2^{\lceil \log_2 q \rceil} + 2}{2^{\lceil \log_2 q \rceil}} \approx \lceil \log_2 q \rceil - 2 \qquad (14)$$

Addition can be expressed as:

$$\frac{1}{2^{\lceil \log_2 q \rceil}} \sum_{i=0}^{\lceil \log_2 q \rceil} i \cdot C^i_{\lceil \log_2 q \rceil} = \frac{\lceil \log_2 q \rceil}{2^{\lceil \log_2 q \rceil}} \cdot 2^{\lceil \log_2 q \rceil - 1} = \frac{\lceil \log_2 q \rceil}{2}, \qquad (15)$$

where $C^a_b = \frac{b!}{(b-a)! \cdot a!}$.

Using the projective Jacobian coordinates for the case when $Z \neq 1$ and $a = -3$, it takes 16 multiplications to add points, and 8 multiplications to double a point.

Statistical analysis of the algorithm demonstrates that the mathematical expectation of number of modular multiplications is about

$$\frac{\lceil \log_2 q \rceil}{2} \cdot 16 + \left(\lceil \log_2 q \rceil - 2\right) \cdot 8 = 16 \log_2 q - 16 \qquad (16)$$

The execution time of the modular multiplication can be estimated as a sum of one multiplication and one addition; hence, $T_{Bin} = (16\lceil \log_2 q \rceil - 16)(M_{Bin} + A_{Bin})$, where M_{Bin} is the execution time of the multiplications and A_{Bin} is the execution time of the addition.

To assess the RNS implementation of the algorithm, first, we consider the RNS to binary T_C and binary to RNS T_E conversion times (Table 2).

The modular multiplication of an elliptic curve point by a scalar in RNS requires one multiplication, one addition, $n(n-1)/4$ divisions by a constant, and one operation for determining the sign of a number, where n is the number of moduli.

The execution time of the RNS implementation can be estimated as

$$T_{RNS} = \left(16\lceil \log_2 q \rceil q - 16\right)\left(M_{RNS} + A_{RNS} + \frac{n(n-1)}{4} DC_{RNS} + S_{RNS}\right) + 2(T_C + T_E),$$

where M_{RNS} is the execution time of multiplication of two numbers in RNS, A_{RNS} is the execution time of addition in RNS, DC_{RNS} is the execution time of the division by constant in RNS, S_{RNS} is the execution time of the sign detection, T_C is the time of binary to RNS conversion, and T_E is the time of RNS to binary conversion.

Thus, for $q = 2^{511} - 1$, $n = 75$, T_{Bin} and T_{RNS} are estimated, in the worst case, as $T_{Bin} = (16 \cdot 511 - 16) \cdot (392,779 + 113,044) = 4,127,515,680$, $T_{RNS} = (16 \cdot 511 - 16) \cdot (10 + 8 + 75 \cdot (75-1)/4 \cdot 8 + 9) + 2 \cdot (384,687,100 + 685,935,110) = 2,232,040,738$. Therefore, $T_{Bin}/T_{RNS} \approx 1.85$ times.

14. DeBole, M.V., et al.: TrueNorth: accelerating from zero to 64 million neurons in 10 years. Computer **52**(5), 20–29 (2019). https://doi.org/10.1109/MC.2019.2903009.
15. Babenko, M., et al.: RNS number comparator based on a modified diagonal function. Electronics **9**, 1784 (2020). https://doi.org/10.3390/electronics9111784
16. Miranda-Lopez, V., et al.: Weighted two-levels secret sharing scheme for multi-clouds data storage with increased reliability. In: 2019 International Conference on High Performance Computing & Simulation (HPCS), pp. 915–922. IEEE (2019). https://doi.org/10.1109/HPCS48598.2019.9188057
17. Babenko, M., Deryabin, M., Tchernykh, A.: The accuracy estimation of the interval-positional characteristic in residue number system. In: 2019 International Conference on Engineering and Telecommunication (EnT), pp. 1–5. IEEE (2019). https://doi.org/10.1109/EnT47717.2019.9030549
18. Kucherov, N., Babenko, M., Tchernykh, A., Kuchukov, V., Vashchenko, I.: Increasing reliability and fault tolerance of a secure distributed cloud storage. In: The International Workshop on Information, Computation, and Control Systems for Distributed Environments (2020) https://doi.org/10.47350/ICCS-DE.2020.16.