Run-based Modular Reduction Method

Zhengjun Cao[1], Zhen Chen[1], Ruizhong Wei[2], and Lihua Liu[3]
(Corresponding author: Zhengjun Cao)
Department of Mathematics, Shanghai University, No. 99, Shangda Road, 200444, Shanghai, China[1]
Department of Computer Sciences, Lakehead University, 955, Oliver Road, Thunder Bay, Canada[2]
Department of Mathematics, Shanghai Maritime University, No.1550, Haigang Ave, Shanghai, China[3]
(Email: caozhj@shu.edu.cn)
(Received Oct. 19, 2018; Revised and Accepted Feb. 7, 2019; First Online June 11, 2019)

Abstract

The existing lookup-table modular reduction methods partition the binary string of an integer into fixed-length blocks such as 32 bits or 64 bits. This approach requires a fixed amount of looking up tables. In this paper, we introduce a new modular reduction method which partitions the binary string of an integer into blocks according to its runs. The new method can efficiently reduce the amount of looking up tables. Its complexity depends essentially on the amount of runs or 1's in the left segment of the binary string of an integer to be reduced. We show that the new reduction is almost twice as fast as the popular Barrett's reduction.

Keywords: Barrett's Reduction; Montgomery's Reduction; Run-based Modular Reduction

1 Introduction

The performance of public key cryptographic schemes depends heavily on the speed of modular reduction. Among several modular reduction algorithms, Montogomery's reduction and Barrett's reduction are more competitive. In 1985, P. Montgomery [13] invented an elegant reduction method. His method is not efficient for a single modular multiplication, but can be used effectively in computations where many multiplications are performed for given inputs.

At Crypto'86, P. Barrett [1] proposed a novel reduction method which is applicable when many reductions are performed with a single modulus. Barrett's reduction and Montgomery's reduction are similar in that expensive divisions in classical reduction are replaced by less-expensive multiplications. At Crypto'93, A. Bosselaers *et al.* [2] compared the performances of classical algorithm, Barrett's algorithm and Montgomery's algorithm. It is reported that these algorithms all have their specific behavior resulting in a specific field of application. No single algorithm is able to meet all demands. In 1998, Win *et al.* [18] reported that the difference between Montgomery's and Barrett's reduction was negligible in their implementation on an Intel Pentium Pro of field arithmetic in $\mathbb{F}_p$ for a 192-bit prime p. In 2011, Dupaquis and Venelli [4] modified Barrett's reduction and Montgomery's reduction. Their technique allows the use of redundant modular arithmetic. The proposed redundant Barrett's reduction algorithm can be used to strengthen the differential side-channel resistance of asymmetric cryptosystems.

In order to further speed up modular reduction, lookup table has been adopted by several researchers [7–9, 12, 14, 15, 17]. If the size of a pre-computed table is manageable, the method is very effective. These reduction methods partition the binary string of an integer into fixed-length blocks such as 32 bits or 64 bits. This approach requires a moderate size table. In 1997, Lim *et al.* [10] experimented on Montgomery's reduction, classical reduction, Barrett's reduction and some reduction algorithms using lookup table. It reported that the proposed lookup-table method runs almost two to three times faster on a workstation than the Montgomery's reduction. Although the experimental results are interesting, they did not present a complexity analysis of these combined lookup table methods.

The principles of the existing reduction algorithms can be briefly summarized as following:

- *Division.* The classical reduction algorithm adopts this principle.
- *Multiplication.* Both Barrett's and Montgomery's reduction adopt this principle.
- *Addition* [look up table according to *fixed-length blocks*]. All current reduction algorithms based on this principle look up table according to fixed-length blocks such as 32 bits or 64 bits.

The last principle, intuitively, is more applicable because it totally eliminates multiplications although it requires a moderate size table and a fixed amount of looking up tables. However, it seems that they are not suitable for small devices [5, 11, 16] such as smart phones.

In this paper, we put forth a new reduction method based on the principle of *addition* [look up table according to *runs*]. Unlike the traditional lookup-table reduction, the proposed method partitions the binary string

of an integer into blocks according to its runs instead of fixed-length blocks. The performance of the new method depends essentially on the amount of runs or 1's in the left segment of the binary string of an integer to be reduced. The new method can efficiently reduce the amount of looking up tables. We also provide a thorough complexity analysis of the method.

2 Related Reduction Methods

2.1 Montgomery's Reduction

Let $R > p$ with $\gcd(R, p) = 1$. The method produces $zR^{-1} \bmod p$ for an input $z < pR$. If $p' = -p^{-1} \bmod R$, then $c = zR^{-1} \bmod p$ can be obtained via

$$c \leftarrow (z + (zp' \bmod R)p)/R, \text{ if } c \geq p \text{ then } c \leftarrow c - p.$$

Given $x \in [0, p)$, let $\tilde{x} = xR \bmod p$. Define $\text{Mont}(\tilde{x}, \tilde{y}) = (\tilde{x}\tilde{y})R^{-1} \bmod p = (xy)R \bmod p$. The transformations $x \mapsto \tilde{x} = xR \bmod p$, and $\tilde{x} \mapsto \tilde{x}R^{-1} \bmod p = x$ are performed only once when they are used as a part of a larger calculation such as modular exponentiation.

2.2 Barrett's Reduction

The following description of Barrett's reduction comes from [6], which calculates $z \bmod p$. The algorithm first selects a suitable base b (e.g., $b = 2^L$ where L is near the word size of the processor). It then calculates $\mu = \lfloor \frac{b^{2k}}{p} \rfloor$, where $k = \lfloor \log_b p \rfloor + 1$. Suppose $0 \leq z < b^{2k}$. Let $q = \lfloor \frac{z}{p} \rfloor$, $r = z \bmod p = z - q\,p$. Since $\frac{z}{p} = \frac{z}{b^{k-1}} \cdot \frac{b^{2k}}{p} \cdot \frac{1}{b^{k+1}}$, we have

$$0 \leq \hat{q} = \left\lfloor \frac{\lfloor \frac{z}{b^{k-1}} \rfloor \cdot \mu}{b^{k+1}} \right\rfloor \leq \left\lfloor \frac{z}{p} \right\rfloor = q.$$

If μ is computed in advance, then the main cost of calculating $\hat{q}$ consists of one multiplication and two types of bit operations for $\lfloor \frac{z}{b^{k-1}} \rfloor$ and $\lfloor \frac{y}{b^{k+1}} \rfloor$, where $y = \lfloor \frac{z}{b^{k-1}} \rfloor \cdot \mu$. Set $\alpha = \frac{z}{b^{k-1}} - \lfloor \frac{z}{b^{k-1}} \rfloor$, $\beta = \frac{b^{2k}}{p} - \left\lfloor \frac{b^{2k}}{p} \right\rfloor$. Then $0 \leq \alpha, \beta < 1$ and

$$q = \left\lfloor \frac{\left(\lfloor \frac{z}{b^{k-1}} \rfloor + \alpha\right)\left(\left\lfloor \frac{b^{2k}}{p} \right\rfloor + \beta\right)}{b^{k+1}} \right\rfloor \leq \left\lfloor \frac{\lfloor \frac{z}{b^{k-1}} \rfloor \cdot \mu}{b^{k+1}} + \frac{\lfloor \frac{z}{b^{k-1}} \rfloor + \left\lfloor \frac{b^{2k}}{p} \right\rfloor + 1}{b^{k+1}} \right\rfloor$$

Since $0 \leq z < b^{2k}$ and $b^{k-1} \leq p < b^k$, we have

$$\left\lfloor \frac{z}{b^{k-1}} \right\rfloor + \left\lfloor \frac{b^{2k}}{p} \right\rfloor + 1 \leq (b^{k+1} - 1) + b^{k+1} + 1 = 2b^{k+1}$$

$$q \leq \left\lfloor \frac{\lfloor \frac{z}{b^{k-1}} \rfloor \cdot \mu}{b^{k+1}} + 2 \right\rfloor = \hat{q} + 2.$$

Therefore, we obtain $\hat{q} \leq q \leq \hat{q} + 2$. Set $\hat{r} = z - \hat{q}\,p$. We get $r = \hat{r} + (\hat{q} - q)p$. That is, at most two subtractions are required to obtain r using $\hat{r}$.

In 2014, Cao and Wu [3] pointed out that the formula

$$\frac{z}{p} = \frac{z}{b^{k-1}} \cdot \frac{b^{2k}}{p} \cdot \frac{1}{b^{k+1}}$$

can be directly replaced with

$$\frac{z}{p} = \frac{z}{2^k} \cdot \frac{2^{2k}}{p} \cdot \frac{1}{2^k}$$

The adaption could further optimize the programming code and solve the data expansion problem in Barrett's reduction.

2.3 Lookup-Table Reduction

Suppose that z and n are two integers, $b^{k-1} \leq n < b^k$, $0 \leq z < b^{2k}$ where $b = 2^L$ is a suitable base. To compute $z \bmod n$, the usual lookup-table reduction computes

$$z = \sum_{j=0}^{k-1} z_j b^j + \sum_{i=0}^{k-1} z_{k+i} A[i] \bmod n, \tag{1}$$

where $0 \leq z_j < b, j = 0, \cdots, 2k-1$, $A[i] = b^{k+i} \bmod n\,(0 \leq i \leq k-1)$ are computed and stored in advance. In 1997, Lim *et al.* [10] suggested taking $b = 2^{32}$. In this method, it only requires a storage for 624 values of modulus size (e.g., about 78 Kbytes for $|n| = 1024$). They experimented on Montgomery's reduction, classical reduction, Barrett reduction and some lookup-table reduction algorithms. It reported that:

1) Modular reduction takes considerably more time than multiplication;
2) Montgomery's algorithm and the combined table lookup method give almost the same performance;
3) The proposed table lookup methods (L224, L624, L1696) run almost two to three times faster on a workstation than Montgomery's reduction. These methods, however, do not give much improvement on a PC.

3 Basic Lookup-Table Reduction

The idea behind the basic lookup-table modular reduction is naive, but useful in some cases. We now describe it as follows.

3.1 Pre-computed Table

Given a positive integer n, choose an integer k such that $2^{k-1} < n < 2^k$. The pre-computed table are constructed as following (see Table 1).

We can specify that $|r[\ell]| \leq \lfloor n/2 \rfloor$, $\ell = k, \cdots, 2k-1$. The size of the pre-computation table $\mathbb{T}$ can be further reduced because $r[i+1] = 2\,r[i]$ for some indexes i.

Table 1: Pre-computation table $\mathbb{T}$ for a modular n

ℓ	$2k-1$	$2k-2$	$\cdots$	k
$r[\ell]$	$2^{2k-1} \bmod n$	$2^{2k-2} \bmod n$	$\cdots$	$2^k \bmod n$

3.2 Basic Method (Method-1)

Denote the binary string of a positive integer z by Binary(z). Suppose that $0 \le z < 2^{2k}$. We directly set the base $b = 2$ in Equation (1). It follows that

$$z \equiv \sum_{i=0}^{k-1} z_{k+i}\, r[k+i] + \sum_{j=0}^{k-1} z_j\, 2^j \bmod n, \tag{2}$$

where $z_j \in \{0,1\}, j = 0, \cdots, 2k-1$, $r[k+i] = 2^{k+i} \bmod n\,(0 \le i \le k-1)$. Since $z_{k+i} \in \{0,1\}$, $0 \le i \le k-1$, we completely eliminated multiplications.

Example 1. $n = 97 = (1100001)_2, k = 7$ *(bit-length)*, $z = 3135 = (110000111111)_2, l = 12$. *Look up for the values* $r[11] = 2^{11} \bmod n = 11$ *and* $r[10] = 2^{10} \bmod n = 54$. *It gives* $z = 3135 \equiv r[11] + r[10] + (111111)_2 = 11 + 54 + 63 \equiv 31 \bmod 97$.

3.3 Cost Analysis

The number of additions in this method depends on the amount of 1's in the left segment of Binary(z). On average, there are about $\lfloor k/2 \rfloor$ 1's in the left segment if the bit-length of z is $2k$. That means it requires $\lfloor k/2 \rfloor$ additions of k-bit integers to compute $r = \sum_{j=0}^{k-1} z_j 2^j + \sum_{i=0}^{k-1} z_{k+i} r[k+i]$. It is expected that the absolute value $|r| < \frac{kn}{4}$, since $|r[\ell]| \le \lfloor n/2 \rfloor$. Hence, it requires $\lfloor k/4 \rfloor$ subtractions to compute $r \bmod n$. In total, Method-1 requires the cost of performing $\lfloor \frac{3k}{4} \rfloor$ additions of k-bit integers.

In the method, addition happened for all values $r[k+i]$ corresponding to $z_{k+i} = 1\,(0 \le i \le k-1)$. In the worst case, $z_k = z_{k+1} = \cdots = z_{2k-1} = 1$, it has to look up table and do addition k times. Clearly, Method-1 is inappropriate for this case.

4 Run-based Reduction

The Method-1 is not good for the worst case when there is only one run of 1's in the left segment of Binary(z), *i.e.*, all the positions are 1's. We now introduce a new reduction method based on lookup table which is much better for the above case.

4.1 The Basic Idea

Given two positive integers k, n, where $2^{k-1} < n < 2^k$, and a positive integer z satisfying $0 \le z < 2^{2k}$, set $\ell_0 = \lfloor \log_2 z \rfloor + 1$. Flipping all bits of z, we obtain the integer z_1 such that $z = (2^{\ell_0} - 1) - z_1$. Set $\ell_1 = \lfloor \log_2 z_1 \rfloor + 1$. Flipping all bits of z_1, we obtain the integer z_2 such that $z = (2^{\ell_0} - 1) - (2^{\ell_1} - 1) + z_2$. By the same procedure, we shall get

$$\begin{aligned} z =& (2^{\ell_0} - 1) - (2^{\ell_1} - 1) + (2^{\ell_2} - 1) + \cdots \\ &+ (-1)^{j-1}(2^{\ell_{j-1}} - 1) + (-1)^j z', \end{aligned} \tag{3}$$

where $\ell_{j-1} > k \ge \ell_j$, ℓ_j is the bit-length of z'. Clearly,

$$\ell_0 > \ell_1 > \cdots > \ell_j. \tag{4}$$

We then look up the pre-computed table for values $r[\ell_0], \cdots, r[\ell_{j-1}]$ using the indexes $\ell_0, \cdots, \ell_{j-1}$ and compute

$$\begin{aligned} r =& (r[\ell_0] - 1) - (r[\ell_1] - 1) + (r[\ell_2] - 1) + \cdots \\ &+ (-1)^{j-1}(r[\ell_{j-1}] - 1) + (-1)^j z'. \end{aligned} \tag{5}$$

Thus, $z \equiv r \bmod n$.

4.2 Description of Method-2

To obtain indexes $\ell_0, \cdots, \ell_{j-1}$ and z' in Equation (5), the above procedure requires to flip all bits of strings. In fact, these indexes and z' depend essentially on the runs in the left segment of Binary(z). Here a run means a maximal substring whose bit positions all contain the same digit 0 or 1. We can obtain them by counting the length of each run in the left segment. Suppose that

$$\text{Binary}(z) = \alpha_0 || \alpha_1 || \cdots || \alpha_{j-1} || \alpha'_j, \tag{6}$$

where the notation $a||b$ means that string a is concatenated with string b, and α_i $(0 \le i \le j-1)$ are runs with lengths d_i respectively, α'_j is the remaining string. We have

$$\begin{aligned} \ell_1 =& \ell_0 - d_0, \ \cdots, \ \ell_{j-1} = \ell_{j-2} - d_{j-2}, \\ \ell_j =& \ell_{j-1} - d_{j-1} \end{aligned} \tag{7}$$

where $\ell_j \le k < \ell_{j-1}$. Note that the length of string α'_j is ℓ_j. Hence, we get

$$z' = \begin{cases} (\alpha'_j)_2, & j \text{ is even}, \\ 2^{\ell_j} - 1 - (\alpha'_j)_2, & j \text{ is odd}, \end{cases}$$

Thus,

$$z \equiv \begin{cases} r[\ell_0] & -r[\ell_1] + r[\ell_2] + \cdots + (-1)^{j-1} r[\ell_{j-1}] \\ & +(\alpha'_j)_2, j \text{ is even}, \end{cases} \tag{8}$$

$$z \equiv \begin{cases} r[\ell_0] & -r[\ell_1] + r[\ell_2] + \cdots + (-1)^{j-1} r[\ell_{j-1}] \\ & +(\alpha'_j)_2 - 2^{\ell_j}, j \text{ is odd}, \end{cases} \tag{9}$$

Example 2. $n = 97 = (1100001)_2, k = 7; z = 3135 = (110000111111)_2, \ell_0 = 12$. *The runs in the left segment of Binary*(z) *are* $\alpha_0 = 11, \alpha_1 = 0000$. *Their lengthes are* $d_0 = 2, d_1 = 4$. *We have* $\ell_1 = \ell_0 - d_0 = 12 - 2 = 10, \ell_2 = \ell_1 - d_1 = 10 - 4 = 6$. *Since* $\ell_2 = 6 < 7 = k$, *we get* $j = 2$, $\alpha'_2 = 111111$. *Therefore,* $z' = (\alpha')_2 = (111111)_2 = 63$. *Thus,* $z = 2^{\ell_0} - 2^{\ell_1} + z' = 2^{12} - 2^{10} + 63 \equiv 22 - 54 + 63 = 31 \bmod 97$.

Table 2: The algorithm for Method-2

INPUT: $n, k = \text{BitLength}(n), 0 \le z < 2^{2k}$, and $\mathbb{T} = \{r[2k-1], r[2k-2], \cdots, r[k]\}$.
OUTPUT: $z \bmod n$.
If $z < n$, then return z.
If $\text{BitLength}(z) = k$, then return $z - n$.
$s \leftarrow \text{Binary}[z]$, $\ell \leftarrow \text{BitLength}[z]$, $y \leftarrow 1$, $r \leftarrow r[\ell]$, $d \leftarrow 0$, $t \leftarrow 0$.
For i from $\ell - 1$ downto 0 do
$b \leftarrow \text{StringTake}[s, \{i\}]$.
If $b = y$, then $d \leftarrow d + 1$.
$\ell \leftarrow \ell - d$, $t \leftarrow t + 1$, $r \leftarrow r + (-1)^t r[\ell]$.
If $\ell > k$, then $y \leftarrow \text{Mod}(y + 1, 2)$, $d \leftarrow 0$.
$\alpha \leftarrow \text{StringTake}\ [s, -\ell]$.
If $\text{Mod}\,(t, 2) = 0$, then $r \leftarrow r + (\alpha)_2$, else $r \leftarrow r + (\alpha)_2 - 2^\ell$.
Break.
While $r \ge n$ do: $r \leftarrow r - n$.
While $r < 0$ do: $r \leftarrow r + n$.
Return r.

4.3 Complexity Analysis

To obtain $\ell_0, \cdots, \ell_{j-1}, z'$, it requires only a handful of less-expensive bit operations. Since $\ell_0, \cdots, \ell_{j-1}$ is ordered, *i.e.*, $\ell_0 > \ell_1 > \cdots > \ell_{j-1}$, the cost of looking up $r[\ell_0], \cdots, r[\ell_{j-1}]$ in $\mathbb{T}$ is negligible. There are j additions for computing r. Since $|r[t]| \le \lfloor n/2 \rfloor, t \in \{\ell_0, \cdots, \ell_{j-1}\}$, we have

$$|r| \le (j+2)\lfloor n/2 \rfloor < \left(\left\lfloor \frac{j+2}{2} \right\rfloor + 1 \right) n.$$

That means it requires at most $\lfloor \frac{j+2}{2} \rfloor$ subtractions for computing $r \bmod n$. In total, the method needs to perform $\lfloor \frac{3j}{2} \rfloor$ additions of k-bit integers. We shall see that $j \approx \lfloor k/2 \rfloor$. That means Method-2 has the similar performance as Method-1.

We now give a comparison between Method-2 and Barrett's reduction. The computation of $\lfloor z/b^i \rfloor \cdot \mu$ dominates the cost of Barrett's reduction. It requires a multiplication. For convenience, we suppose that it is a multiplication of k-bit integers.

The Method-2 requires more cost for bit scans if the cost for one byte scan is considered to be approximately equal to that for one bit scan. But we here stress that the whole cost for bit scans is less than the cost for an addition of k-bit integers.

The quantity j is of great importance to the comparison. Clearly, $j \le k$. If the left segment of $\text{Binary}(z)$ is $\underbrace{1010\cdots10}_{k-\text{bit}}$, then $j = k$. Given a random $2k$-bit integer z, it is expected that there are about k runs and k 1's. Thus, we have $j = \lfloor k/2 \rfloor$. That means the new reduction is faster than Barrett's reduction at the expense of a little storage. The storage requirement in such case is acceptable to most devices at the time.

5 A Fast Reduction Method

As we mentioned previously, Method-1 is inappropriate for dealing with the string $11\cdots1$, whereas Method-2 can deal efficiently with such a string. Method-2 is not as efficient as Method-1 to deal with the string $1010\cdots10$. When hundreds of modular multiplications are required for modular exponentiation, it is better to use the two methods alternatively. Since they require a same pre-computed table, we can combine these two methods. We now present a description of such a combined reduction method.

5.1 A Combined Reduction Algorithm

Suppose that n is the modular, $0 \le z < 2^{2k}$, $k = \text{BitLength}(n)$ and $\mathbb{T}$ is the pre-computed table. To compute $z \bmod n$, the combined reduction method proceeds as follows.

1) Set Υ to be the left segment of $\text{Binary}(z)$ such that the length of the right segment equals to k.
2) Count the amount of 1's in Υ and denote it by ϕ.
3) Count the amount of runs in Υ and denote it by ψ.
4) If $\phi \le \psi$ then use Algorithm-1. Otherwise, use Algorithm-2.

5.2 Refined Algorithm

It is possible to refine the above algorithm. For example, considering a segment of $(101010111101)_2$. For this string, $\phi = 8$ and $\psi = 9$. So Algorithm-1 will be used. However, it is easy to see that the right part of the string is better to use Algorithm-2. So it is better to use Algorithm-1 for first 6 bits and use Algorithm-2 for last 6 bits. In general, if we have a long run of 1, then we should use Algorithm-2 for that run.

The following algorithm can be used to calculate $z \bmod n$, where $n < z < n^2$.

Table 3: Comparison between Barrett's reduction and Method-2

	arithmetic operation (k-bit integers)	pre-computation	byte/bit scans
Barrett's reduction	1 multiplication, 3 additions	value μ	$k/8$ byte
Method-2	$\lfloor \frac{3k}{4} \rfloor$ additions	table $\mathbb{T}$ (k items)	k bit

1) Set $\ell_0 = \text{BitLength}[z]$. Set Υ to be the left segment of Binary(z) such that the length of the right segment equals to k. Count the amount of 1's in Υ and denote it by ϕ. If $\phi \geq \lfloor k/2 \rfloor$, then flip all bits of Binary(z). Denote the new number by $\hat{z}$. Here $z = (2^{\ell_0} - 1) - \hat{z}$. In such case, the number of 1's in the corresponding left segment of $\hat{z}$ is less than $\lfloor k/2 \rfloor$. So, we consider $\hat{z} \bmod n$. For convenience, we now assume that $\phi \leq \lfloor k/2 \rfloor$.

2) Count runs in Υ to obtain $R = (l_0, r_0;\ l_1, r_1; \ldots; l_j, r_j)$, where l_0 is the length of the first run of 1 in Υ and r_0 is the length of the first run of 0 in Υ, $\ldots$, l_j is the length of the last run of 1 in Υ and r_j is the length of the last run of 0 in Υ. Here $l_i \geq 1$ for $0 \leq i \leq j$ and $r_i \geq 1$ for $0 \leq i \leq j-1$ while $r_j \geq 0$.

3) Let $\ell_t = k + \sum_{i=t}^{j}(l_i + r_i)$, $0 \leq t \leq j$. For t from 0 to j calculate S_t: if $l_t \leq 2$, $S_t = \sum_{m=\ell_t - l_t + 1}^{\ell_t} r[m-1]$; if $l_t > 2$, $S_t = r[\ell_t] - r[\ell_t - l_t]$.

4) Compute $LS = \sum_{t=0}^{j} S_t$ which can be used to calculate $z \bmod n$.

Note that the refined algorithm only needs to look up the pre-computation table $1 + \lfloor k/2 \rfloor$ times at most, *i.e.*, it requires about $\lfloor k/2 \rfloor$ additions of k-bit integers *at worst*. Since Barrett's reduction requires one multiplication of k-bit integers, the method is expected to be almost twice as fast as the Barrett's reduction.

Example 3. *Suppose* $z = 58809 = (1110010110111001)_2$, $n = 267 = (100001011)_2$. *Then* $k = 9$, $\Upsilon = (1110010)$, $R = (3, 2; 1, 1)$. *Therefore* $S_0 = r[16] - r[13] = 121 - 182 = -61, S_1 = r[10] = -44, LS = -61 - 44 = -105$. *So* $z = -105 + (110111001)_2 = -105 + 441 = 69 \bmod 267$.

6 Implementation Tips

Some experiments on modular reduction algorithms have been implemented, including the common lookup table reduction, the refined run-based reduction, Montgomery's reduction, Barret's reduction, the improved Barret reduction (see [3]) and the general repeated square reduction for the computation $c^d \bmod n$, where

$c =$ 55103280959622143570402130367663431846883890024225365746631236013125897340714776982730249289966488343996755920163957112016132956975401238007039707639868810208777108408089829058605678271696502129955757569123179449702471331787304364959839519775265074084061593327434500118603083495853207768231485190054148583981,

$d =$ 17970154009029862760662344073406083538245587958989134228820996621710832903953558853778906950976745158065143728393505657901184045798332028289815093774137325178448521127388065678503478658724581654937781809973937551742257916140835853898828972640247878231859992853336005115520383724262443403384025327820646467533,

$n =$ 13506641086599522334960321627880596993888147560566702752448514385152651060485953383394028715057190944179820728216447155137368041970396419174304649658927425623934102086438320211037295872576235850964311056407350150818751067659462920556368552947521350085287941637732853390610975054433499981115005697723689092756 3.

n is just the RSA-1024 number. The programming codes are written in Wolfram language. Nevertheless, their performances were not as expected strictly. It means the current high level languages cannot make the most of bit, byte or run scanning. That is to say, the underlying assembly language should be exploited for Montgomery's reduction, Barret's reduction and run-based reduction.

7 Conclusion

A new modular reduction method based on lookup table is introduced, which requires less arithmetic operations at the expense of a little storage. We show that the new reduction is almost twice as fast as Barrett's reduction. Interestingly, the method scans bit-by-bit. This feature makes it more portable and more suitable for small devices.

Acknowledgements

We thank the National Natural Science Foundation of China (Project 61411146001). The authors gratefully acknowledge the reviewers for their valuable suggestions.

References

[1] P. Barrett, "Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor," in *Proceedings of 6th Annual Cryptology Conference, Advances in Cryptology (CRYPTO'86)*, pp. 311–323, Aug. 1987.

[2] A. Bosselaers, R. Govaerts, and J. Vandewalle, "Comparison of three modular reduction functions," in *Proceedings of 13th Annual Cryptology Conference, Advances in Cryptology (CRYPTO'93)*, pp. 175–186, Aug. 1993.

[3] Z. J. Cao and X. J. Wu, "An improvement of the barrett modular reduction algorithm," *International Journal of Computer Mathematics*, vol. 91, no. 9, pp. 1874–1879, 2014.

[4] V. Dupaquis and A. Venelli, "Redundant modular reduction algorithms," in *Proceedings of 10th IFIP WG 8.8/11.2 International Conference on Smart Card Research and Advanced Applications (CARDIS'11)*, pp. 102–114, Sep. 2011.

[5] C. Guo, C. C. Chang, and S. C. Chang, "A secure and efficient mutual authentication and key agreement protocol with smart cards for wireless communications," *International Journal of Network Security*, vol. 20, no. 2, pp. 323–331, 2018.

[6] D. Hankerson., A. Menezes, and S. Vanstone, *Guide to Elliptic Curve Cryptography*, 2004. (`http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.394.3037&rep=rep1&type=pdf`)

[7] S. Hong, S. Oh, and H. Yoon, "New modular multiplication algorithms for fast modular exponentiation," in *Proceedings of International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT'96)*, pp. 166–177, May 1996.

[8] L. C. Huang, T. Y. Chang, and M. S. Hwang, "A conference key scheme based on the diffie-hellman key exchange," *International Journal of Network Security*, vol. 20, no. 6, pp. 1221–1226, 2018.

[9] S. Kawamura and K. Hirano, "A fast modular arithmetic algorithm using a residue table," in *Proceedings of International Conference on the Theory and Application of Cryptographic Techniques, Advances in Cryptology (EUROCRYPT'88)*, pp. 245–250, May 1988.

[10] C. Lim, H. Hwang, and P. Lee, "Fast modular reduction with precomputation," in *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JW-ISC'97)*, pp. 65–79, Oct. 1997.

[11] Y. J. Liu, C. C. Chang, and S. C. Chang, "An efficient and secure smart card based password authentication scheme," *International Journal of Network Security*, vol. 19, no. 1, pp. 1–10, 2017.

[12] D. Mahto and D. K. Yadav, "Performance analysis of rsa and elliptic curve cryptography," *International Journal of Network Security*, vol. 20, no. 4, pp. 625–635, 2018.

[13] P. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, no. 44, pp. 519–521, 1985.

[14] B. Parhami, "Analysis of tabular methods for modular reduction," in *Proceedings of 28th Asilomar Conference Signals, Systems, and Computers*, pp. 526–530, Nov. 1994.

[15] B. Parhami, "Modular reduction by multi-level table lookup," in *Proceedings of Midwest Symposium on Circuits and Systems (MWSCAS'97)*, pp. 381–384, Aug. 1997.

[16] C. Y. Tsai, C. Y. Yang, I. C. Lin, and M. S. Hwang, "A survey of e-book digital right management," *International Journal of Network Security*, vol. 20, no. 5, pp. 998–1004, 2018.

[17] C. Walter, "Faster modular multiplication by operand scaling," in *Proceedings of 11th Annual Cryptology Conference, Advances in Cryptology (CRYPTO'91)*, pp. 313–323, Aug. 1991.

[18] E. Win, S. Mister, B. Preneel, and M. Wiener, "On the performance of signature schemes based on elliptic curves," in *Proceedings of Algorithmic Number Theory*, pp. 252–266, June 1998.

Zhengjun Cao is an associate professor with the Department of Mathematics, Shanghai University. He received his Ph.D. degree in applied mathematics from Academy of Mathematics and Systems Science, Chinese Academy of Sciences. He had served as a post-doctor in Computer Sciences Department, Université Libre de Bruxelles. His research interests include cryptography, discrete logarithms and quantum computation.

Zhen Chen is currently pursuing his M.S. degree from Department of Mathematics, Shanghai university. His research interests include information security and cryptography.

Ruizhong Wei is a professor with the Department of Computer Science, Lakehead University, Canada. He received his Ph.D. degree in applied mathematics from Waterloo University. His research interests include combinatorics, algebraic code, algorithm design and analysis.

Lihua Liu is an associate professor with the Department of Mathematics, Shanghai Maritime University. She received her Ph.D. degree in applied mathematics from Shanghai Jiao Tong University. Her research interests include combinatorics and cryptography.