RSA Cryptography using Montgomery Multiplication

Tim Salomonsson
University of Victoria
Department of Software Engineering
V00807959
tsalomon@uvic.ca

Introduction

Security has become a concern with the rise of communication systems. An RSA cryptosystem underlies modern security protocols on the Internet. RSA utilizes modular arithmetic as its core operation. Modular exponentiation on large numbers is used to encrypt data. This is a slow process with repeated modular multiplications.

The efficiency of RSA crypto operations relies on the speed of modular multiplication. Montgomery Multiplication is an efficient method of modular multiplication. This project involves implementation of an RSA cryptosystem using Montgomery multiplication.

As security devices become smaller, they require embedded processors which have limited processing power. However, data encryption operations must remain efficient on embedded processors. Therefore, the RSA cryptosystem is optimized for a embedded processor. The optimized cryptosystem must complete data operations within 10 seconds. The cryptosystem is measured to determine the fulfilment of this requirement.

The cryptosystem is implemented in the C language and optimized for and embedded ARM processor. The author is the main contributor to the project.

Theoretical Background

A cursory understanding of RSA cryptography and modular arithmetic is necessary to understand design choices and optimizations of the cryptosystem. Also the relevant details of the specific embedded device are discussed.

RSA Cryptography

RSA cryptography involves encryption/decryption of data using a key. A larger key requires more repeated operations and provides higher security at the cost of performance. Keys are numbers usually between 512-2048 bit long. The crypto implementation supports up to 2048 bit keys.

RSA performs arithmetic using these large keys. However, computer processors only intrinsically support arithmetic with less than 64-bit numbers. Therefore, support for arithmetic with up to 2048 bit numbers (aka. big numbers) is required. The big number

arithmetic should use the existing 16/32/64 bit arithmetic operations already supported by the processor.

Modular Arithmetic

The main operations of RSA are modular exponentiation and repeated modular multiplications. An example of both is shown below:

- modular exponentiation (ME)
  e.g. 345^{235} mod 678
- modular multiplication (MM)
  e.g. (123 * 364) mod 678

These examples use very small numbers. These operations must support up to 2048-bit numbers.

The ARM Processor

The cryptosystem is optimized for a FriendlyArm Mini2440 development board with an ARM926T processor [1]. The ARM processor (ARM926T) Technical Manual[2] provides all processor-specific design details. The processor supports 16-bit and 32-bit ARM assembly instructions. 32-bit arithmetic with a carry flag is also supported.

Design

The major components of the RSA cryptosystem are modular exponentiation, modular multiplication, and a data structure for big numbers.

Modular Exponentiation

An algorithm for modular exponentiation - multiply and square - has been chosen for implementation. The algorithm performs a modular multiplication at each step to keep any intermediate results within the range of the modulus. This is desired since this reduces the memory required for computation.

Given the modulus rarely changes, for some applications it is beneficial to compute a lookup table offline for certain intermediate results. This will greatly improve the efficiency of the implementation, at the cost of the memory needed to store the table.

Lookup Table

The size of the lookup table is related to the RSA key size (key size + 1). Given big numbers are 260 bytes and the key size is 2048, the lookup table requires ~520 kilobytes.

The ARM processor has a 16KB data cache. This means the whole lookup table cannot fit in cache and thus cache misses may be encountered.

Montgomery Multiplication

MM and ME can be computed using multiplication and division operations. However, multiplication and division are expensive for computers to compute. An improved algorithm was developed by Professor Montgomery[3], named Montgomery Multiplication (MontM). The new algorithm mostly uses addition and bitwise shifts. These ops are relatively inexpensive. Thus this algorithm is suited to embedded systems with limited computational power.

The power of Montgomery's algorithm is Montgomery reduction (MontRed). MontRed convert operands into a modular representation where division and multiplication are equivalent to addition and bitwise shifts. This allows efficient computation of intermediate values. The final result must be converted back.

MontRed uses Montgomery multiplication and requires a pre computed value (R), related to the size of the modulus:

e.g $R = 2^{2m} \bmod M$, where m = #bits of in the modulus

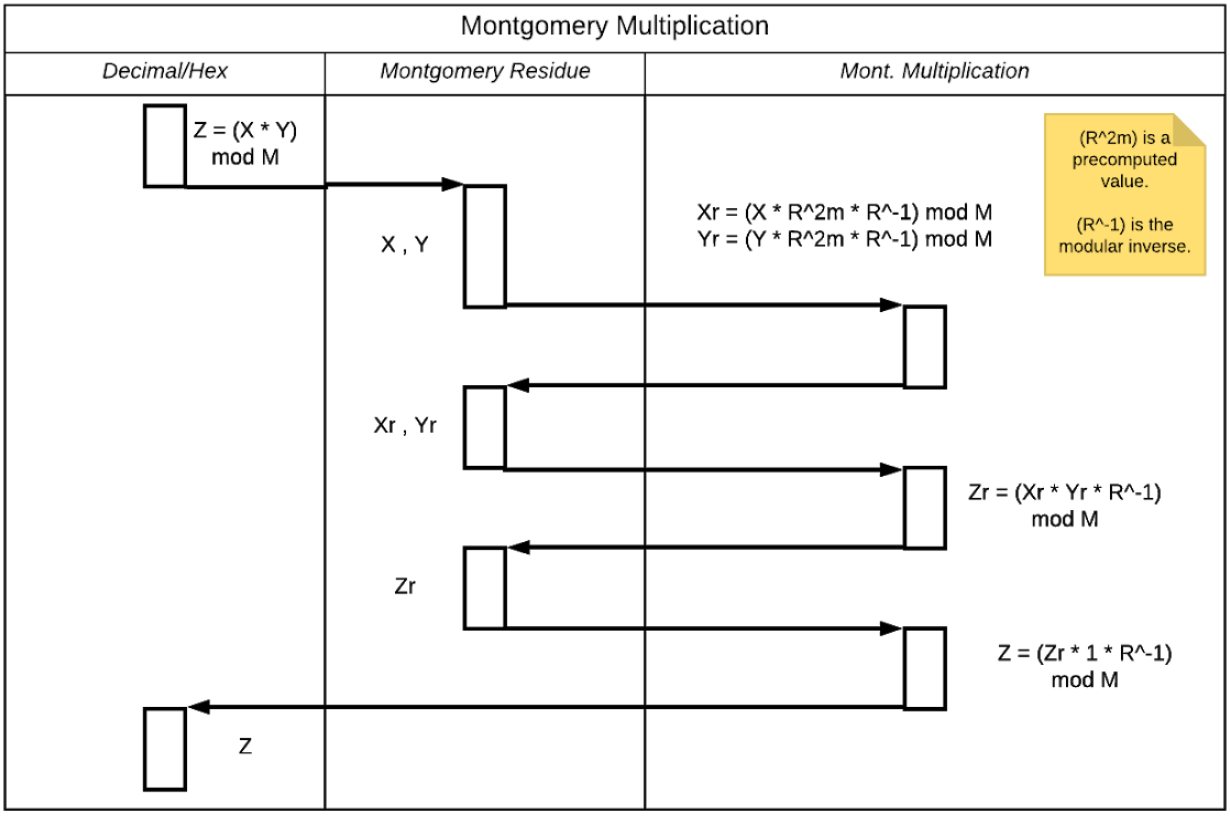

Figure 1: Montgomery multiplication process (swimlane diagram)

Big Numbers

A custom data structure is required to support arithmetic on up to 2048 bit numbers. The highest priority is an efficient implementation. Since RSA utilizes positive numbers only, signed representation is not necessary.

Optimal Memory Utilization

A naive data structure would represent each positional digit of a number as a 32-bit integer in an array. Given a n-bit number in base 10, this method requires a maximum of $\log_{10}(2^n)$ integers.

Size of number *n* [bits]	# integers	#bits
2048	log_10(2^2048) = 616.5 = 617	617*32 = **19744 bits**

The optimal data structure would represent an *n* bit number with n bits. This means the 2048 bit number can be represented by an array of 32 bit integers. An additional bit is required to handle arithmetic overflows.

Size of number *n* [bits]	# integers	#bits
2048 + 32	2080/32 = 65	65*32 = **2080 bits**

This is a 89% decrease in the memory footprint of a big number.

Bignum Library

This implementation uses a big number library written in C name *tiny-bignum-c*[4]. The library is modified and extended for use with RSA cryptography operations. By default, a few features suited to embedded systems are provided:

- Optimal bignum memory utilization
- Static memory usage and allocation
- Small code size: ~600 lines

The above features reduce the processor overhead and thus improve performance on embedded systems.

Code Organization

The program code and data is organized into 4 files:

- RSA_Montgomery.c — The RSA crypto implementation
- bn.c — Big number arithmetic implementation
- bn.h — Big number definitions (header file)
- LUT.txt — A lookup table for ME and testing

The RSA cryptosystem is implemented in *RSA_Montgomery.c*. This includes functions for:

- Modular exponentiation
  - With a lookup table — *modExp()*
  - Without a lookup table — *modExpLUT()*
- Modular multiplication — *montMult()*
- Reading the lookup table — *parseLUT()*

The bignum library consists of *bn.c* and *bn.h*. The C file contains necessary bignum arithmetic functions, such as addition and right shifts.

The C header file includes many definitions, including the word size of the processor (*WORD_SIZE*) and the number of integers in a big number data structure (*BN_ARRAY_SIZE*).

Performance and Cost Evaluation

The goal of this project is to optimize an RSA cryptosystem for an embedded ARM processor. The optimization effort should be directed toward the code which provides the most optimization benefits. The functions which require the most cpu time have the highest potential for optimization. Profile-driven compilation tools will help determine which functions will be targeted for optimization.

The GNU Compiler Collection (gcc) offers a profiling tool: *gprof*. The top five *gprof* results are shown in Table 1 below:

% Time	Total Calls	Function Name
59.15	8,370,160	bignum_add
39.34	8,245,748	bignum_rshift
0.55	-	_lshift_word
0.48	24,775,136	bignum_getbit
0.34	5390	montMult

Table 1: *gprof* time profiling results

The amount of cpu time a function requires only allows optimization within a function. Profiling also determines the caller of each function. This allows the possible reduction of function calls.

Parent caller information returned by *gprof* is displayed in Table 2:

Parent Function Name	Child Function	Total Calls
montMult	bignum_add	8237552
	bignum_rshift	8245748
	bignum_get_bit	24737244

Table 2: *grof* parent caller results

Table 2 shows the modular multiplier calls the most used functions. This is expected since its known that “the modular multiplier largely determine[s] the performance of modular exponentiation” [5].

Other functions within the big number implementation will be optimized. However, these functions have a lower priority for optimization since they will not drastically influence the efficiency.

Test Bench

The ARM926T processor does not have a cycle-accurate performance monitoring unit. This means objective and accurate cycle counts cannot be obtained. Therefore, data encryption/decryption completion time will be the best metric for evaluation.

Furthermore, the implementation will be tested with a range of key sizes: 12, 512, 1024, 2048 bits. Predefined keys were obtained using an online key generator[6].
Pre-computed values and the lookup table were computed offline with C code.

Optimizations

Knowledge of the embedded processor architecture and general optimization techniques are used to optimize the RSA cryptosystem.

Techniques Used

RSA crypto involves repeated operations within loops; i.e. modular multiplication. This means loop overhead has a high performance cost. Thus loop optimization will be the highest priority and should increase the performance. The software optimization techniques used are provided in Table 3.

Function Name	Loop Unrolling	Op Strength Reduction	Loop init and exit	Init to Zero
Bignum_add	Y	Y	Y	Y
Bignum_rshift	Y	Y	Y	Y
Bignum_getbit	Y	Y	N	N
Bignum_init	Y	N	Y	Y
Bignum_assign	Y	N	Y	Y
Bignum_numbits	N/A	Y	Y	N

Table 3: Optimization techniques used

The big number implementation features arbitrary precision numbers; a big number could contain 16-bit or 32-bit integers. Initially, arithmetic operations used the integer size for calculation.

The ARM processor has a 32-bit wordsize. Given this, many operations can be reduced in strength. For example, finding the number of bits in a word could require division by the word size. If the 32-bit word size is assumed, division is replaced by a more performant bitwise shift.

Bignum_add

Loop overhead and assertions utilize the most cpu time in this function. These require many comparison and branch instructions. Loop overhead is reduced with loop unrolling (n = 4). The assertions are removed and zero initialization is optimized

Unoptimized C code

```
void bignum_add(struct bn* a, struct bn*
b, struct bn* c){

  require(a, "a is null");
  require(b, "b is null");
  require(c, "c is null");

  DTYPE_TMP tmp;
  int carry = 0;
  int i;

  for (i = 0; i < BN_ARRAY_SIZE; ++i)
  {
    tmp = (DTYPE_TMP)a->array[i] +
b->array[i] + carry;
    carry = (tmp > MAX_VAL);
    c->array[i] = (tmp & MAX_VAL);
  }
}
```

Optimized C code

```
void bignum_add(struct bn* a, struct bn*
b, struct bn* c){

  DTYPE_TMP tmp;
  int carry = carry - carry;
  int i = 1;

  tmp = (DTYPE_TMP)a->array[0] +
b->array[0] + carry;
  carry = (tmp > MAX_VAL);
  c->array[0] = (tmp & MAX_VAL);

  for (i; i < (BN_ARRAY_SIZE); i+=4) {
    tmp = (DTYPE_TMP)a->array[i] +
b->array[i] + carry;
    carry = (tmp > MAX_VAL);
    c->array[i] = (tmp & MAX_VAL);

… 8 lines deleted….

    tmp = (DTYPE_TMP)a->array[i+3] +
b->array[i+3] + carry;
    carry = (tmp > MAX_VAL);
    c->array[i+3] = (tmp & MAX_VAL);
  }
}
```

Unoptimized Assembly (auto optimizations only):

The unoptimized assembly contains 5 functions. Function overhead includes many branch instructions. Branch instructions are expensive.

```
bignum_add:
        stmfd   sp!, {r4, r5, r6, r7, r8, lr}
        subs    r8, r0, #0
        mov     r7, r1
        mov     r0, r2
        beq     .L156
        cmp     r1, #0
        beq     .L157
        cmp     r2, #0
        movne r5, #0
        movne ip, r5
        beq     .L158
.L153:
        ldr     r3, [r7, ip]
        ldr     r1, [r8, ip]
        mov     r4, #0
        mov     r2, #0
        mov     r6, r5, asr #31
        adds    r3, r3, r1
        adc     r4, r4, r2
        adds    r3, r3, r5
        adc     r4, r4, r6
        str     r3, [r0, ip]
        add     ip, ip, #4
        subs    r5, r4, #0
        movne           r5, #1
        cmp     ip, #260
        bne     .L153
        ldmfd   sp!, {r4, r5, r6, r7, r8, lr}
        bx      lr
.L156:
        ldr     r0, .L159
        ldr     r1, .L159+4
        mov     r2, #210
        ldr     r3, .L159+8
        bl      __assert_fail
.L158:
        ldr     r0, .L159+12
        ldr     r1, .L159+4
        mov     r2, #212
        ldr     r3, .L159+8
        bl      __assert_fail
.L157:
        ldr     r0, .L159+16
        ldr     r1, .L159+4
        mov     r2, #211
        ldr     r3, .L159+8
        bl      __assert_fail
```

Optimized Assembly (manual and auto optimizations):

The optimized assembly has no expensive branches. The addition of a loop prologue and loop unrolling in the C code caused the compiler to remove the loop overhead completely. Now the loop body instructions repeat in a single assembly function.

```
bignum_add:
        ldr     r3, [r0, #0]
        stmfd   sp!, {r4, r5, r6, r7, r8, r9, sl}
        ldr     r9, [r1, #0]
        mov     r4, #0
        mov     sl, #0
        adds    r9, r9, r3
        str     r9, [r2, #0]
        ldr     r3, [r0, #4]
        ldr     r5, [r1, #4]
        adc     sl, sl, r4
        mov     r6, #0
        subs    r7, sl, #0
```

```
... deleted many repeating lines…

        movne   r0, #1
        mov     r4, #0
        mov     r6, #0
        adds    r3, r3, r5
        mov     r1, #0
        adc     r4, r4, r6
        adds    r3, r3, r0
        adc     r4, r4, r1
        str     r3, [r2, #256]
        ldmfd   sp!, {r4, r5, r6, r7, r8, r9, sl}
        bx      lr
```

Optimization Results

These metrics represent the exact benefit of the add optimizations. The 2nd column In Table 4 lists the cpu time when ONLY the add function is unoptimized.

	Optimized CPU Time [seconds]		**% Manual Optimization**
Key Size	***Auto***	***Auto & Manual***	
12	0.00	0.00	N/A
512	2.42	2.09	13.6%
1024	9.58	8.28	13.6%
2048	38.49	33.25	13.6%

Table 4: CPU time saved with add optimizations

Bignum_rshift

The right bitwise shift function uses expensive division and multiplication operations. Therefore, operator strength reduction will have the best optimization benefit. Zero initialization is also optimized. The helper function, *_rshift_word,* uses loop unrolling to reduce loop branch and compare overhead.

Unoptimized C code

```
void bignum_rshift(struct bn* a, struct bn* b, int nbits){
  require(a, "a is null");
  require(b, "b is null");
  require(nbits >= 0, "no negative shifts");

  /* Handle shift in multiples of word-size */
  const int nbits_pr_word = (WORD_SIZE * 8);
  int nwords = nbits / nbits_pr_word;
  if (nwords != 0){
    _rshift_word(a, nwords);
    nbits -= (nwords * nbits_pr_word);
  }

  if (nbits != 0){
    int i;
    for (i = 0; i < (BN_ARRAY_SIZE - 1); ++i){
      a->array[i] = (a->array[i] >> nbits) | (a->array[i + 1] << ((8 * WORD_SIZE) - nbits));
    }
    a->array[i] >>= nbits;
  }
  bignum_assign(b, a);
}
```

```
static void _rshift_word(struct bn* a, int nwords){
  require(a, "a is null");
  require(nwords >= 0, "no negative shifts");

  int i;
  for (i = 0; i < nwords; ++i){
    a->array[i] = a->array[i + 1];
  }
  for (; i < BN_ARRAY_SIZE; ++i){
    a->array[i] = 0;
  }
```

Additionally, a constant value is recalculated within a loop: *((8 * WORD_SIZE) - nbits)*. Instead, this value can be calculated once and used within in the loop.

Optimized C Code

```
void bignum_rshift(struct bn* a, struct
bn* b, int nbits){
  /* Handle shift in multiples of word-size
*/
  int nwords = nbits >> 5;
  if (nwords != 0){
    int z = nwords << 5;
    _rshift_word(a, nwords);
    nbits -= (z);
  }

  if (nbits != 0){
    int z = 32 - nbits;
    int i = i - i;
    for (; i < (BN_ARRAY_SIZE - 1); i++)
    {
      a->array[i] = (a->array[i] >> nbits) |
(a->array[i + 1] << (z));
    }
    a->array[i] >>= nbits;
  }
}
```

```
static void _rshift_word(struct bn* a, int
nwords)
{
  int i = i-i;
  for (; i < nwords; i+=4)
  {
    a->array[i]   = a->array[i + 1];
    a->array[i+1] = a->array[i + 2];
    a->array[i+2] = a->array[i + 3];
    a->array[i+3] = a->array[i + 4];
  }
  register int z = z-z;
  for (; i < BN_ARRAY_SIZE; i+=4)
  {
    a->array[i]   = z;
    a->array[i+1] = z;
    a->array[i+2] = z;
    a->array[i+3] = z;
  }
```

Optimization Results

	Optimized CPU Time [seconds]		**% Manually Optimized**
Key Size	***Auto***	***Auto & Manual***	
12	0.00	0.00	N/A
512	2.42	1.91	21.1%
1024	9.57	7.55	21.1%
2048	38.48	30.36	21.1%

Table 5: CPU time saved with right shift optimizations

Bignum_getbit

This function returns the zero-indexed bits of big number. This function is used within the modular multiplier. This method will be optimized with operator strength reduction. The expensive division and modulus operations can be replaced by bitwise shifts, subtraction and logical AND.

Unoptimized C Code

```
int bignum_getbit(struct bn* a, int n){
        int arrayInd = (n / 32);
        int numShift = (n  %  32);
        uint32_t num = a->array[arrayInd];
        uint32_t numChanged = (num >>
numShift);
        return numChanged % 2;
}
```

Optimized C Code

```
int bignum_getbit(struct bn* a, int n){
        int arrayInd = (n >> 5);
        int shift = (n - (arrayInd << 5));
        return (a->array[arrayInd] >>shift)&1;
}
```

Use of variables smaller than the word size of the processor (e.g unsigned short int) would not increase efficiency since ARM must clear/extend the unused portion of a word. In fact, loading a smaller variable may take 1 additional cycle to store in a register[7].

Unoptimized/Optimized Assembly Code

Using automatic compiler optimizations, the assembly code reduces from 32 to 13 instructions:

```
bignum_getbit:
        cmp     r1, #0
        mov     r2, r1, asr #31
        add     r3, r1, #31
        mov     r2, r2, lsr #27
        movge   r3, r1
        mov     r3, r3, asr #5
        add     r1, r1, r2
        ldr     r0, [r0, r3, asl #2]
        and     r1, r1, #31
        rsb     r1, r2, r1
        mov     r0, r0, lsr r1
        and     r0, r0, #1
        bx      lr
```

With the manual optimizations outlined above, the assembly code is further reduced to 6 instructions.

```
bignum_getbit:
    mov     r3, r1, asr #5
    ldr     r0, [r0, r3, asl #2]
    and     r1, r1, #31
    mov     r0, r0, lsr r1
    and     r0, r0, #1
    bx      lr.
```

Code Optimization Results

The results of all code optimizations are presented in the Table 6 below:

	Optimized CPU Time [seconds]					**% Manual Optimization**
Key Size	***None***	***Add & Rshift only***	***Auto***	***Add & Rshift & Auto***	***All Opt. & Auto***	
12	0.01	0.00	0.00	0.00	0.00	N/A
512	7.40	5.77	2.42	1.58	1.56	35.54%
1024	29.15	22.70	9.58	6.26	6.17	35.59%
2048	116.31	90.07	38.49	25.13	24.81	35.54%

Table 6: code optimization results

This table allows comparison between the optimization benefit of:

1. Automatic compiler optimizations: *Auto*
2. Big number addition optimizations: *Add*
3. Big number right shift optimization: *Rshift*

The results show that manual optimizations improved efficiency by almost 36%.

Lookup Table Optimization Results

The lookup table (*LUT.txt*) contains pre-computed modular multiplication results. Use of the table means modular exponentiation requires less CPU time and is therefore more efficient.

	CPU Time [seconds]			% LUT Optimization
Key Size	***Auto***	***Auto & LUT***	***All Opt., Auto, & LUT***	
12	0.00	0.00	0.00	N/A
512	2.42	0.82	0.53	66.1%
1024	9.58	3.18	2.05	66.8%
2048	38.49	12.81	8.27	66.7%

Table 7: Lookup table optimization results

Table 7 shows use of the lookup table increases efficiency by 66%.

Conclusions

In conclusion, RSA crypto operations must remain performant on embedded processors. For usability in many different applications, data operations must complete in under 10 seconds on an embedded process. The performance is measured on a ARM926T processor to determine whether this requirement is met.

	CPU Time [seconds]		% Total Optimization
Key Size	***Auto Opt.***	***All Opt.***	
12	0.00	0.00	N/A
512	2.42	0.53	78.1%
1024	9.58	2.05	78.6%
2048	38.49	8.27	78.5%

Table 8: Total optimization

Table 8 above shows the efficiency increase from code optimization and use of a lookup table. The last row shows data operations with a 2048-bit key require 8.27 seconds. Therefore, the RSA cryptosystem meets and exceeds the performance target of 10 seconds.

Bibliography

[1] "Mini2440 | S3C2440 ARM9 Board - FriendlyARM", *Friendlyarm.net*, 2018. [Online]. Available: http://www.friendlyarm.net/products/mini2440. [Accessed: 19- Jul- 2018].

[2] *ARM920T Technical Reference Manual*, 1st ed. ARM Limited, 2001.

[3] "Montgomery Modular Multiplication", *en.wikipedia.org*, 2018. [Online]. Available: https://en.wikipedia.org/wiki/Montgomery_modular_multiplication. [Accessed: 19- Jul- 2018].

[4] *tiny-bignum-c*. https://github.com/kokke/tiny-bignum-c/, 2018.

[5] M. Sima, *Lesson 103: RSA Cryptography*. Victoria, 2018, p. 14.

[6] R. Lie, "Online RSA key generation", *Mobilefish.com*, 2018. [Online]. Available: https://www.mobilefish.com/services/rsa_key_generation/rsa_key_generation.php. [Accessed: 19- Jul- 2018].

[7] *ARM920T Technical Reference Manual*, 1st ed. ARM Limited, 2018, pp. 12-5.