A Fast Modular Reduction Method

Zhengjun Cao[1,*], Ruizhong Wei[2], Xiaodong Lin[3]

[1]Department of Mathematics, Shanghai University, China. *caozhj@shu.edu.cn

[2]Department of Computer Science, Lakehead University, Canada.

[3]Business and Information Technology, University of Ontario Institute of Technology.

Abstract

We put forth a lookup-table-based modular reduction method which partitions the binary string of an integer to be reduced into blocks according to its runs. Its complexity depends on the amount of runs in the binary string. We show that the new reduction is almost twice as fast as the popular Barrett's reduction, and provide a thorough complexity analysis of the method.

Keywords. Barrett's reduction; Montgomery's reduction; lookup-table-based reduction; run-length-based reduction.

1 Introduction

The performance of public key cryptographic schemes depends heavily on the speed of modular reduction. Among current modular reduction algorithms, Barrett's reduction and Montogomery's reduction are two popular algorithms. Montgomery's reduction [9] is not efficient for a single modular multiplication, but can be used effectively in computations where many multiplications are performed for given inputs. Barrett's reduction [1] is applicable when many reductions are performed with a single modulus. Both two methods are similar in that expensive divisions in classical reduction are replaced by less-expensive multiplications.

In 1993, A. Bosselaers et al [2] reported that classical reduction, Barrett's reduction, and Montgomery's reduction have their specific behaviors resulting in a specific field of application. In 1998, the authors [13] reported that the difference between Montgomery's reduction 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, V. Dupaquis and A. 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 2013, Cao and Wu [3] proved that the base $b \geq 3$ in Barrett's reduction can be replaced by the usual

base 2. The improvement solves the data expansion problem in Barrett's reduction so as to give a little of cost saving.

In order to further speed up modular reduction, lookup table method has been adopted by several researchers [6–8, 10–12]. If the size of a pre-computed table is manageable, the method is very effective. These modular reductions partition the binary string of an integer to be reduced into fixed-length blocks such as 32 bits or 64 bits. In 1997, C. Lim et al [8] experimented on Montgomery's reduction, classical reduction, Barrett's reduction and some reduction algorithms using lookup table. It reported [8] that their lookup-table-based reductions run almost two to three times faster on a workstation than Montgomery's reduction. Although the experimental results are interesting, they did not present a complexity analysis of these reductions.

The principles behind the existing reduction algorithms can be summarized as follows.

- *Division.* For example, the classical reduction adopts this principle.
- *Multiplication.* Barrett's reduction and Montgomery's reduction adopt this principle.
- *Addition* [look up table according to *fixed-length blocks*]. All current reductions using lookup table take fixed-length blocks such as 32 bits or 64 bits.

The last principle, intuitively, is more applicable because it eliminates multiplications although it requires a moderate size table and a number of additions.

In this paper, we put forth a new reduction method based on the following principle: *addition* [look up table according to *runs*]. Unlike the current lookup-table-based method, the new method partitions the binary string of an integer to be reduced into blocks according to its runs. Its performance depends essentially on the amount of runs or 1's in the left segment of the binary string. The new method needs less additions in comparison to the current lookup-table-based method. We will 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 $zR^{-1} \mod p$ for an input $z < pR$. If $p' = -p^{-1} \mod R$, then $c = zR^{-1} \mod p$ can be obtained via

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

Given $x \in [0, p)$, let $\tilde{x} = xR \mod p$. Define $\mathrm{Mont}(\tilde{x}, \tilde{y}) = (\tilde{x}\tilde{y})R^{-1} \mod p = (xy)R \mod p$. The transformations

$$x \mapsto \tilde{x} = xR \mod p, \text{ and } \tilde{x} \mapsto \tilde{x}R^{-1} \mod p = x$$

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 [5]. 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 $0 \le z < b^{2k}$. 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

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

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 $\alpha = \frac{z}{b^{k-1}} - \left\lfloor \frac{z}{b^{k-1}} \right\rfloor$, $\beta = \frac{b^{2k}}{p} - \left\lfloor \frac{b^{2k}}{p} \right\rfloor$. Then $0 \le \alpha, \beta < 1$ and

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

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

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

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

Therefore, we obtain $\hat{q} \le q \le \hat{q} + 2$. 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 $\hat{r}$.

2.3 Current lookup-table-based modular reductions

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

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

where $0 \le z_j < b, j = 0, \cdots, 2k-1$, $A[i] = b^{k+i} \mod n \, (0 \le i \le k-1)$ are computed and stored in advance. For example, let $L = 32$ which equals to the bit-length of a word. In 1997,

C. Lim et al [8] suggested taking $b = 2^{32}$ and storing the following values:

$$\begin{aligned} n_a[j] &= jb^k \mod n \text{ for } 0 \le j < 128, \quad n_b[j] = j2^7 b^k \mod n \text{ for } 0 \le j < 128, \\ n_c[j] &= j2^{14} b^k \mod n \text{ for } 0 \le j < 128, \; n_d[j] = j2^{21} b^k \mod n \text{ for } 0 \le j < 128, \\ n_e[j] &= j2^{28} b^k \mod n \text{ for } 0 \le j < 102. \end{aligned}$$

They experimented on Montgomery's reduction, classical reduction, Barrett reduction and some lookup-table based reductions. It reported [8] that: (1) modular reduction takes considerably more time than multiplication; (2) Montgomery's algorithm and the combined lookup-table-based reduction give almost the same performance; (3) the lookup-table-based reductions L224, L624 and L1696 [see Ref. 8] run almost two to three times faster on a workstation than Montgomery's reduction. However, this method does not give much improvement on a PC.

The experimental results in Ref.[8] are interesting. But they have not provided a complexity analysis of these reductions. Even worse, they have not specified these methods. For example, in L624 method [see Ref. 8], if Bitlength$(n) = 1024$ then the formula becomes

$$\sum_{t=0}^{1023} n_t b^t + \sum_{i=0}^{31} (n_a[j_{i1}] + n_b[j_{i2}] + n_c[j_{i3}] + n_d[j_{i4}] + n_e[j_{i5}])\, 2^{32i} \mod n.$$

It seems wrong to store $n_e[j] = j2^{28} b^k \mod n$ for $0 \le j < 102$. In our opinion, the amount of pre-computed values should be 4×2^8 where the base is 32, not 624, *if we not count the parity bit in each byte as usual*. By the way, according to their presentation the amount of stored values is 128+128+128+128+102=614, not 624.

3 Basic lookup-table-based modular reduction

The idea behind the basic lookup-table-based 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 follows (see Table 1).

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

ℓ	$2k-1$	$2k-2$	$\ldots$	k
$r[\ell]$	$2^{2k-1} \mod n$	$2^{2k-2} \mod n$	$\ldots$	$2^k \mod n$

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

3.2 Basic method (Method-1)

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

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

where $z_j \in \{0,1\}, j = 0, \cdots, 2k-1$, $r[k+i] = 2^{k+i} \mod n \, (0 \leq i \leq k-1)$.

Example 1. $n = 97 = (1100001)_2, k = 7$ (bit-length), $z = 3135 = (110000111111)_2, l = 12$. Look up a pre-computed table for the values $r[11] = 2^{11} \mod n = 11$ and $r[10] = 2^{10} \mod n = 54$. It gives

$$z = 3135 \equiv r[11] + r[10] + (111111)_2 = 11 + 54 + 63 \equiv 31 \mod 97.$$

Algorithm-1

INPUT: $n, k = \text{BitLength}(n), 0 \leq z < 2^{2k}$, and $\mathbb{T} = \{r[2k-1], r[2k-2], \cdots, r[k]\}$.

OUTPUT: $z \bmod n$.

1. If $z < n$, then return(z).
2. If BitLength(z) $= k$, then return $(z - n)$.
3. $s \leftarrow$ Binary(z), $r \leftarrow 0$. For i from BitLength(z) $- 1$ downto k do

   If $s[i] = 1$, $r \leftarrow r + r[i]$.
4. $r \leftarrow r + \sum_{j=0}^{k-1} s[j] 2^j$.
5. While $r \geq n$ do: $r \leftarrow r - n$.
6. While $r < 0$ do: $r \leftarrow r + n$.
7. Return (r).

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 $|r[\ell]| \leq \lfloor n/2 \rfloor$. Hence, it requires $\lfloor k/4 \rfloor$ subtractions to compute $r \mod n$. In total, Method-1 requires the cost of performing $\lfloor \frac{3k}{4} \rfloor$ additions of k-bit integers.

In algorithm-1, addition happened for all values $r[k+i]$ corresponding to $z_{k+i} = 1 \, (0 \leq i \leq k-1)$. In the worst case,

$$z_k = z_{k+1} = \cdots = z_{2k-1} = 1,$$

it has to look up the pre-computed table and do addition k times. Clearly, Method-1 is inappropriate for this case.

4 Run-length-based modular 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 idea behind the new modular reduction

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

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

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

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

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

The following example explains vividly the idea behind this method.

Example 2. $n = 97 = (1100001)_2, k = 7$ (bit-length), $z = 3135 = (110000111111)_2, \ell_0 = 12$.

Table 2: An example for Method-2

$n = 97 \to 1100001$	k=7	
$z = 3135 \to 110000111111$	$\ell_0 = 12$	$r[\ell_0] = 2^{\ell_0} \equiv 22 \mod 97$
$\overset{\text{flip}}{- - --} \dashrightarrow 1111000000$	$\ell_1 = 10$	$r[\ell_1] = 2^{\ell_1} \equiv 54 \mod 97$
$\overset{\text{flip}}{- - --} \dashrightarrow 111111$	$\ell_2 = 6\,(< k)$	$z' = (111111)_2$
		$(r[\ell_0] - 1) - (r[\ell_1] - 1) + (111111)_2 = 31$

4.2 Description of the new modular reduction (Method-2)

To obtain indexes $\ell_0, \cdots, \ell_{j-1}$ and z' in Eq.(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

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

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,

$$z \equiv \begin{cases} r[\ell_0] - r[\ell_1] + r[\ell_2] + \cdots + (-1)^{j-1} r[\ell_{j-1}] + (\alpha'_j)_2, & j \text{ is even}, \quad (8) \\ r[\ell_0] - r[\ell_1] + r[\ell_2] + \cdots + (-1)^{j-1} r[\ell_{j-1}] + (\alpha'_j)_2 - 2^{\ell_j}, & j \text{ is odd}, \quad (9) \end{cases}$$

Example 3. $n = 97 = (1100001)_2,\ k = 7;\ z = 3135 = (110000111111)_2,\ \ell_0 = 12$. 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, $z = 2^{\ell_0} - 2^{\ell_1} + z' = 2^{12} - 2^{10} + 63 \equiv 22 - 54 + 63 = 31 \mod 97$.

Remark 1. We know run-length encoding is a very simple and useful form of data compression in which runs of data are stored as a single data value rather than as the original run. But nobody, to the best of our knowledge, has mentioned such a run-length reduction method to this day.

4.3 Complexity analysis of Method-2

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

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

Algorithm-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: $z \bmod n$.

1. If $z < n$, then return(z).
2. If BitLength$(z) = k$, then return $(z - n)$.
3. $s \leftarrow$ Binary$[z]$, $\ell \leftarrow$ BitLength$[z]$, $y \leftarrow 1$, $r \leftarrow r[\ell]$, $d \leftarrow 0$, $t \leftarrow 0$.
4. For i from $\ell - 1$ downto 0 do

   4.1 $b \leftarrow$ StringTake$[s, \{i\}]$.

   4.2 If $b = y$, then $d \leftarrow d + 1$.

   4.3 $\ell \leftarrow \ell - d$, $t \leftarrow t + 1$, $r \leftarrow r + (-1)^t r[\ell]$.

      4.3.1 If $\ell > k$, then $y \leftarrow \text{Mod}(y + 1, 2)$, $d \leftarrow 0$.

      4.3.2 $\alpha \leftarrow$ StringTake $[s, -\ell]$.

         If Mod $(t, 2) = 0$, then $r \leftarrow r + (\alpha)_2$, else $r \leftarrow r + (\alpha)_2 - 2^\ell$.

         Break.
5. While $r \ge n$ do: $r \leftarrow r - n$.
6. While $r < 0$ do: $r \leftarrow r + n$.
7. Return (r).

That means it requires at most $\left\lfloor \frac{j+2}{2} \right\rfloor$ subtractions for computing $r \mod n$. In total, the method requires to perform $\left\lfloor \frac{3j}{2} \right\rfloor$ 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.

Table 3: Cost 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	$\left\lfloor \frac{3j}{2} \right\rfloor$ additions	table $\mathbb{T}$ (k items)	k bit

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, $j \leq k$. If the left segment of 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 (see Appendix 1). 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. By the way, the storage space is about 1 M for the pre-computed table with respect to a modular of 1024 bits, which is acceptable to most devices at the time.

5 A fast modular reduction method

As mentioned earlier, Method-1 is inappropriate for handling 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 these two methods properly. Since they require a same pre-computed table, we can simply combine these two methods. We now present a description of such a combined reduction method.

5.1 A combined modular reduction algorithm

Suppose that n is the modular, $0 \leq z < 2^{2k}$, $k = \text{BitLength}(n)$ 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 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 $\phi \leq \psi$ then use Algorithm-1. Otherwise, use Algorithm-2.

5.2 Refined algorithm

It is possible to refine the above algorithm. For example, consider 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 $n < z < n^2$, for general situations.

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 an integer vector

$$R = (l_0, r_0;\ l_1, r_1; \ldots; l_j, r_j),$$

   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 $z \bmod n$.

Good performance. We here stress 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.

Remark 2. The theoretical result is so attractive that we shall carry out some experiments on this method on various platforms after we could solicit some skillful engineers.

Example 4. Suppose $z = 58809 = (1110010110111001)_2$ and $n = 267 = (100001011)_2$ such that $z < n^2$. Then $k = 9$ and $\Upsilon = (1110010)$. $R = (3, 2; 1, 1)$. Therefore

$$\begin{aligned} S_0 &= r[16] - r[13] = 121 - 182 = -61 \\ S_1 &= r[10] = -44 \\ LS &= -61 - 44 = -105. \end{aligned}$$

So $z = -105 + (110111001)_2 = -105 + 441 = 69 \bmod 267$.

6 Conclusion

In this paper, we introduce a fast modular reduction method based on lookup table 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. More significantly, our proposed method only needs a moderate size storage space, less than 1 M for 1024-bit modulus, which makes it more portable and more suitable for small devices such as smartphones.

References

[1] Barrett P: Implementing the Rivest Shamir and Adleman public key encryption algorithm on a standard digital signal processor. In: Advances in Cryptology-CRYPTO'86, LNCS, 263, pp. 311-323. Springer-Verlag (1987)

[2] Bosselaers A., Govaerts R., Vandewalle J.: Comparison of three modular reduction functions. In: Advances in Cryptology-CRYPTO'93, LNCS 773, PP. 175-186. Springer-Verlag (1993)

[3] Cao ZJ, Wu XJ: An improvement of the Barrett modular reduction algorithm. International Journal of Computer Mathematics. DOI:10.1080/00207160.2013.862237. Taylor & Francis (2013)

[4] Dupaquis V., Venelli A.: Redundant Modular Reduction Algorithms. In: Proc. of CARDIS 2011, LNCS, 7079, pp. 102-114. Springer-Verlag (2011)

[5] Hanerson D., Menezes A, Vanstone S.: Guide to elliptic curve cryptography. Springer-Verlag (2004)

[6] Hong S., Oh S., Yoon H.: New modular multiplication algorithms for fast modular exponentiation. In: Adcances in Cryptology-EUROCRYPT'96, LNCS 1070, pp. 166-177. Springer-Verlag(1996)

[7] Kawamura S., Hirano K.: A fast modular arithmetic algorithm using a residue table. In: Advances in Cryptology-EUROCRYPT'88, LNCS 330, pp. 245-250. Springer-Verlag (1988)

[8] Lim C., Hwang H., Lee P.: Fast modular reduction with precomputation. In: Proc. of 1997 Korea-Japan Joint Workshop on Information Security and Cryptology, pp. 65-79 (1997)

[9] Montgomery P: Modular multiplication without trial division. Mathematics of Computation, 44, pp. 519-521 (1985)

[10] Parhami B.: Analysis of tabular methods for modular reduction. In: Proc. 28th Asilomar Conf. Signals, Systems, and Computers. Pacific Grove, CA, 1994, pp. 526-530.

[11] Parhami B.: Modular reduction by multi-level table lookup. In: Proc. Midwest Symposium on Circuits and Systems, MWSCAS'97. IEEE. pp. 381-384 (1997)

[12] Walter C.: Faster modular multiplication by operand scaling. In: Advnaces in Cryptology-CRYPTO'91, LNCS 576, pp. 313-323. Springer-Verlag (1991)

[13] Win E., Mister S., Preneel B., Wiener M.: On the performance of signature schemes based on elliptic curves. Algorithmic Number Theory-ANTS-III. LNCS, 1423, pp. 252-266. Springer-Verlag (1998)

Appendix 1: Amount of runs or 1's in binary strings

```
(* The average amount of runs or 1's in binary strings *)
T1 = {}; T2 = {}; f = 511; t = 10; s = 32;
For[ h = 512, h < 1025, h += s, r = 0; d = 0;
    Do[k; str = Join[{1}, RandomInteger[1, f]]; (*set the leftmost bit as 1*)
            c = Count[str, 1]; (*count the amount of 1's*)
            j = 1; y = 1;
            For[ i = 2, i < Length[str] + 1, i += 1,
            If[Part[str, i] != y, j += 1];
            If[Mod[j, 2] == 0, y = 0, y = 1]]; (*count the amount of runs*)
            r += j; d += c, {k, t} ];
    AppendTo[T1, {h, r/t}]; AppendTo[T2, {h, d/t}]; f += s];
Print["Table-1----", T1]; Print["Table-2----", T2];
ListPlot[{T1, T2}, PlotMarkers ->
    {{\[FilledCircle], 12}, {\[FilledSmallSquare], 12}},
    PlotStyle -> {Red, Blue}, Joined -> {True, True},
    ImageSize -> Scaled[0.8], AxesLabel -> {"bits", ""}]
```

$$\text{Table-1----}\left\{\{512, \frac{1268}{5}\}, \{544, 279\}, \left\{576, \frac{1444}{5}\right\}, \left\{608, \frac{1488}{5}\right\}, \left\{640, \frac{3153}{10}\right\}, \left\{672, \frac{3321}{10}\right\}, \left\{704, \frac{3421}{10}\right\}, \left\{736, \frac{747}{2}\right\}, \left\{768, \frac{3813}{10}\right\}, \left\{800, \frac{3943}{10}\right\}, \left\{832, \frac{4137}{10}\right\}, \left\{864, \frac{2122}{5}\right\}, \{896, 447\}, \{928, 461\}, \left\{960, \frac{4879}{10}\right\}, \{992, 500\}, \{1024, 508\}\right\}$$

$$\text{Table-2----}\left\{\{512, \frac{1311}{5}\}, \left\{544, \frac{547}{2}\right\}, \left\{576, \frac{2873}{10}\right\}, \left\{608, \frac{2977}{10}\right\}, \left\{640, \frac{3183}{10}\right\}, \left\{672, \frac{1651}{5}\right\}, \left\{704, \frac{1772}{5}\right\}, \left\{736, \frac{1807}{5}\right\}, \left\{768, \frac{757}{2}\right\}, \left\{800, \frac{1974}{5}\right\}, \left\{832, \frac{4193}{10}\right\}, \left\{864, \frac{4283}{10}\right\}, \left\{896, \frac{4513}{10}\right\}, \{928, 466\}, \{960, 491\}, \left\{992, \frac{2497}{5}\right\}, \left\{1024, \frac{2569}{5}\right\}\right\}$$