University of Victoria
SENG 440 Embedded Systems

Final Report

RSA Cryptography Based On Montgomery Multiplication

Authors:
Poomrapee Chuthamsatid, V00942601, pchuthamsatid@uvic.ca, Software Engineering (SENG)

Harsh Patel, V00946008, harshpatel@uvic.ca, Software Engineering (SENG)

August 5th, 2024

Abstract

This report will aim to provide insights on the comparison of different techniques of implementing encryption and decryption for integers. For the purpose of this report, a brute force approach, Modular Exponentiation with Montgomery Modular Multiplication (MMM) algorithm, and optimized version of Modular Exponentiation with MMM algorithm will be discussed. In RSA Cryptography, the problem with encryption and decryption lies in the way of handling large integers. In this report, differences between different techniques of implementing RSA Cryptography will be highlighted along with their advantages and limitations. This report will allow the potential user to decide which approach to utilize for encrypting and decrypting large integers based on their project requirements. In the optimized version of the Modular Exponentiation with MMM algorithm, several optimization techniques were implemented such as operator strength reduction, loop unrolling, software pipelining, use of predicate operations, register and restrict keywords, branch elimination, and lookup tables. In this report, the performance of different techniques will be evaluated on the basis of runtime and number of instructions and such data will be presented numerically to understand them better. Moreover, flow charts will be provided to assist in visualizing the implementation logic of different algorithms. Moreover, this experiment was conducted on a QEMU virtual machine with default ARMv7 processor along with the GMP and time library to generate random 16 bit prime numbers and measure runtime respectively. Similarly, Valgrind was utilized to determine the number of instructions in the code. Furthermore, assembly will be evaluated to investigate the performance challenges of different algorithms. After the evaluation and performance testing of different algorithms, the Modular Exponentiation with MMM algorithm without the lookup is the most effective in terms of reducing the number of instructions. While the Modular Exponentiation with MMM algorithm with the lookup is the most efficient in terms of runtime.

Table of Contents

Introduction

In this report, the implementation of RSA Cryptography based on Montgomery Multiplication will be discussed along with its advantages and limitations in comparison to the brute force approach of RSA Cryptography. In addition, the optimizations made to the Modular Exponentiation with the Montgomery Modular Multiplication (MMM) algorithm of RSA will be presented. RSA is an acronym developed from the name of its inventors, **R**ivest, **S**hamir, and **A**dleman. In general, RSA cryptography is a technique used to securely transmit data over wireless channels through the encryption of plaintext integers. To encrypt and decrypt data, the RSA algorithm utilizes two distinct odd prime numbers P and Q. These two prime numbers are used to generate public key-pair $(E,\ PQ)$ to encrypt data and private key-pair $(D,\ PQ)$ to decrypt data. For the purpose of this report, P and Q are 3 to 16 bits odd prime integers.

Similarly, E is a 16 bit prime number since E must be greater than 1 and less than the $P\ *\ Q$. In addition, E and $(P\ -\ 1)(Q\ -\ 1)$ are relatively prime, meaning both integers do not have any prime factors in common. Moreover, it is assumed that E is a prime number to make the random generation E simple and focus more on the actual implementation logic of the algorithm rather than its input parameters. Furthermore, M is a maximum of 32 bit integers since it is equivalent to $P\ *\ Q$ where P and Q are a maximum of 16 bit integers. Additionally, D is a 64 bit integer derived from $D\ =\ (X(P\ -1)(Q\ -1)\ +\ 1)/E$, where X is an integer. Similarly, integer X is derived from $X\ =\ (kE\ -\ 1)\ /\ (P\ -\ 1)(Q\ -\ 1)$, where k is a modulo inverse of $(E,\ (P\ -\ 1)(Q\ -\ 1))$.

In RSA, the encryption and decryption functions are written as $C\ =\ T^E\ mod\ PQ$ and $T\ =\ C^D\ mod\ PQ$ respectively, where T and C are 31 and 64 bits positive integers known as ciphertext and plaintext respectively. In addition, T must be less than the modulus, PQ, so T is a maximum of 31 bits integers since if P and Q are the smallest 16 bits integers, i.e., 32771, then PQ is a 31 bit integer. Furthermore, E is a public exponent so it can be revealed to the public however D must never be revealed to anybody since it is a secret exponent. The main difference between the brute force approach and MMM algorithm for RSA Cryptography lies in the computation of encryption and decryption functions. Imagine, C is a uint64_t integer, $T\ =\ 18455884$ and $E\ =\ 13$, then T^E will become a 314 bit integer, resulting in an error since the scope of the program is at maximum uint64_t. However, if Modular Exponentiation technique is implemented using MMM algorithm, T^E can be computed successfully since modulo function is performed at each step of the algorithm to keep any intermediate variables within the integer range PQ.

In addition, assume modular exponentiation, $X * Y \, mod \, M$ needs to be calculated using MMM algorithm where M is PQ. Let $R = 2^m$, where m is the bit-length of M. The MMM algorithm calculates $Z = X * Y * R^{-1} \, mod \, M$, so somehow that R^{-1} needs to be removed to obtain $X * Y \, mod \, M$. Therefore, both X and Y needs to be pre-scaled up with scale factor R^2, so that
$\overline{X} = X * R^2 * R^{-1} \, mod \, M = X * R \, mod \, M$ and
$\overline{Y} = Y * R^2 * R^{-1} \, mod \, M = Y * R \, mod \, M$. Then,
$\overline{Z} = (X * R) * (Y * R) * R^{-1} \, mod \, M = X * Y * R \, mod \, M$. However, there still exists a scale factor R, but it can be removed by performing MMM algorithm again without passing any scale factor since MMM algorithm inherently multiples by R^{-1}, so $Z = X * Y * R * R^{-1} \, mod \, M$, and from this the final result $Z = X * Y \, mod \, M$ can be achieved.

In terms of team contributions, both team members, Harsh and Poom, worked together at all times, including setting up the virtual machine, writing a progress report, coding the project, asking questions about the project to the professor, Dr. Mihai Sima, creating presentation slides and writing this final report. However, the breakdown of which team member worked on a specific topic the most can be observed from table 1.

Table 1: Breakdown of team contributions

Task	Team Member
Set up QEMU Virtual Machine	Poom
Montgomery Modular Multiplication Algorithm Implementation	Harsh
Modular Exponentiation Implementation	Poom
Generate Random Prime Numbers	Both
Mod Inverse	Both
GCD Extended	Both
Compute X	Both
Code Optimizations	Both
Write Performance Tests	Poom
Setup Valgrind to determine number of instructions in a function	Harsh

Analyze Assembly	Harsh
Presentation Slides	Both
Progress Report	Both
Final Report	Both

This report aims to provide details regarding the setup of the QEMU virtual machine with ARMv7 processor in the background section. Furthermore, implementation logic, assembly, and flow charts of the brute force approach and MMM algorithm to the RSA Cryptography will be presented in the design section. In addition, there will be three .c files, main.c, optimized_main.c, and optimized_no_lookup_main.c, in the submission both containing the code of MMM algorithm but the optimized_main.c and optimized_no_lookup_main.c file will contain the optimized version of the code of main.c. The assembly of main.c, optimized_main.c, and optimized_no_lookup_main.c will be evaluated to measure the performance of both files numerically in terms of runtime and number of instructions. Ultimately, the advantages and limitations of implemented logic will be presented along with the future work that needs to be done on this project. As well, a conclusion will be made based on the presented work with proper justifications.

Background

This section will discuss our developing environment, selection of libraries, and dependencies. We will define their usage and explain the rationale behind each decision.

Virtual Machine

1. **QEMU**

   QEMU is a tool that emulates a development environment and enables software to be compiled for one architecture on a different architecture.

   Our team used QEMU images, obtained from the course website [2] to create and run a virtual machine (VM). This VM provides necessary development environment dependencies and isolates from our host system, ensuring reliability and consistency in our team development.

```
$ qemu-system-arm -m 1G -smp 1 -hda Fedora-Minimal-armhfp-29-1.2-sda.qcow2 -mach
ine virt-2.11 -kernel vmlinuz-4.18.16-300.fc29.armv7hl -initrd initramfs-4.18.16
-300.fc29.armv7hl.img -append "console=ttyAMA0 rw root=LABEL=_/ rootwait ipv6.di
sable=1" -nographic -netdev user,id=seng440,hostfwd=tcp::2222-:22 -device virtio
-net-pci,netdev=seng440
[    0.000000] Booting Linux on physical CPU 0x0
[    0.000000] Linux version 4.18.16-300.fc29.armv7hl (mockbuild@buildvm-armv7-06.arm.fedoraproject.org) (gcc version 8.2.1 20180801 (Red Hat 8.2.1-2) (GCC)) #1 SMP Sun Oct 21 00:56:28 UTC 2018
[    0.000000] CPU: ARMv7 Processor [414fc0f0] revision 0 (ARMv7), cr=10c5387d
[    0.000000] CPU: div instructions available: patching division code
[    0.000000] CPU: PIPT / VIPT nonaliasing data cache, PIPT instruction cache
[    0.000000] OF: fdt: Machine model: linux,dummy-virt
```

Figure 1: QEMU Virtual Machine [2]

Operating System And Processor

1. Fedora

Fedora is the default operating system from the QEMU image.

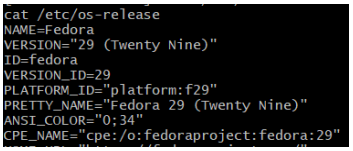

Figure 2: Fedora Operating system

2. ARMv7 Processor

Arm7 is a default processor from the QEMU image.

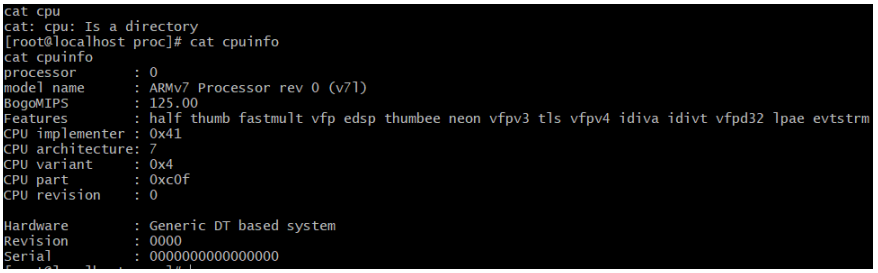

Figure 3: ARMv7 Processor

Libraries

1. NEON

NEON provides SIMD to boost performance for applications that can leverage parallel processing, reducing the number of instructions and cycles performed on computations [6].

NEON is a default library, given to us in the QEMU image downloaded from the course page. However, we do not incorporate the NEON library in our RSA application, since all of our computations are sequentially dependent and heavily rely on the results of the previous computation. Thus, parallelization would not be straightforward and could hinder the exploitation of the NEON library.

2. GNU Multiple Precision Arithmetic Library (GMP)

GMP is a library that gives us the ability to operate with arbitrary precision integers, rationales, and floating-point numbers. It simplifies the process of computations with a large number [3].

GMP is an external library that our team downloaded to accommodate our RSA processing. We employ the library to precisely generate a large (or

small) random prime number to compute p, q, and e values, used to process data cryptography. We incorporate GMP instead of coding entirely new functions to ensure the performance and reliability of arbitrary precision integer computations.

3. **sys/time**

   sys/time is the C standard library that provides functions and data structures for manipulating time and dates [7].

   sys/time is a standard library provided by Unix-like operating systems, including our Fedora operating system. We employ the library to measure the runtime between the start and the end of our application execution process.

```
gcc -mfpu=neon -march=armv7-a -mtune=cortex-a9 main.c -lgmp -o main -lm
./main
```

Figure 4: the commands to compile and run our main.c with all dependency flags

4. **Valgrind**

   Valgrind is a CPU profiling tool and callgrind is a tool incorporated into Valgrind. The callgrind is a call-graph generating cache and branch prediction profiler. In this project, callgrind is used to determine the number of instructions in a function.

```
[root@localhost RSA]# sudo yum -y install valgrind
```

Figure 5: Install Valgrind in Fedora operating system [4]

```
[root@localhost RSA]# valgrind --tool=callgrind ./optimized_main
==974== Callgrind, a call-graph generating cache profiler
==974== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
```

Figure 6: generate callgrind.out.* file for the optimized_main executable

```
[root@localhost RSA]# callgrind_annotate callgrind.out.974 | grep main
Profiled target:  ./optimized_main (PID 974, part 1)
```

Figure 7: determine the number of instructions in optimized_main file

Design

C Code Structure

Initially, the brute force approach was utilized for the cryptography of large integers. However, we found that the brute force approach was inefficient in terms of both runtime and space complexity. Additionally, it produced errors when dealing with larger values of T and E in the operation T^E. Subsequently, we incorporated MMM algorithms to optimize the performance. We further applied optimization techniques, including loop unrolling, software pipelining, grafting, operator strength reduction, and predicate execution, to reduce the number of instructions. The implementation of such optimization techniques resulted in improvement in terms of number of instructions for the modular_exponentation function. Finally, we added a lookup table to quickly retrieve prime numbers for the values of P, Q, and E, resulting in a significant runtime improvement.

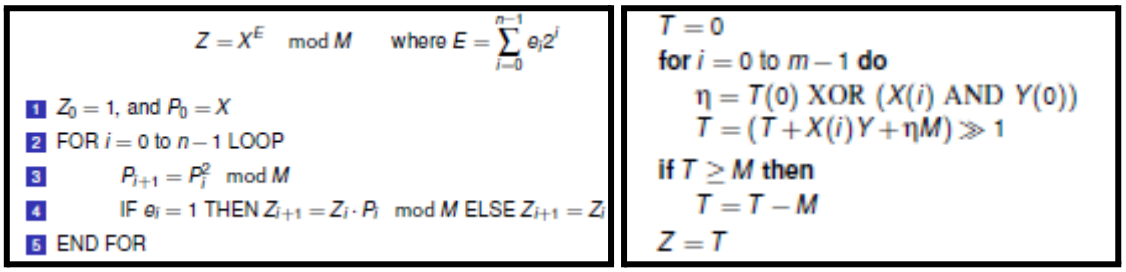

Figure 8: Pseudo code of modular exponentiation (left image) and MMM algorithm (right image)

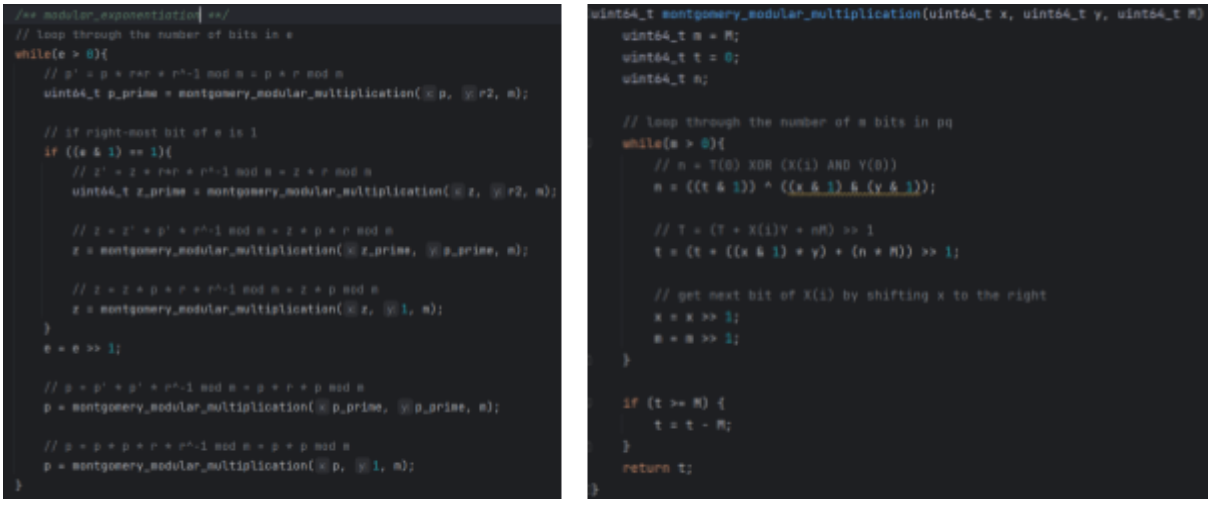

Figure 9: Implemented code of modular exponentiation (left image) and MMM algorithm (right image)

Even though pseudo code is using a for loop, we implemented the code using bit-wise while loop so that array is not required to retrieve Zi and Pi. The pseudo code of modular exponentiation, as shown in left image of figure 8, was implemented in main.c as shown in left image of figure 9. Later, that code was optimized in optimized_no_lookup_main.c and optimized_main.c and it can be found on line 119

to 159 and 113 to 153 respectively. Similarly, the pseudo code of modular exponentiation, as shown in the right image of figure 9, was implemented in main.c as shown in the right image of figure 9. Later, that code was optimized in optimized_no_lookup_main.c and optimized_main.c and it can be found on line 169 to 228 and 163 to 203 respectively. Moreover, the optimization version of the code is using predicate operations with 2's complement to eliminate multiplication operations of MMM algorithm. Furthermore, the branch was eliminated using predicate operation.

Assembly

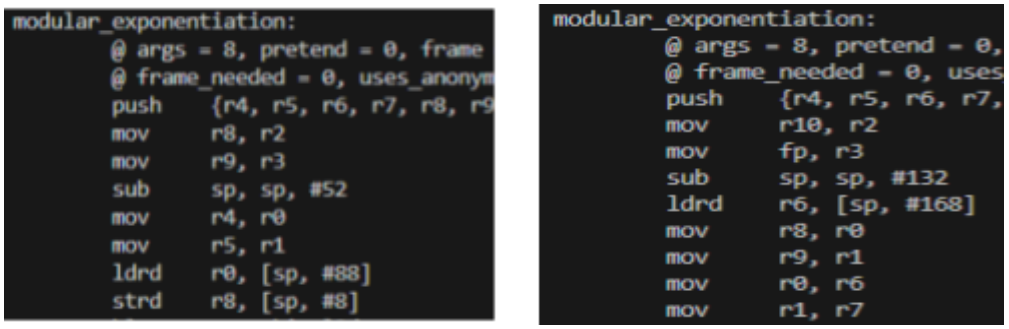

Figure 10: modular_exponentiation function of main.s (left image) and its optimized version (right image) with -O3 optimization flag

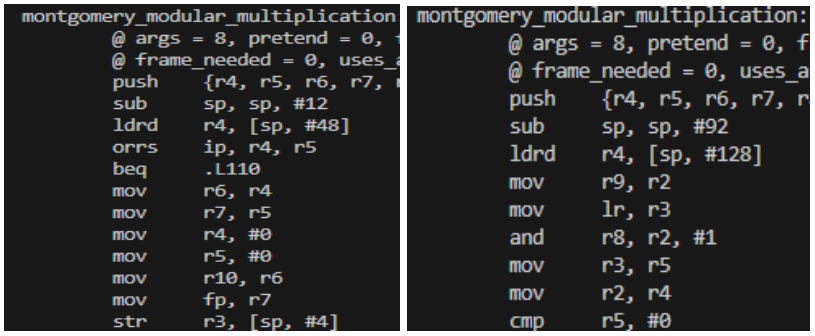

Figure 11: montgomery_modular_multiplication of main.s (left image) and its optimized version (right image) with -O3 optimization flag

For the purpose of this report, only a few lines of assembly are provided to show the difference of operations used for the modular_exponentiation (figure 10) and montgomery_modular_multiplication (figure 11) functions in main.s and optimized_no_lookup_main.s. From figure 10 and 11, it can be observed that the optimized_no_lookup_main.s (right image) is using less expensive operations than main.s (left image) to perform the same task as well as instructions are being reordered such that parallel computation is possible. These optimizations were possible because operator strength reduction, software pipelining, loop unrolling, and branch elimination was performed on these functions. It can be observed from figure 10 that the optimized version (right image) is using mov operation rather than strd being used on the main version (left image). Similarly, it can be observed in figure 11 that the optimized version (right image) is predicate operations whereas the main

version is using beq. Due to the implementation of several optimization techniques discussed in the Performance section of this report, the number of instructions were reduced by approximately 80.6 %, in the modular_exponentiation function, as shown in figure b.

Flow Charts

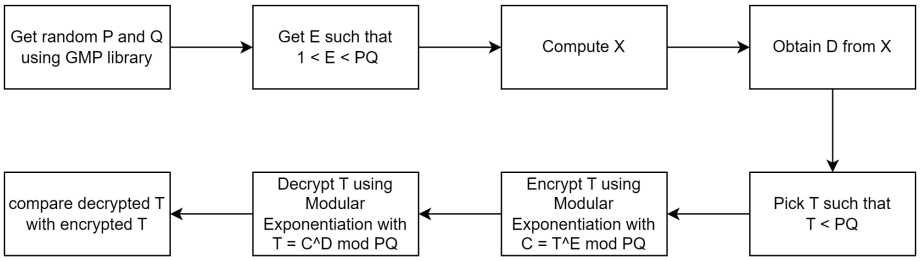

Figure 12: Encrypt and decrypt T using Brute Force Approach

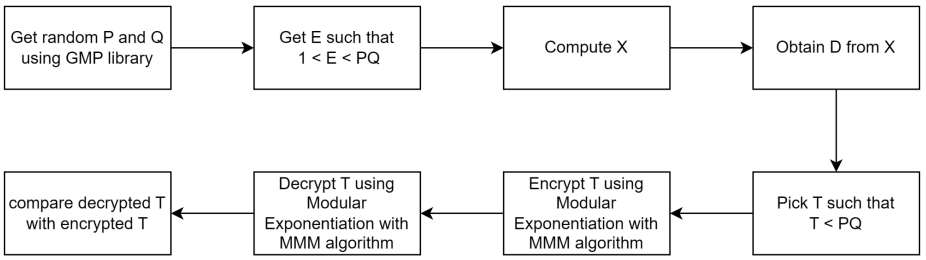

Figure 13: Encrypt and decrypt T using Modular Exponentiation with MMM algorithm

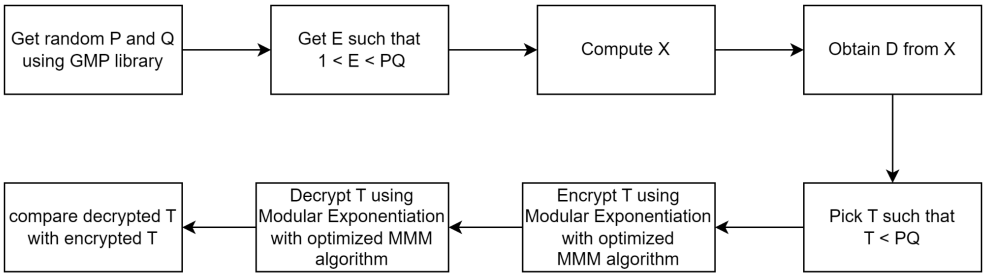

Figure 14: Encrypt and decrypt T using Modular Exponentiation with optimized MMM algorithm

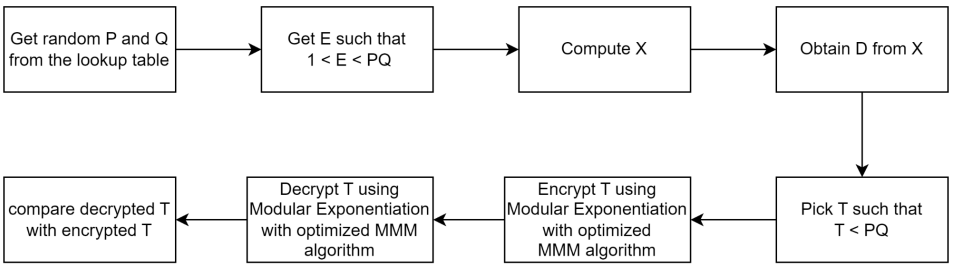

Figure 15: Encrypt and decrypt T using Modular Exponentiation with optimized MMM algorithm and lookup table

The algorithm flow for encryption and decryption of large integers using brute force, modular exponentiation with MMM algorithm, and modular exponentiation with optimized MMM is shown in Figures a, b, and c, respectively. The brute force approach has a very different flow than the other two approaches since it directly encrypts T using C = T^E mod M, meaning if T and E are large integers then the error will be thrown since enough space is not allocated to handle large integers.

On the other hand, the algorithm with MMM and optimized MMM are handling large integers by ensuring that intermediate integers are within the range of PQ. The main difference between the algorithm with MMM and optimized MMM is the reduction of expensive operations such as replacing multiplication with predicate operations with 2's complement as well as applying software pipelining and loop unrolling optimization techniques. This further optimization on MMM significantly improves the performance, since the function is recursively called during the cryptography process.

Finally, we experimented with a lookup table. The lookup table stores a set of 1,000 prime numbers, each 15 or 16 bits long. Although we could have stored all 15- and 16-bit prime numbers, we chose to limit the number to maintain the readability of our codebase. The lookup table reduces the prime number search from a non-constant to a constant runtime. This allows us to randomly retrieve prime numbers for p, q, and e with consistent time complexity.

Performance

RSA Cryptography Optimization is a project that aims to optimize the RSA encryption and decryption processes. The optimizations include the use of Montgomery Modular Multiplication and Modular Exponentiation algorithms to reduce the computational strength of multiplication and exponentiation operations. In addition to algorithms, this project applies different optimization techniques including loop unrolling, software pipelining, branch elimination, operator strength reduction, predicate operations, lookup table, restrict-qualified pointers and register keywords for variables to reduce the number of instructions and improve runtime.

Optimizations

main.c:
Initially, we improved the Brute Force RSA by implementing Modular Multiplication and Modular Exponentiation algorithms, which reduce the computational strength of multiplication and exponentiation operations. This upgrade improves the plaintext input capacity from 2^1 to 2^{29} size. This limitation arises because the minimum size of the product of two 15-bit prime numbers is smaller than the plaintext. As a result, we obtain the main.c file. (See Design Section)

optimized_no_lookup_main.c:
Furthermore, we applied various optimization techniques to the main.c, resulting in the optimized_no_lookup_main.c.

In Montgomery Modular Multiplication, loop unrolling was used to perform m bit-shifting three times at once. However, further optimization through parallel computing between iterations was not feasible due to the dependency between computations. By reordering instructions, software pipelining was performed to enable parallel computations of instructions within the 'while loop'. We performed operator strength reduction on 't = (t + ((x & 1) * y) + (n * M)) >> 1;' to remove multiplication instructions by using predicate operations along with 2's complement. Finally, A branch created by the 'if' statement was also eliminated, and replaced with a predicate operation (See Figure 16.)

```
// Loop through the number of m bits in pq
// loop unrolled thrice
// software pipelining by reordering instructions
while (m > 2) {
    // First iteration
    n = ((t & 1)) ^ ((x & 1) & y_and_1);
    // replace multiplications with predicate operations and 2's complement
    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 predicate operations and 2's complement
    x_and_1 = (x & 1);
    xy = -x_and_1 & y;
    nm = (-n & M);
    t = (t + xy + nm) >> 1;
    x = x >> 1;

    // third iteration (unrolled)
    n = ((t & 1)) ^ ((x & 1) & y_and_1);
    // replace multiplications with predicate operations and 2's complement
    x_and_1 = (x & 1);
    xy = -x_and_1 & y;
    nm = (-n & M);
    t = (t + xy + nm) >> 1;
    x = x >> 1;

    // improve by shifting 3 bits at once
    m = m >> 3;
}

// remaining iterations (1 to 2)
while (m > 0) {
    n = ((t & 1)) ^ ((x & 1) & y_and_1);
    // replace multiplications with predicate operation and 2's complement
    x_and_1 = (x & 1);
    xy = -x_and_1 & y;
    nm = (-n & M);
    t = (t + xy + nm) >> 1;
    x = x >> 1;

    m = m >> 1;
}

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

Figure 16: optimization in montgomery_modular_multiplication function

Variables were declared with the register keyword to speed up the retrieval of frequently used variables since Montgomery Modular Multiplication is called recursively. The ‘restrict’ keyword was used for pointers initialization to ensure they do not overlap (See Figure 17.)

```
uint64_t montgomery_modular_multiplication(register uint64_t x, register uint64_t y, register uint64_t M);
uint64_t modular_exponentiation(register uint64_t p, register uint64_t e, register 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 bits, int seed);
void gcd_extended(register int e, register int phi, int* restrict x, int* restrict y);
```

Figure 17: use of registers and restrict for variable initialization

Another ‘if’ statement branch operation was eliminated using predicate operations in the main function (See Figure 18.)

```
// branch elimination: minor performance improvements, but make the code less readable
(pq) < t && (printf("Our plain text t must be less than p * q\n"), exit(-1), 0);
```

Figure 18: elimination of branch operation from the main function

Since ‘y & 1’ is a constant value, it was precomputed for later use to save computations. Then, we added the ‘register’ keyword to speed up the retrieval of frequently used variables (See Figure 19.)

```
uint64_t montgomery_modular_multiplication(register uint64_t x, register uint64_t y, register uint64_t M) {
    register uint64_t m = M;
    register uint64_t t = 0;
    register uint64_t y_and_1 = y & 1; // precompute y & 1
    register uint64_t n, xy, nm, x_and_1;
```

Figure 19: precomputation of y & 1 and use of registers for variable initialization

We know that r is in the form of 2^x where x is the n bits. The code is rearranged in $r \times r = 2^{x+x}$ which can be computed by the left shifting of m_bits. Thus, this is an operator strength reduction to eliminate multiplication with bit-shifting (See Figure 20.)

```
// r = 2^m bits
uint64_t m_bits = log2(m) + 1;
uint64_t r = 1ULL << m_bits;

// operator strength reduction, replacing multiplication with bit-shifting
// r*r mod m to pre-scale values later
uint64_t r2 = (r << m_bits) % m;
```

Figure 20: operator strength reduction in modular_exponentiation function

The GMP library was employed to ensure the accuracy of a random 16-bit prime number generation.

```c
uint16_t get_16bit_prime(int bits, 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);
```

Figure 21: use of GMP library in get_16bit_prime function

Optimized_main.c:

We continued to develop our ‘optimized_no_lookup_main.c’ solution to further speed up the runtime, resulting in the new optimized solution, ‘optimized_main.c’. Our application faced a bottleneck in the process of selecting 16-bit prime numbers. To reduce the time complexity, we incorporated a lookup table to achieve a constant runtime for this process. We created a large table of 1,000 elements containing 15- and 16-bit prime numbers. We then used a seed to randomly select elements from this table to generate p, q, and e values. As a result, performance improved significantly by 98%. Although we could increase the table size beyond 1,000 elements, we limited it to a thousand for readability (See Figure 22).

```c
// List of all primes
static const uint32_t primes_16bit[] = {
        [0]: 33811, [1]: 33827, [2]: 33829, [3]: 33851, [4]: 33857, [5]: 33863, [6]: 33871, [7]: 33889, [8]: 33893, [9]: 33911, [10]: 33923, [11]: 33931, [12]:
        [101]: 34877, [102]: 34883, [103]: 34897, [104]: 34913, [105]: 34919, [106]: 34939, [107]: 34949, [108]: 34961, [109]: 34963, [110]: 34981, [111]: 35023
        [201]: 35993, [202]: 35999, [203]: 36007, [204]: 36011, [205]: 36013, [206]: 36017, [207]: 36037, [208]: 36061, [209]: 36067, [210]: 36073, [211]: 36083
        [301]: 36997, [302]: 37003, [303]: 37013, [304]: 37019, [305]: 37021, [306]: 37039, [307]: 37049, [308]: 37057, [309]: 37061, [310]: 37087, [311]: 37097
        [401]: 38083, [402]: 38113, [403]: 38119, [404]: 38149, [405]: 38153, [406]: 38167, [407]: 38177, [408]: 38183, [409]: 38189, [410]: 38197, [411]: 38201
        [501]: 39161, [502]: 39163, [503]: 39181, [504]: 39191, [505]: 39199, [506]: 39209, [507]: 39217, [508]: 39227, [509]: 39229, [510]: 39233, [511]: 39239
        [601]: 40193, [602]: 40213, [603]: 40231, [604]: 40237, [605]: 40241, [606]: 40253, [607]: 40277, [608]: 40283, [609]: 40289, [610]: 40343, [611]: 40351
        [701]: 41281, [702]: 41299, [703]: 41333, [704]: 41341, [705]: 41351, [706]: 41357, [707]: 41381, [708]: 41387, [709]: 41389, [710]: 41399, [711]: 41411
        [801]: 42307, [802]: 42323, [803]: 42331, [804]: 42337, [805]: 42349, [806]: 42359, [807]: 42373, [808]: 42379, [809]: 42391, [810]: 42397, [811]: 42403
        [901]: 43391, [902]: 43397, [903]: 43399, [904]: 43403, [905]: 43411, [906]: 43427, [907]: 43441, [908]: 43451, [909]: 43457, [910]: 43481, [911]: 43487,
};

#define NUM_16BIT_PRIMES (sizeof(primes_16bit) / sizeof(primes_16bit[0]))
```

```c
/**
 * Generates a random prime number with a specified number of bits using the GMP library
 * Parameters: int seed - the seed for the random number generator
 * Returns: uint16_t - the generated prime number
* */
uint16_t get_16bit_prime(int seed) {
    return primes_16bit[seed % NUM_16BIT_PRIMES];
}
```

Figure 22: The lookup table of size 1000 elements containing 15- and 16-bit prime numbers.

Number of Instructions

Figure 23 shows that the total number of instructions in stage 2 is 186632. Stage 2 contains the implementation of modular exponentiation using the MMM algorithm, but the code was not optimized. On the other hand, stage 3 was optimized using techniques discussed in the design section but it does not include a lookup table. After optimizations, as shown in Figure 24, it had a total of 36093 instructions, which is 150,539 instructions less than stage 2. In stage 4, we experimented with a lookup table to optimize performance, but it failed to reduce the number of instructions as

shown in figure 25, however the runtime was improved (See Average Runtime Section).

```
[root@localhost RSA]# callgrind_annotate callgrind.out.1201 | grep main
Profiled target:  ./main_O3 (PID 1201, part 1)
  186,632  ???:modular_exponentiation'2 [/root/RSA/main_O3]
[root@localhost RSA]# callgrind_annotate callgrind.out.1201 | grep main
```

Figure 23: compilation of main using O3 flag

```
[root@localhost RSA]# callgrind_annotate callgrind.out.1203 | grep main
Profiled target:  ./optimized_no_lookup_main_O3 (PID 1203, part 1)
   72,957  ???:montgomery_modular_multiplication [/root/RSA/optimized_no_lookup_main_O3]
   53,757  ???:__udivsi3 [/root/RSA/optimized_no_lookup_main_O3]
   45,262  ???:montgomery_modular_multiplication.constprop.1 [/root/RSA/optimized_no_lookup_main_O3]
   36,093  ???:modular_exponentiation'2 [/root/RSA/optimized_no_lookup_main_O3]
```

Figure 24: compilation of optimization with lookup using O3 flag

```
Profiled target:  ./optimized_main_O3 (PID 1202, part 1)
 75,900  ???:montgomery_modular_multiplication [/root/RSA/optimized_main_O3]
 53,260  ???:__udivsi3 [/root/RSA/optimized_main_O3]
 51,060  ???:montgomery_modular_multiplication.constprop.1 [/root/RSA/optimized_main_O3]
 47,621  ???:modular_exponentiation'2 [/root/RSA/optimized_main_O3]
```

Figure 25: compilation of optimization with lookup using O3 flag

Average Runtime

The average runtime performance tests were performed on 4 different stages of the improvements including Brute Force (no algorithms), Algorithms (With Montgomery Modular Multiplication and Modular Exponentiation), Algorithms & Optimization (With the Algorithms and the optimization techniques), and Algorithms & Optimization & Lookup Table (With the Algorithms, the optimization techniques, and the lookup table) stages. We ran a stress test suite that executes 3000 cryptography processes to measure the runtime. We repeated the same testing thrice for each stage and found their average runtime. Finally, we applied a percent improvement formula ($percent\ improvement\ = \frac{previous\ time - current\ time}{previous\ time} \times 100$) to compare the results. We discovered that the average runtime improves by 10% from stage 2 to stage 3 and 98.2% from stage 3 to stage 4 [See figure 26]

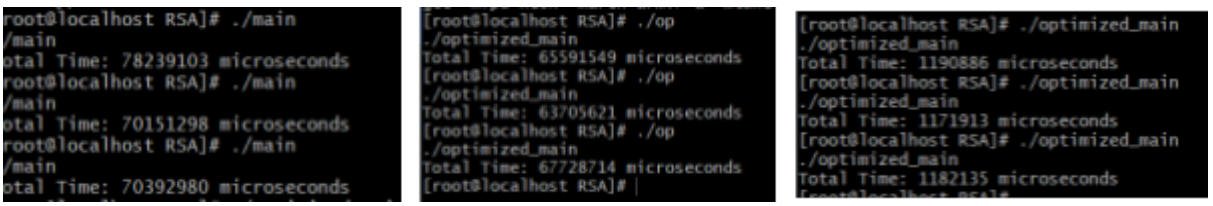

Figure 26: Runtime performance (microseconds) of stages 2, 3, and 4 respectively

Table 1: Show the performance measurement matrix by the Line of Instruction and the Average Runtime (µSec)

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%

Discussion

Achievements

In this project, various challenges were encountered such as optimizing memory intensive algorithms to handle large integers. This can be due to the lack of code optimization knowledge before this project started. There was a massive learning curve in implementing optimization techniques. For example, implementing software pipelining, loop unrolling, reducing operator strength, use of register and restrict for variable and pointer declaration, branch elimination, and utilizing predicate operations.

To make the learning simpler, the brute force approach was implemented initially but due to its limitation of handling T of only 2 bits, other algorithms needed to be implemented for encrypting and decrypting T. Later, unoptimized modular_exponentiation was implemented by following the slides provided by Dr. Mihai Sima. However, since several memory intensive operations were being executed, it needed to be improved further. For example, executing several multiplications and modulo operations, having branch operations, and lacking parallel computation of instructions.

To ensure that the algorithm handles such expensive operations efficiently, various optimization techniques were implemented. For example, software pipelining, loop unrolling, operator strength reduction, branch elimination, predicate operations with 2’s complement, and declaration of register variable and restrict pointers. To validate if optimizations are making any improvements, Valgrind was utilized to determine the number of instructions in optimized and unoptimized .c files. Similarly, the time library

was used to measure the execution time of the optimized and unoptimized .c files. After comparing such files, it was observed that optimized files reduced the number of instructions by 80.6%, but there was only a 10% improvement in execution time.

Therefore, we decided to develop a lookup table for 16 bits prime numbers since it was observed that generating such large integers using the GMP library was inefficient. After comparing the version of unoptimized code with optimized code with a lookup table, it was observed that the executive time was 61 times faster and the number of instructions was reduced by 74.4% when compared with the unoptimized algorithm. Using optimization techniques, the program was able to encrypt and decrypt $2^{31} - 1$ integer size.

Limitations

The main limitation of our RSA Cryptography is the size of the encrypted plaintext. Our solution is designed to support a plaintext input up to a size $2^{31} - 1$. This restriction is intentional, as it aligns with our focus on the 64-bit system to eliminate the overhead of handling sizes beyond the C standard's built-in size.

To further explain this limitation, the RSA Cryptography requires that the product PQ of prime numbers P and Q must be greater than the plaintext. We restrict the size of P and Q to 16 bits to prevent overflow when performing modular exponentiation. This is because squaring the PQ value can lead to a 64-bit overflow. For example, if $P \times Q = PQ, then\ 2^{16} \times 2^{17} = 2^{33}$, resulting in $(2^{33})^2 = 2^{66}$ when computing modular_exponentiation.

Another limitation is that our solution does not loop back to recompute the P and Q values. Instead, we use a seed to select sufficiently large P and Q values, assuming the P and Q seeds are chosen to be larger than the plaintext. However, we implemented a check to inform if the selected seeds violate this condition.

Future Work

Future works should mitigate the aforementioned two limitations. It should enable a large plaintext size of 512-bit or 1024-bit. The high-level solution involves splitting a larger plaintext into smaller pieces that fit within the $2^{31} - 1$ size limitation. Then, we encrypt each chunk separately. Eventually, we can reassemble the encrypted chunks for decryption. Regarding the limitation of P and Q product (PQ) being less than the plaintext, future work could involve designing an efficient loop to recompute PQ if the initial selected PQ is less than the plaintext.

Conclusion

In conclusion, it can be concluded that brute force approach can only handle T of up to 2 bits. This is due to its limitation of handling large integers when T^E is performed. So, the modular exponentiation with MMM algorithm was utilized to handle large integers. In addition, the implemented modular exponentiation with MMM algorithm is able to handle T of up to $2^{30} - 1$ size since it is repeatedly performing modulo operations to keep the intermediate variable within the integer range of PQ. But, the unoptimized version of modular exponentiation with MMM algorithm was slow due to the large number of instructions and use of memory intensive operations, such as multiplication, branching, division, and modulo.

However, this problem was solved by implementing fast and inexpensive operations to decrease the number of instructions and improve the runtime. For example, a lookup table was implemented to get the random prime numbers for P, Q, and E. Moreover, branches and multiplications were replaced with predicate operations along with 2's complement. Similarly, loop unrolling and software pipelining was performed in combination with operator strength reduction technique to decrease the number of instructions and improve the runtime performance. Furthermore, register and restrict keywords were used to declare variables and pointers to ensure that frequently used variables are stored in the CPU and pointers never overlap.

Even though the number of instructions were reduced and runtime was improved, the loop unrolling can possibly increase the memory usage. In addition, the implemented lookup for retrieving random prime numbers table is efficient for the purpose of this report since it is only focusing on generating 16 bits integers, but in real world lookup tables can be inefficient since the specific bit limit is not set for generating random prime numbers. Therefore, based on the design, optimizations, and discussions, it can concluded that an optimized version of main.c that does not include lookup table, i.e., optimized_no_lookup_main.c is the best solution to encrypt and decrypt plaintext T in terms of reducing the number of instructions such that P, Q, and E are up to 16 bits prime numbers. But, the optimized version of main.c that does include lookup table, i.e., optimized_main.c is the best solution to encrypt and decrypt plaintext T in terms of runtime performance such that P, Q, and E are up to 16 bits prime numbers.

References

[1] University of Victoria, “SENG 440”, SENG Software Repository - University of Victoria https://sw.seng.uvic.ca/repo/seng440/ (accessed June. 30, 2024).

[2] S. Weil, “QEMU Binaries for Windows (64 bit),” QEMU for Windows – Installers (64 bit), https://qemu.weilnetz.de/w64/ (accessed Jun. 30, 2024).

[3] gmplib, “5 Integer Functions”, Integer Function (GNU MP 6.3.0), https://gmplib.org/manual/Integer-Functions (accessed July. 04, 2024).

[4] Ankur, “Euclidean algorithms (Basic and Extended), Euclidean algorithms (Basic and Extended) - geeks for geeks, https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended (accessed July. 04, 2024).

[4] G. John, “Install Valgrind on Linux: Debian, Ubuntu, Arch, Fedora, RHEL, Raspberry, etc.,” Install Valgrind on Linux: Debian, Ubuntu, Arch, Fedora, RHEL, Raspberry, etc., https://www.linuxuntu.com/install-valgrind-linux/ (accessed July. 04, 2024).

[5] “20 Hardcodes Primes That Fit Into 16 Bit,” Harcodes Primes That Fit Into 16 Bit, https://www3.risc.jku.at/people/hemmecke/AldorCombinat/combinatse20.html (accessed July 30, 2024).

[6] armDeveloper, “Neon”, Neon, https://developer.arm.com/Architectures/Neon (accessed Aug 01, 2024).

[7] die.net, “gettimeofday(3) - Linux man page,” gettimeofday(3): date/time - Linux man page, https://linux.die.net/man/3/gettimeofday (accessed Aug. 01, 2024).

[8] Flexo - Save the data dump, “Timing functions in C [duplicate],” profiling - Timing functions in C - Stack Overflow, https://stackoverflow.com/questions/8129879/timing-functions-in-c (accessed Aug. 01, 2024).

[9] ggiroux, “GCC multiple optimization flags,” GCC multiple optimization flags, https://stackoverflow.com/questions/5557261/gcc-multiple-optimization-flags (accessed Aug. 01, 2024).

[10] gcc, “3.10 Options That Control Optimization,” Optimize Options - Using the GNU Compiler Collection (GCC), https://gcc.gnu.org/onlinedocs/gcc-3.3.6/gcc/Optimize-Options.html (accessed Aug. 04, 2024).

[11] S. Delić, “Branchless programming — Why your CPU will thank you,” Branchless programming | by Srdjan Delić | Medium, https://sdremthix.medium.com/branchless-programming-why-your-cpu-will-thank-you-5f405d97b0c8 (access Aug. 06, 2024)

[12] dcastro, “Will a long-integer work on a 32 bit system?,” Will a long-integer work on a 32 bit system? - stackoverflow, https://stackoverflow.com/questions/22363503/will-a-long-integer-work-on-a-32-bit-system (access Aug. 06, 2024)

Appendix

Question 1: Was there a difference in assembly by eliminating the branch in the optimized version of the code?

In an unoptimized assembly, we can see that the if statement results in a jump to a branch (bcc .L18) if it is a ‘less than’ case, otherwise continuing to process the code (See Figure 27).

In an Optimized assembly, we cannot see any jump, resulting from the branch elimination (see Figure 16 and 27).

We did further research and saw that this technique is called branchless programming/Conditional Move Instructions. These instructions allow data to be moved conditionally without creating a branch [11].

```
.L16:
        ldrd    r2, [fp, #-36]
        orrs    r3, r2, r3
        bne     .L17
        ldrd    r0, [fp, #-44]
        ldrd    r2, [fp, #4]
        cmp     r1, r3
        cmpeq   r0, r2
        bcc     .L18
        ldrd    r0, [fp, #-44]
        ldrd    r2, [fp, #4]
        subs    ip, r0, r2
        str     ip, [fp, #-108]
        sbc     r3, r1, r3
        str     r3, [fp, #-104]
        ldrd    r2, [fp, #-108]
        strd    r2, [fp, #-44]
```

```
.L18:
        ldrd    r2, [fp, #-36]
        orrs    r3, r2, r3
        bne     .L19
        ldrd    r0, [fp, #-44]
        ldrd    r2, [fp, #4]
        cmp     r1, r3
        cmpeq   r0, r2
        movcs   r3, #1
        movcc   r3, #0
        uxtb    r3, r3
        uxtb    r2, r3
        mov     r3, #0
        ldr     r1, [fp, #4]
        mul     r0, r3, r1
        ldr     r1, [fp, #8]
        mul     r1, r2, r1
```

Figure 27: Compare assembly before and after branch elimination

Question 2: How will uint64_t work in a 32 bit system architecture?

As shown in figure 28, if the system is a 32-bit architecture that is not able to handle uint64_t, it will utilize ldrd operation to load dual registers, meaning it separates 64 bit unsigned integers into two 32 bit unsigned integers [12].

```
.L13:
        ldrd    r2, [fp, #4]
        strd    r2, [sp]
        ldrd    r2, [fp, #-60]
        ldrd    r0, [fp, #-84]
        bl      montgomery_modular_multiplication
        strd    r0, [fp, #-68]
        ldrd    r2, [fp, #-92]
        mov     r0, #1
        mov     r1, #0
        and     r8, r2, r0
        and     r9, r3, r1
        orrs    r3, r8, r9
        beq     .L12
        ldrd    r2, [fp, #4]
        strd    r2, [sp]
        ldrd    r2, [fp, #-60]
        ldrd    r0, [fp, #-36]
        bl      montgomery_modular_multiplication
```

Figure 28: Load uint64_t with ldrd instructions from function parameters in a 32 bit architecture