University of Victoria
SENG 440 Embedded Systems

RSA Cryptography Based On Montgomery Multiplication

August 06, 2024

Harsh Patel, V00946008, harshpatel@uvic.ca
Poomrapee Chuthamsatid, V00942601, pchuthamsatid@uvic.ca

Table of Contents

Outline

1. **Background**
    a. Introduction
    b. VM and Tools
    c. Difficulties
2. **Design**
    a. Flow Chart
    b. Algorithms
    c. Optimization
3. **Performance**
    a. Execution Time
    b. Number of Instructions
    c. Results
4. **Conclusion**
    a. Winner

Background

Introduction to RSA

RSA Cryptography is a technique to securely transmit data over wireless channels through the encryption and decryption of plaintext integers.

Public key-pair (E, PQ)

- Encrypt data
- Can be shared
- E is public exponent
- $C = T^E \bmod PQ$

Private key-pair (D, PQ)

- Decrypt data
- Must not be shared
- D is secret exponent
- $T = C^D \bmod PQ$

VM and Tools

Virtual Machine

- QEMU
- Fedora
- ARMv7

Tools

- GMP
- Sys/time
- Valgrind
- Assert

«Arithmetic without limitations»

Difficulties

Algorithms

- MMM
- ME
- Random Primes
- Extended GCD

Optimizations

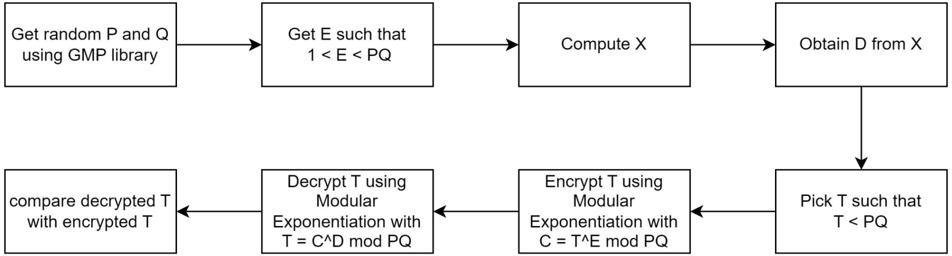
- Software Pipelining
- Loop Unrolling
- Predicate Operations
- Operator Strength Reduction
- Lookup Table

Data Type

- Short $\rightarrow$ P & Q
- Int $\rightarrow$ PQ, T
- Long Long $\rightarrow$ R^2,T^E
- Unsigned

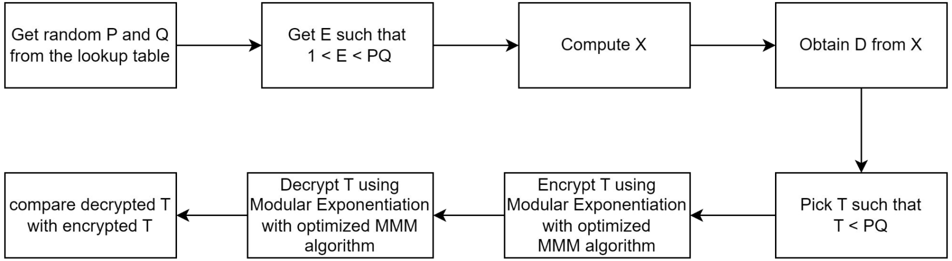
Design

Flow Chart



Brute Force

Optimized



Modular Exponentiation

Algorithms

- Replaces square with multiplication
- Accumulate multiplication control by mod m
- Replace $C = T^E \bmod PQ$ and $T = C^D \bmod PQ$
- But ...

```
uint64_t modular_exponentiation(uint64_t p, u
    uint64_t z = 1;
    p = p % m; //to ensure that p does not be
    while(e > 0){
        if ((e & 1) == 1){
            z = (z * p) % m;
        }

        e = e >> 1;
        p = (p * p) % m;
    }
    return (int32_t)z;
}
```

Montgomery Modular Multiplication

Algorithms

- Replaces multiplication with Add and Predicate
- Perform multiplication X and Y mod m
- Give X * Y * R^{-1} mod m
- Replace Z = Z * P mod m
- But need to pre-scale up to remove R^{-1} / add R

```
// p' = p * r*r * r^-1 mod m = p * r mod m
uint64_t p_prime = montgomery_modular_multiplication( x: p,  y: r2, m);
// z' = z * r*r * r^-1 mod m = z * r mod m
uint64_t z_prime = montgomery_modular_multiplication( x: z,  y: r2, m);
// z = z' * p' * r^-1 mod m = z * p * r mod m
z = montgomery_modular_multiplication( x: z_prime,  y: p_prime, m);
// z = z * p * r * r^-1 mod m = z * p mod m
z = montgomery_modular_multiplication( x: z,  y: 1, m);
```

```
uint64_t montgomery_modular_multiplication(uint64_t x, uint
    uint64_t m = M;
    uint64_t t = 0;
    uint64_t n;

    // loop through the number of m bits in pq
    while(m > 0){
        // n = T(0) XOR (X(i) AND Y(0))
        n = ((t & 1)) ^ ((x & 1) & (y & 1));

        // T = (T + X(i)Y + nM) >> 1
        t = (t + ((x & 1) * y) + (n * M)) >> 1;

        // get next bit of X(i) by shifting x to the right
        x = x >> 1;
        m = m >> 1;
    }

    if (t >= M) {
        t = t - M;
    }
    return t;
}
```

Optimizations

```
// loop through the number of m bits in pq
while(m > 0){
    // n = T(0) XOR (X(i) AND Y(0))
    n = ((t & 1)) ^ ((x & 1) & (y & 1));

    // T = (T + X(i)Y + nM) >> 1
    t = (t + ((x & 1) * y) + (n * M)) >> 1;

    // get next bit of X(i) by shifting x to
    x = x >> 1;
    m = m >> 1;
}
```

Loop Unrolling

Operator Strength Reduction

Software Pipelining

```
while (m > 2) {
    // First iteration
    n = ((t & 1)) ^ ((x & 1) & y_and_1);
    // replace multiplications with pred
    x_and_1 = (x & 1);
    xy = -x_and_1 & y;
    nm = (-n & M);
    t = (t + xy + nm) >> 1;
    x = x >> 1;

    // Second iteration (unrolled)
    n = ((t & 1)) ^ ((x & 1) & y_and_1);
    // replace multiplications with pred
    x_and_1 = (x & 1);
    xy = -x_and_1 & y;
    nm = (-n & M);
    t = (t + xy + nm) >> 1;
    x = x >> 1;
```

Optimizations

```
uint64_t montgomery_modular_multiplication(uint64_t x, uint64_t y, uint64_t M);
uint64_t modular_exponentiation(uint64_t p, uint64_t e, uint64_t m);
int32_t compute_x(uint32_t phi, uint16_t e);
int mod_inverse(int e, int phi);
uint16_t get_16bit_prime(int seed);
void gcd_extended(int e, int phi, int *x, int *y);
```

Register and Restrict

```
uint64_t montgomery_modular_multiplication(register uint64_t x, regis
uint64_t modular_exponentiation(register uint64_t p, register uint64_
int32_t compute_x(uint32_t phi, uint16_t e);
int mod_inverse(int e, int phi);
uint16_t get_16bit_prime(int seed);
void gcd_extended(register int e, register int phi, int* restrict x,
```

```
uint16_t get_16bit_prime(int seed) {
    gmp_randstate_t state;
    mpz_t prime;

    // initializes the random state
    gmp_randinit_default(state);
    gmp_randseed_ui(state, seed);
    mpz_init(prime);

    // Generate a random number with t
    mpz_urandomb(prime, state, 15);
```

Lookup Table

```
// List of all primes
static const uint32_t primes_16bit[] = {
        33811, 33827, 33829, 33851, 33857, 3386
        34877, 34883, 34897, 34913, 34919, 3493
        35993, 35999, 36007, 36011, 36013, 3601
        36997, 37003, 37013, 37019, 37021, 3703
        38083, 38113, 38119, 38149, 38153, 3816
        39161, 39163, 39181, 39191, 39199, 3920
        40193, 40213, 40231, 40237, 40241, 4025
        41281, 41299, 41333, 41341, 41351, 4135
        42307, 42323, 42331, 42337, 42349, 4235
        43391, 43397, 43399, 43403, 43411, 4342
};
```

```
if (t >= M) {
    t = t - M;
}
```

Predicate Operations

```
// branch elimination
t -= (t >= M) * M;
```

Optimizations (Assembly)

```
modular_exponentiation:
        @ args = 8, pretend =
        @ frame_needed = 0, u
        push    {r4, r5, r6,
        mov     r8, r2
        mov     r9, r3
        sub     sp, sp, #52
        mov     r4, r0
        mov     r5, r1
        ldrd    r0, [sp, #88]
        strd    r8, [sp, #8]
```

Unoptimized

Optimization made to the C code resulted in use of less memory intensive operations, i.e., use of mov instead of strd and shifting ldrd ahead to parallely compute other operations

```
modular_exponentiation:
        @ args = 8, preten
        @ frame_needed =
        push    {r4, r5, r
        mov     r10, r2
        mov     fp, r3
        sub     sp, sp, #1
        ldrd    r6, [sp, #
        mov     r8, r0
        mov     r9, r1
        mov     r0, r6
```

Optimized

Performance

Execution Time

- Use sys/time library
- Measure starting and ending period
- Execute stress test suites
    - 10 plaintext
    - 3 sets of prime seeds
    - 100 repetition
    - Total 3000 execution!
- Exercise: Brute Force, Algo, Algo & Opti, Algo & Opti & Lookup

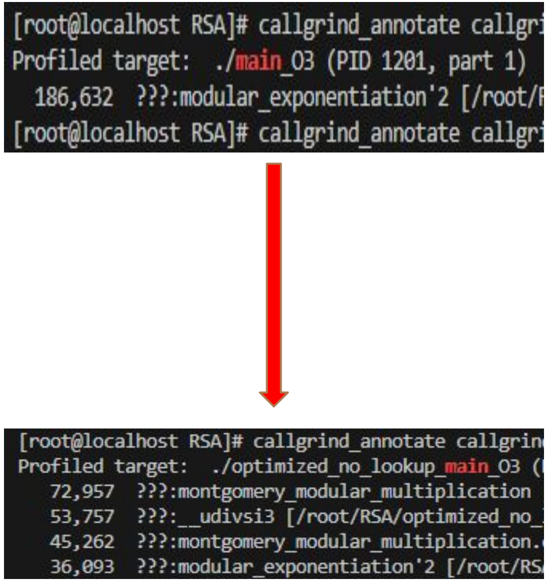
```
[root@localhost RSA]# ./main
./main
Total Time: 78239103 microseconds
[root@localhost RSA]# ./main
./main
Total Time: 70151298 microseconds
[root@localhost RSA]# ./main
./main
Total Time: 70392980 microseconds
```

```
[root@localhost RSA]# ./op
./optimized_main
Total Time: 65591549 microseconds
[root@localhost RSA]# ./op
./optimized_main
Total Time: 63705621 microseconds
[root@localhost RSA]# ./op
./optimized_main
Total Time: 67728714 microseconds
[root@localhost RSA]#
```

```
[root@localhost RSA]# ./optimized_main
./optimized_main
Total Time: 1190886 microseconds
[root@localhost RSA]# ./optimized_main
./optimized_main
Total Time: 1171913 microseconds
[root@localhost RSA]# ./optimized_main
./optimized_main
Total Time: 1182135 microseconds
```

Number of Instructions

- Apply optimizations techniques
- Compile the .c file with GCC -O3 flag
    - optimize during compilation
    - move operands to registers
    - eliminate iteration variables
    - integrate simple functions into callers
- Reduction of 150,539 instructions



Results

$$percent\ improvement = \frac{previous\ time - current\ time}{previous\ time} \times 100$$

Optimization Stage	Line of Instruction	Improvement %	Average Runtime (µSec)	Improvement %
1. Brute Force	Failed	Failed	Failed	Failed
2. Algorithms	186632	-	72927794	-
3. Algorithms & Optimization	36093	80.6%	65675295	+10.0%
4. Algorithms & Optimization & Lookup Table	47621	-24.2%	1181644	+98.2%

Conclusion

Optimized version

- Reduced number of instructions by 80%
- 10% better execution time
- Simulate real world

Optimized version with lookup table

- Reduced number of instructions by 74%
- 98% faster execution time
- Lookup table involves more complex logic than our implementation

Questions