Computing Mod Without Mod

Mark A. Will and Ryan K. L. Ko

Cyber Security Lab
The University of Waikato, New Zealand
{willm,ryan}@waikato.ac.nz

Abstract. Encryption algorithms are designed to be difficult to break without knowledge of the secrets or keys. To achieve this, the algorithms require the keys to be large, with some algorithms having a recommend size of 2048-bits or more. However most modern processors only support computation on 64-bits at a time. Therefore standard operations with large numbers are more complicated to implement. One operation that is particularly challenging to implement efficiently is modular reduction. In this paper we propose a highly-efficient algorithm for solving large modulo operations; it has several advantages over current approaches as it supports the use of a variable sized lookup table, has good spatial and temporal locality allowing data to be streamed, and only requires basic processor instructions. Our proposed algorithm is theoretically compared to widely used modular algorithms, before practically compared against the state-of-the-art GNU Multiple Precision (GMP) large number library.

1 Introduction

Modular reduction, also known as the modulo or mod operation, is a value within Y, such that it is the remainder after Euclidean division of X by Y. This operation is heavily used in encryption algorithms [6][8][14], since it can "hide" values within large prime numbers, often called keys. Encryption algorithms also make use of the modulo operation because it involves more computation when compared to other operations like add. Meaning that computing the modulo operation is not as simple as just adding bits.

Modern Intel processors have implemented the modulo operation in the form of a single division instruction [2]. This instruction will return both the quotient and remainder in different registers. Therefore the modulo operation can be computed in a single clock cycle, even though the nature of the operation requires more computation than add. However this instruction only supports values up to 64-bits, and recommended key sizes for encryption algorithms can be higher than 2048-bits [9]. This is why it is challenging to compute the modulo operation, because processors cannot directly support these large numbers.

We propose an algorithm for modular reduction which has been designed so that it is hardware friendly. It only requires simple operations such as addition and subtraction, unlike some other solutions which use multiplication or division [3]. This allows for less costly custom hardware implementations in terms of

area and power. While also making better use of the register inside processors. Our proposed algorithm only requires chucks of data at once, starting at the uppermost bits. Meaning the data can be streamed into a processor or custom hardware. This gives it good spatial and temporal locality, and reduces cache misses. Another useful property is that it guarantees the amount of bits required for computation will that be of the modulo value Y. While allowing for an arbitrary bit length of the input value X, unlike the algorithm shown in [5]. This allows implementations to have fixed bus sizes, giving better performance.

Our proposed algorithm also makes use of a precomputed lookup table, which supports variable sizes. Therefore allowing it to be customised for its application, and so that it fits inside cache. This lookup table also has the unique property of being able to be used for an arbitrary number of bits, regardless of the lookup tables key size.

Existing algorithms will be described in Section 2, including the two most popular algorithms, Barrett's reduction and Montgomery's reduction. Then in Section 3.1 we will describe our proposed algorithm in detail, giving the theorems behind the algorithm, and their respective proofs. Implementation techniques and remarks are given in Section 4, including the use of lookup tables. Before comparisons and performance results are given in Sections 5 and 6 respectively. Then future work is described in Section 7, concluding with Section 8.

2 Previously Known Techniques

This section will cover the popular and related modulus algorithms. The first two reduction techniques, Barrett and Montgomery are the most commonly used. A less common technique, the Diminished Radix Algorithm is mentioned, before a newly proposed algorithm is also described.

2.1 The Barrett Reduction

The general idea behind the Barrett Reduction [3][7] is

$$X \; mod \; Y \equiv X - \lfloor X/Y \rfloor Y$$

where X is divided by Y, floored (i.e. remove decimal places), and multiplied by Y. This value gives the closest multiple of Y to X, therefore we minus this value from X to give the modulus.

The actual equation proposed by Barrett [3] is modified so that

$$X \; mod \; Y \equiv X - \left\lfloor \frac{\frac{X}{b^{n-1}} u}{b^{n+1}} \right\rfloor Y$$

where $u = \left\lfloor \frac{b^{2n}}{Y} \right\rfloor$. This assumes that divisions by a power of b are computationally cheap, and that u is pre-computed. Therefore this is less expensive than the

general equation. It can be improved further so that partially multi-precision multiplication are used when needed, which gives

$$X \; mod \; Y \approx (X \; mod \; b^{n+1} - (m\hat{q} \; mod \; b^{n+1})) \; mod \; b^{n+1}$$

where $\hat{q}$ is an estimate of X/Y, resulting in the result being an estimate itself. Therefore a few subtractions of Y from the result can be required to give the correct modulo value.

2.2 The Montgomery Reduction

Montgomery Reduction [13] is one of the most common algorithms used for modulus reduction [citations]. The unique property of this algorithm is that it does not compute the modulus directly, but instead the modulus multiplied by a constant. An overview of the algorithm is shown in Algorithm 1 [15], where Y is odd, X is limited to $0 \leq X < Y^2$, and K is the number of bits.

Algorithm 1 Montgomery Reduction

Compute: $2^{-K}X \; mod \; Y$
1: $x = X$
2: **for** $k = 1$ *to* K **do**
3: **if** x *is odd* **then**
4: $x = x + Y$
5: $x = x/2$
return x

Algorithm 1 is a simple interpretation of the Montgomery Reduction, with further improvements required to match the performance of the Barrett Reduction. This is because currently the algorithm requires $2K^2$ single precision shifts and K^2 additions [15]. A small improvement is shown in Algorithm 2, where K must now satisfy the condition $2^K > Y$.

Algorithm 2 Montgomery Reduction Modification

Compute: $2^{-K}X \; mod \; Y$
1: $x = X$
2: **for** $k = 1$ *to* K **do**
3: **if** *the* k^{th} *bit is high* **then**
4: $x = x + 2^k Y$
return $x/2^K$

Now the number of single precision shifts has been reduced to $K^2 + K$, however the divide in the return statement can be done in one shift. This results in K^2 single precision shifts, and 1 right shift of size K.

There are other modifications of the Montgomery Reduction [10][4] for both software and hardware. And due to the number of variations, in this paper we will only compare our proposed algorithm against the two Montgomery Reductions algorithms, as shown in Algorithm 1 and 2.

2.3 The Diminished Radix Algorithm

The Diminished Radix Algorithm [12] has been designed so that certain moduli offer performance gains [15]. Because of this, it is not a universally applicable algorithm, so it will not be compared against in this paper.

2.4 Fast Modular Reduction Method

A newly proposed algorithm [5] by Cao et al. for modular reduction, which makes use of a fixed size lookup table similar to precomputed tables described in [11]. Even though [5] is not one of the main algorithms for modular reduction, it will be described in detail in this paper. This is because it shares some of the core theorems with our proposed algorithm in Section 3.1. Also [5] represents the common way of implementing a lookup table for modular reduction, and will help show why our proposed algorithm makes better use of a lookup table than other algorithms.

Cao et al. describe two algorithms, the second being an improvement over the first. Therefore we will discuss the first algorithm as shown in Algorithm 3, and just make note that the second algorithm is better for the worst-case situation.

Algorithm 3 Fast Modular Reduction Method

Compute: $X \; mod \; Y$

$k = bitlength(Y)$
$r(\alpha) = 2^{\alpha} \; mod \; Y$
$\mathbb{T} = \{r(2k-1), r(2k-2), \ldots, r(k)\}$

if $X < Y$ **then**
    return X
if $bitlength(X) = k$ **then**
    return $X - Y$
$s = binary(X)$
$m = 0$
for $i = bitlength(X) - 1$ *downto* k **do**
    if $s[i]$ *is high* **then**
        $m = m + \mathbb{T}[i-k]$
$m = m + \sum_{j=0}^{k-1} s[j]2^j$
while $m \geq Y$ **do**
    $m = m - Y$
while $m < 0$ **do**
    $m = m + Y$
return m

Algorithm 3 uses a precomputed lookup table $\mathbb{T}$, which contains the modulus answer for each bit from index k to $2k-1$. Then it loops through each bit index of X at or above k and adds the value from the lookup table. Once the loop is complete, it then adds the bits from index 0 to $k-1$ of X onto the result m. This is the main reduction, but some further can be required hence the last two while loops.

3 Proposed Algorithm

3.1 Overview

The algorithm we propose for calculating the modulus is shown in Algorithm 4, which computes $X \ mod \ Y$. The functions are shown before the pseudo code, were α for example denotes the parameter into the function. Lower case variables with a subscript represent bits at an index in the upper case variable, as shown by the definition of T. Finally $\hat{}$ denotes the variable is an array or set. These notations will be the same for the theorems and proofs in Section 3.3.

Algorithm 4

Compute: $X \ mod \ Y$

$Width(\alpha) = width \ of \ \alpha \ in \ terms \ of \ bits$
$Split(\alpha, w) = \{\sum_{i=0}^{w-1} \alpha_i 2^i, ..., \sum_{i=0}^{w-1} \alpha_{Width(\alpha)-w+i} 2^i\}$
$Num(\hat{\alpha}) = number \ of \ elements \ in \ \hat{\alpha}$
$T = \sum_{i=0}^{Width(T)-1} t_i 2^i, \ t_i \in \{0, 1\}$

$\hat{G} = Split(X, Width(Y))$
$N = Num(\hat{G}) - 1$
while $N > 0$ **do**
    $T = \hat{G}[N]$
    for $i = Width(Y) - 1$ *downto* 0 **do**
        $T = T << 1$
        while $t_{Width(Y)} = 1$ **do**
            $t_{Width(Y)} = 0$
            $T = T + (2^{Width(Y)} \ mod \ Y)$
    $\hat{G}[N-1] = \hat{G}[N-1] + T$
    while $\hat{G}[N-1]_{Width(Y)} = 1$ **do**
        $\hat{G}[N-1]_{Width(Y)} = 0$
        $\hat{G}[N-1] = \hat{G}[N-1] + (2^{Width(Y)} \ mod \ Y)$
    $N = N - 1$
while $\hat{G}[0] > Y$ **do**
    $\hat{G}[0] = \hat{G}[0] - Y$
return $\hat{G}[0]$

The bit shifting operation is the key feature of our algorithm, as it allows the use of a single precomputed value to find the modulus of all bits above the bit width of Y, unlike other solutions. We can also use multiple precomputed values in the form of a lookup table and use a larger bit shift to improve performance, which is described in Section 4.1. Where the algorithms in [5] require a precomputed value for each bit above the bit width of Y. This can be very costly as the precomputed values have a fixed size, where our algorithm can vary the number of precomputed values. The other key property of our algorithm is that it only reads data of X once (reads an element in $\hat{G}$ once), starting from the uppermost bits and working down to the 0^{th} bit. This gives our algorithm excellent spatial and temporal locality. In terms of a hardware implementation, it also means the data can be streamed from memory. Then because only a fixed amount of data is read and computed at a time, custom hardware can be faster, have better area usage, and be more power efficient. This guaranteed data width is another property that other solutions cannot offer easily.

3.2 Description

Before the algorithm is described in further detail, first the functions must be explained.

- **Width:** Given the parameter α, this function will return the minimum number of bits in α. Put simply, it results the index of the uppermost high bit, plus one. For example if 13 were inputted, the result would be 4.
- **Split:** The first parameter α is the value to be split, and the second w is a bit width value. This function splits α into a vector, so that each element is bit width w. If α cannot be be split up evenly (i.e. the bit width of α is not a multiple of w), then α can be padded with zero bits. For example splitting 35 into a vector with an element bit width of 4, results in $\{3, 2\}$. In terms of binary values, $35 = 100011_2$, so the result is $\{0011_2, 0010_2\}$ (note the padded zeros at index 1).
- **Num:** Number of elements in the vector $\hat{\alpha}$ after the split function.

The first line of Algorithm 4 splits X into a vector $\hat{G}$ so that the elements have the same width as Y. Then we set N to the uppermost index of $\hat{G}$, because we will process the elements in reverse. Once we enter the loop, we will set T to element N for ease of understanding the algorithm.

The For loop will count from 0, to the number of bits in Y. Each iteration we shift T left by one (i.e. double T). Then if an overflow occurs, meaning that the width of T is no longer equal to that of Y (i.e. the bit at index Width(Y) is high), we clear this overflow bit, and add $2^{Width(Y)} \ mod \ Y$ to T. This value we add should be precomputed so that it is already in modulo Y, therefore it costs just one addition operation. The reason we do this is because if an overflow occurs, we are guaranteed that T is greater than Y. So we add the modulo value of this overflow bit back to the answer, which keeps T the same bit width as Y. Therefore this means that T is, or is close to the correct value. When adding

$2^{Width(Y)}$ $mod\, Y$ to T, another overflow may occur, which requires us to repeat line 7 until no overflow occurs.

Note that clearing the overflow bit could also be achieved by subtracting Y from T. But depending on implementation, it is possible that multiple subtractions of Y are required to clear the overflow bit. Also subtraction would not allow the use of a lookup table which is described in Section 4.1.

Once the we have finished the For loop, we have therefore performed $Width(Y)$ number of shifts on T. Then we add T to the next element in $\hat{G}$. Whenever we perform an add, it is possible for an overflow to occur, so we have to include the loop on Line 11 to deal with this when adding T.

When we reach element 0, which are the lowermost bits of X, so we know that we are already close to the correct result, meaning we can stop the loop. Then some subtractions could be required before the correct result is required (depending on implementation). However implementing the algorithm as is, at most only one subtraction would be required.

We are now left with the correct answer in element 0, which can be returned. In order to prove that this algorithm will produce the correct modulus answer, we must now discuss the theorems and proofs that the algorithm is based upon.

3.3 Theorems and Proofs

The algorithms main operation is to double the value of T, and by doing this, we are also doubling the modulus. This is shown in Theorem 1. Note that it is important to find the modulus of $2M$ because by doubling M, the result could become greater than Y.

Theorem 1. *If $X\ mod\, Y = M$, then $2X\ mod\, Y = 2M\ mod\, Y$*

Proof. Given $X\ mod\, Y = M$ where $X, Y \in \mathbb{Z}$ and $M \in \mathbb{Z}_0^Y$

$$\frac{X}{Y} = Q + M \quad (Q = Quotient)$$
$$\therefore X\ mod\, Y \equiv M\ mod\, Y$$

$$\frac{2X}{Y} = 2(Q + M)$$
$$\frac{2X}{Y} = 2Q + 2M$$
$$\therefore 2X\ mod\, Y \equiv 2M\ mod\, Y$$

Because an integer represented in binary is made up of powers of 2s, it is said that the number is the sum of power of 2s. Therefore if we take the modulus of each power of 2 and sum them, it is equivalent to summing the power of 2s then taking the modulus, as shown in Theorem 2.

Theorem 2. *Given an integer X, which is the sum of power of 2s, then the sum of the modulus's of the power of 2s in Y, is equivalent to the modulus of X in Y.*

Proof. Given $X \; mod \, Y$ where $X, Y \in \mathbb{Z}$

$$\begin{aligned} X &= \sum_{i=0}^{n-1} x_i 2^i \quad (\text{where } x_i \in \{0,1\}) \\ \frac{x_i 2^i}{Y} &= Q_i + M_i \quad (\text{where } Q_i, M_i \in \mathbb{Z}) \\ x_i 2^i \; mod \, Y &\equiv M_i \; mod \, Y \\ \therefore X \; mod \, Y &\equiv \sum_{i=0}^{n-1} M_i \; mod \, Y \end{aligned}$$

Theorem 3 in the general context of the algorithm is the same as Theorem 1. However we require Theorem 3 because our proposed algorithm can use a lookup table (described in Section 4.1), which Theorem 1 does not make clear.

Theorem 3. *Given an integer X, which is to be shifted bit left by w, the modulus of X bit shifted by w is equivalent to finding the modulus of X after being bit shifted.*

Proof. Given $(X << w) \; mod \, Y = M$ where $X, Y, M, w \in \mathbb{Z}$

Because left bit shifting X by w is equivalent to doubling X w times, we can use Theorem 1 to prove that:

$$((X \; mod \, Y) << w) \; mod \, Y \equiv (X << w) \; mod \, Y \equiv M$$

The first 3 theorems are the basic underlying operations used in our algorithm. Now we need to prove the fundamental idea behind the algorithm. Theorem 4 proves the initial step of the algorithm, splitting up X into elements of an array/vector so that they can be concatenated together to give X. Then when finding the modulus of X, we can sum the modulus of each element (with bit shifting).

Theorem 4. *When finding $X \; mod \, Y$, if X is larger then Y in terms of number of bits, the modulus is equivalent to dividing X into elements of the same bit size as Y, follow by the bit shifting and summing the modulus values of each element.*

Proof. Given $X \; mod \; Y = M$ where $X, Y, M \in \mathbb{Z}$

$$Y = \sum_{i=0}^{k-1} y_i 2^i \quad (\text{where } y_i \in \{0,1\} \text{ and } k > 0)$$

$$X = \sum_{i=0}^{n-1} x_i 2^i \quad (\text{where } x_i \in \{0,1\} \text{ and } n \; mod \; k = 0)$$

$$\hat{X} = \{ \left(\sum_{i=n-k}^{n-1} x_i 2^i >> (n-k) \right), \; ... \; , \left(\sum_{i=0}^{k-1} x_i 2^i \right) \}$$

$$Y = \sum_{i=0}^{k-1} y_i 2^i \quad (\text{where } y_i \in \{0,1\} \text{ and } k > 0)$$

$$\begin{aligned} X &\equiv \bigg|\bigg|_{i=(n/k)-1}^{0} \hat{X}_i \\ &\equiv \sum_{i=0}^{(n/k)-1} (\hat{X}_i << ki) \end{aligned}$$

Because the concatenation of $\hat{X}$ is equivalent to X. Then by shifting each item in $\hat{X}$ to the correct bit position and summing, is equivalent to X. Therefore we can use Theorem 2 and 3 so that

$$\sum_{i=0}^{(n/k)-1} (\hat{X}_i << ki) \; mod \; Y \equiv X \; mod \; Y$$

Note: if $n \; mod \; k \neq 0$ then X can be padded with $0s$ until $n \; mod \; k = 0$.

Theorem 5 proves that a single precomputed value (or a lookup table), can be used to help find the modulus in each element greater than 0. This is an important property of the algorithm, as it allows the use of more precomputed values in the form of a lookup table.

Theorem 5. *When calculating the modulus off $\hat{X}_i$ in Y where $i > 0$, the modulus is equivalent to multiplying $\hat{X}_i$ by $\left(2^k \; mod \; Y << k(i-1)\right)$.*

Proof.

$$\begin{aligned} & \hat{X}_i << ki \; mod \; Y \\ \equiv \; & \hat{X}_i 2^{ki} \; mod \; Y \\ \equiv \; & (\hat{X}_i \; mod \; Y) \times (2^{ki} \; mod \; Y) \\ \equiv \; & (\hat{X}_i \; mod \; Y) \times (2^k << k(i-1) \; mod \; Y) \end{aligned}$$

Instead of multiplying (as in Theorem 5), we instead shift each individual bit in an element and add if an overflow occurs. These are proven to be equivalent in Theorem 6.

Theorem 6. *Left shifting $\hat{X}_i$ by ki in modulo Y where $i > 0$, is equivalent to shifting $\hat{X}_i$ one shift at a time. Then if the kth bit becomes high after any shift (or add operation), drop it, and add $2^k \bmod Y$ to $\hat{X}_i$. Therefore keeping the bit width of $\hat{X}_i$ constant, while remaining in modulo Y.*

Proof.

$$\begin{aligned} & \hat{X}_i \times 2^k i \bmod Y \\ \equiv\ & \hat{X}_i << ki \bmod Y \\ \equiv\ & (((\hat{X}_i << 1 \bmod Y) \ldots) << 1 \bmod Y) \end{aligned}$$

After each shift we put $\hat{X}$ back into modulo Y, and because $\hat{X}$ has k bits, if the kth bit becomes high (i.e. overflow), we know that $\hat{X}$ is definitely bigger than Y. Then by using Theorem 4 and 5, we can say that $\hat{X}$ is equivalent in modulo Y to adding $2^k \bmod Y$ to $\hat{X}$ (after dropping the kth bit).

Theorem 6 is the main theorem for this algorithms operation (i.e. producing the correct answer). However for implementation, it would suffer from performance issues. This is due to the amount of shifting and potential adding which we would need to perform. So instead of shifting an element to its correct position directly, we can just shift it by the number of bits in Y, then start shifting the next element as well. This is shown in Theorem 7, and in terms of performance of the algorithm, is the most important theorem.

Theorem 7. *If we start calculating the modulus of X in Y at $\hat{X}_{(n/k)-1}$, then instead of performing $k\,(((n/k)-1)-1)$ number of shifts, we can perform k shifts, then add $\hat{X}_{(n/k)-1}$ to $\hat{X}_{(n/k)-2}$. Repeating until we reach $\hat{X}_0$ which will contain the result.*

Proof.

$$\begin{aligned} & \left(\hat{X}_{(n/k)-1} << k((n/k)-2)\right) + \left(\hat{X}_{(n/k)-2} << k((n/k)-3)\right) \\ \equiv\ & \left(\hat{X}_{(n/k)-1} << k << k((n/k)-3)\right) + \left(\hat{X}_{(n/k)-2} << k((n/k)-3)\right) \\ \equiv\ & \left(\left(\hat{X}_{(n/k)-1} << k\right) + \left(\hat{X}_{(n/k)-2}\right)\right) << k((n/k)-3) \end{aligned}$$

3.4 Example

Below is an example on how to solve 1620 *mod* 11 on a 4-bit processor using our proposed algorithm.

$$1620 = 011001010100_2$$
$$11 = 1011_2$$
$$\hat{G} = \{0110_2,\ 0101_2,\ 0100_2\}$$

$T = \hat{G}[2]$
$= 0110_2$
$T = T << 1$
$= 1100_2$
$T = T << 1$
$= 1\ 1000_2$
$T = 1000_2 + (10000_2\ mod\ 1011_2)$
$= 1000_2 + 0101_2$
$= 1101_2$
$T = T << 1$
$= 1\ 1010_2$
$T = 1010_2 + 0101_2$
$= 1111_2$
$T = T << 1$
$= 1\ 1110_2$
$T = 1110_2 + 0101_2$
$= 1\ 0011_2$
$= 0011_2 + 0101_2$
$= 1000_2$

$T = T + \hat{G}[1]$
$= 1000_2 + 0101_2$
$= 1101_2$
$T = T << 1$
$= 1\ 1010_2$
$T = 1010_2 + 0101_2$
$= 1111_2$
$T = T << 1$
$= 1\ 1110_2$
$T = 1110_2 + 0101_2$
$= 1\ 0011_2$
$T = 0011_2 + 0101_2$
$= 1000_2$
$T = T << 1$
$= 1\ 0000_2$
$T = 0000_2 + 0101_2$
$= 0101_2$
$T = T << 1$
$= 1010_2$

$T = T + \hat{G}[0]$
$= 1010_2 + 0100_2$
$= 1110_2$
$T = T - 11$
$= 1110_2 - 1011_2$
$= 0011_2$

$\therefore$ **1620 mod 11 = 3**

4 Implementation

4.1 Lookup Table

A useful property of our algorithm is that is allows for the use of a variable sized lookup table. Lookup tables may not be suitable for all applications or functions, such as key generation, because the overhead in creating the table could be too expensive. However for applications where large amounts of data

must be computed with the same modulo value, such as smart-cards or secure tunnels, there is a performance gain. The algorithm in its most basic form already uses a lookup table. For each shift, if an overflow occurs, we add a precomputed value, else we add nothing. This gives a simple lookup table as shown in Table 1.

Table 1.

Key	**Value**
0	0
1	$2^k \ mod \ Y$

This can be extended to look at more bits at a time. For example, if we were to look at 2-bits, the lookup table would be that of Table 2.

Table 2.

Key	**Value**
00	0
01	$2^k \ mod \ Y$
10	$2^{k+1} \ mod \ Y$
11	$(2^k + 2^{k+1}) \ mod \ Y$

This allows us to shift by two bits at a time (instead of a single shift), but we still only require a single add. However this makes a software implementation slightly more difficult, because we cannot use the overflow bit anymore. Therefore we must look at the upper 2-bits of T before shifting. The current software implementation uses an and operation then a shift to get these upper bits. On a 64-bit processor, a lookup table with a key size of up to 64-bits, only requires the and and shift operations to be executed on a single 64-bit register, regardless of the size of Y. To possibly improve performance even more, one approach that could be explored is reversing the bits in each element (and keys), meaning only an and operation would be required.

Given that the lookup table is of variable size, it is important to make the whole table fit into the processors cache. This makes the lookup time require significantly less clock cycles than fetching from main memory, and thus improving performance. The size of Y has the biggest impact on the size of the lookup table. For example, if Y is a 2048-bit value, then each item in the table requires 2048-bits, plus the number of bits for each key.

4.2 Implementation Techniques

Using the algorithm proposed in Algorithm 4 for implementation on a standard processor can have some performance issues. This is because we are reliant on the number of bits in Y. For example, if we are using a 64-bit processor, and we have Y which has 130-bits. Then X will be divided into elements of 130-bits, which is 2x64-bit registers (A and B), with the remaining 2-bits needs to go into another register (C). Then if we were to use a lookup table with a 4-bit key. We would need to combine the 2-bit value from register C, with the upper 2-bits of register B.

One solution to solve this is to round the number of bits in Y up, so that it evenly divides the processors Arithmetic Logic Unit (ALU) width. In the example above, where Y had 130-bits, we would round it to 192-bits. This problem may also apply to custom hardware implementations, depending on the design. For example a smaller design may only use a 8-bit ALU, and could require Y to be rounded. But a design working on the full width of Y would not. Solving this problem is related to the design of the implementation, and this is just one proposed solution, of which they are many.

By rounding the bits in Y, this means that our final answer can require more than one subtraction of Y

4.3 Memory Access

This algorithm has been designed in such a way that memory access has been kept to a minimum. Because the lookup table can be configured so that it can be stored in cache. Once it is loaded, the only memory access required is the data for X. Then since the algorithm only works on segments of X at a time, it can load an element of X, process it, then load the next element. Therefore reducing the number of fetches (also cache misses), and improving performance. Each element loaded is the same number of bits as Y, meaning if Y is 2048-bits, then the processor only needs to load 2048-bits at a time. This allows the data to be streamed into the processor, which is also important for a custom hardware implementation.

5 Comparisons

5.1 Comparisons with Barrett Reduction

Comparing between our proposed algorithm and the Barrett Reduction is very difficult in theory. This is because the Barrett Reduction only seems to use a few operations. However when looking at implementation, it is the nature of these operations which cause performance issues. This is because when working with large numbers, for example 2048-bits, a 64-bit processor cannot simply execution a single instruction. Instead it must execution many instructions to compute the result.

Addition operations on large numbers are straight forward assuming the instruction set supports a carry bit. The words of the two input values can just added together. For example on Intel x86 processors, the *adc* instruction can be used to add two words together, plus add a carry bit if an overflow occurred on the previous *add* or *adc*. Therefore if we are computing $r = a + b$, our pseudo assembly code would be

$$\begin{gathered} add\ r[0],\ a[0],\ b[0] \\ adc\ r[1],\ a[1],\ b[1] \\ \ldots \\ adc\ r[n],\ a[n],\ b[n] \end{gathered}$$

where each word of a and b are added together. Therefore the number of instructions required is the number of words in the value. Subtraction can be achieved in a similar manner, by using the *sub* and *sbb* instructions.

Multiplication is not as simple as addition or subtraction, because each word in a needs to be multiplied by all words in b. One technique to implement this is using basic long multiplication. For example if we are computing 33×52 on a 4-bit processor, we get

$$\begin{array}{r} 0010\ 0001 \\ \times\ 0011\ 0100 \\ \hline 1000\ 0100 \\ +\ 0110\ 0011\ 0000 \\ \hline 0110\ 1011\ 0100 \end{array}$$

where we are only multiplying or adding 4-bits per cycle. Therefore in this example, we require 4 multiples, and 3 adds. This can get more complex if overflows occur when multiplying, often requiring more additions in modern instruction sets [2]. Scaling this up to multiplying two 2048-bit values on a 64-bit processor, 1024 multiplications and even more additions depending on the amount of overflows that occur. Ignoring other instructions such as fetch, move and store, just this simple multiplication requires over 2048 instructions. The number of registers within the processor will also impact the performance because large numbers (i.e. 2048-bits) can be difficult to fit into registers all at once.

This is the disadvantage of using the Barrett reduction, because even though the operations used seem simple, in reality they can be complicated to implement. The algorithm also makes use of the division operation which is more computationally intense than multiplication. However it depends on the value of b.

In contrast, our algorithm only uses simple operations like bit shifting and addition. It also uses memory efficiently as elements of the value are only accessed once. However the number of instructions required depends on the size of the key and lookup table. For example given $X\ mod\ Y$, if Y is only 1024-bits,

and X is 2048-bits, therefore the main reduction is computed on the upper 1024-bits of X. Then the lookup table key can be 8-bits, resulting in a size of approximately 258Kb (1024-bit values + keys). This means that 1024/8 lookups are required. The bulk of the instructions are adding the lookup values each time. In this case, on a 64-bit processor, 2048 add instructions are needed. This is therefore comparable to a single multiplication in the Barrett Reduction (because the multiplication will be computed on 2048-bits). The shifts required will also require approximately 2048 instructions, which makes equals two multiplication operations. There are other instructions that will be required of course, like some additions for joining the elements together, and some final subtractions to get to the correct result. But these heavy depend on the inputted values.

At this point it is not possible to definitely claim our proposed algorithm is better than the Barrett Reduction. However by analysing the instructions required, we can show that it will use less instructions when compared to the Barrett Reduction, because of the Barrett Reductions heavy use of multiplication and division instructions. Comparing implementations of both algorithms in software is a difficult approach. Because both algorithms would be needed to be implemented fairly, with neither having any better code than the other (i.e. optimisations). Therefore only once a fully operational hardware implementation of our algorithm is complete, will we be able to provide a definite answer.

5.2 Comparisons with Montgomery Reduction

Unlike the comparison for the Barrett Reduction, comparing our proposed algorithm against the Montgomery Reduction is more straight forward. This is because the main operation used in Algorithm 2 is addition, which is the same as our proposed algorithm. Given that an addition in Algorithm 2 only occurs if a bit is high, we will say that on average, half of the bits are high. Meaning if the input is 2048-bits, only 1024-bits are high. Which therefore requires 1024 addition operations. As discussed in the Barrett Reduction comparison, one addition uses many instructions on a standard processor. Therefore on a 64-bit processor, 16384 add instructions are required if the input is 2048-bit values. This is far more than our proposed algorithm would require, which was approximately 2048 add instructions, and approximately 2048 shift instructions, as shown in the Barrett comparison. The Montgomery Reduction also makes no guarantees on the number of bits required for each operation. For example, given $X \; mod \; Y$, where X is 4096-bits and Y is 1024-bits, then each add operation will need to compute over 4096-bits. However our proposed algorithm will only compute over 1024-bits at a time. This is a very useful property for hardware implementations, and for making efficient use of registers and lower level cache.

The Montgomery Reduction does not always give the correct answer first time, often requiring additional steps to compute the desired result. Where as our proposed algorithm will give the correct result after one run. Another point is that the Montgomery Reduction also requires a lot of shifting operations for the $2^k Y$ computation. Therefore using this simple theatrical comparison, our

proposed algorithm should allow for better implementations in both software and hardware over the Montgomery Reduction algorithms shown.

5.3 Comparisons with Fast Modular Reduction Method

This newly proposed algorithm in [5] is actually similar to the Montgomery Reduction in the sense that it looks at high bits. The difference is that it only processes the bits above bits of the modulus (Y), like our proposed algorithm. For example, given $X \ mod \ Y$, where X is 2048-bits and Y is 1024-bits, then it will only process the upper 1024-bits of X. Then for each high bit i, it uses a lookup table to get the result of $2^i \ mod \ Y$ and adds it to the result. Therefore if we again say that half of the bits are high, 512 add operations are required. The bit width of the result is not clearly stated, however given that only values are added, it has to be 2048-bits (the same as X). Meaning at least 8192 add instructions would be needed, but will probably be closer to 16384. Our proposed algorithm would use far less instructions, and it guarantees the width of the result. Also because it keeps the result at a fixed width, the amount of subtractions required should be less on average, but this depends on the input.

Comparing the lookup tables, this algorithm [5] has a fixed size lookup table. So for example, if Y is 2048-bits, then there must be 2048 entries in the table, each of a size of 2048-bits. Resulting in a size of approximately 4Mb. Also because of this, the bit width of the input X, can be no more than double the bit width of Y. This is because the lookup table only contains entries for the bits up to double that of Y, which for this example is 2^{2048} to 2^{4096}. This is a major limitation of this algorithm. However when looking at our lookup table, it can vary in size, and can support an arbitrary bit length of X. This is important to allow the table to fit in cache, and for devices which have limited storage. Using the same example where Y is 2048-bits, if the lookup table key is 8-bits, then the total size is approximately 0.5Mb. Our algorithm even allows for a lookup size of 1, if space is really limited.

Therefore our proposed algorithm is superior to that proposed in [5], both in terms of required instructions and the effectiveness of the lookup table. It is also a better option for hardware implementations because it can guarantee bit widths, and support varying sized lookup tables.

6 Performance Results

This algorithm is more suitable for a custom hardware implementation. However for early benchmarks, a software version will be compared against the state-of-the-art large number library, GNU Multiple Precision (GMP). Comparing our proposed algorithm against the state-of-the-art in software was an ideal way to gauge the performance of this algorithm. Because the GMP mod operation is highly optimised to run as fast as possible on many processor architectures. Also their development team would have chosen the algorithm they deemed the fastest for computing the modulus.

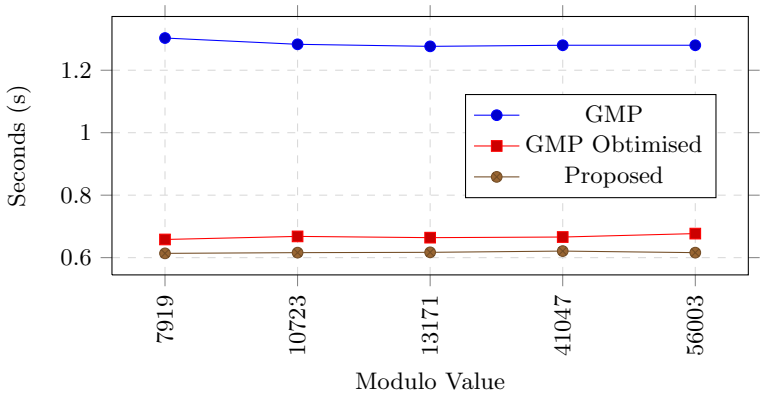

Fig. 1. 2048-bit *mod* 64-bit for 4000000 operations

The benchmarks were performed using an early 2013 MacBook Pro Retina 13", which is equipped with a 2.6GHz Intel Core i5-3230M, 8GB 1600MHz DDR3, and a 128GB solid state drive. GMP was setup using both a default installation on Mac OS X using the *brew* package manager [1], and an optimized configuration. Given $X \; mod \; Y$, the benchmarks in Figure 1 use a 2048-bit value for X and a value within the first 10000 primes for Y. Note that the results for our proposed algorithm include the time for computing the lookup table, which had a key size of 16-bits.

With the software implementation of our proposed algorithm still a work in progress. The current results shown in Figure 1 are more of a approximation of potential results instead of definite results. Therefore at this stage we cannot claim our software implementation is faster than GMP, even though early results are promising.

7 Future Work

The next step for this algorithm is to create an implementation on a Field-Programmable Gate Array (FPGA). Then to compare not only the performance, but also the area and power required, as well as looking into the effectiveness of side channel attacks. The current software implementation needs to be expanded to support an arbitrary length of input, for both X and Y. We also would like to experiment with compile time optimisations to give the algorithm better performance. This could lead to a large number library for software.

8 Conclusion

We propose an algorithm for computing the modulus operation which is superior to an other algorithms. This is because it has been designed to keep memory access at a minimum, decreasing the time the processor has to wait for data

to be fetched. It is more advanced than other algorithms using lookup tables, such as in [5] and [11], because our algorithm allows the table to be used on an arbitrary input, while supporting a variable size. This is a truly unique property of our solution when compared to others. Our proposed algorithm could greatly impact the cyber-security landscape, by improving the performance of many security applications we use today.

References

1. Brew. Online http://brew.sh (Accessed 21/08/2014).
2. Intel 64 and ia-32 architectures software developer's manual. Online http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf (Accessed 27/08/2014).
3. Paul Barrett. Implementing the rivest shamir and adleman public key encryption algorithm on a standard digital signal processor. In *Advances in cryptology—CRYPTO'86*, pages 311–323. Springer, 1987.
4. Lejla Batina and Geeke Muurling. Montgomery in practice: How to do it more efficiently in hardware. In *Topics in Cryptology—CT-RSA 2002*, pages 40–52. Springer, 2002.
5. Zhengjun Cao, Ruizhong Wei, and Xiaodong Lin. A fast modular reduction method. *IACR Cryptology ePrint Archive*, 2014:40, 2014.
6. Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer, 2002.
7. Vincent Dupaquis and Alexandre Venelli. Redundant modular reduction algorithms. In *Smart Card Research and Advanced Applications*, pages 102–114. Springer, 2011.
8. Craig Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.
9. Burt Kaliski. Twirl and rsa key size. RSA Laboratories Technical Note, 2003.
10. Taek-Won Kwon, Chang-Seok You, Won-Seok Heo, Yong-Kyu Kang, and Jun-Rim Choi. Two implementation methods of a 1024-bit rsa cryptoprocessor based on modified montgomery algorithm. In *Circuits and Systems, 2001. ISCAS 2001. The 2001 IEEE International Symposium on*, volume 4, pages 650–653. IEEE, 2001.
11. Chae Hoon Lim, Hyo Sun Hwang, and Pil Joong Lee. Fast modular reduction with precomputation. In *Proceedings of Korea-Japan Joint Workshop on Information Security and Cryptology (JWISC'97)*, pages 65–79. Citeseer, 1997.
12. Chae Hoon Lim and Pil Joong Lee. Generating efficient primes for discrete log cryptosystems.
13. Peter L Montgomery. Modular multiplication without trial division. *Mathematics of computation*, 44(170):519–521, 1985.
14. Ronald Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.
15. Tom St. Denis and Greg Rose. *BigNum Math: implementing cryptographic multiple precision arithmetic*. Syngress Publishing, 2006.