[1]

RSA Cryptography with Montgomery Modular Multiplication

SENG 440 - Embedded Systems

Adam Leung
David Mah

Table of Contents

Introduction **3**
- Purpose 3
- RSA Cryptography with Montgomery Modular Multiplication 3
- Contributions and Project Organization 3

Theoretical Background **4**

Design Process **5**
- Implementation Design 5

Optimization and Performance Testing **9**
- Increment and Exit Condition 10
- Constants 10
- Removing Unused Variables 10
- Loop Unroll 10
- Pipeline 11
- Loop fusion 11
- Reduce operator Strength 11
- Pipeline With 2 Loop Unrolls Instead of 4 11
- Registers 11
- Reduce Operator Strength for Constants 12
- Remove Unnecessary Operations 12
- Assembly Code Optimization 13

Conclusion **14**

References **15**

Appendix A: Performance Results **15**

Introduction

Purpose

This report covers the coding structure, models, performance, and optimization steps taken to implement the RSA Cryptography efficiently. Montgomery Modular Multiplication will be used to perform the encryption and decryption methods, in place of the standard modulus operations.

RSA Cryptography with Montgomery Modular Multiplication

RSA Cryptography is a public key system primarily used for secured data transmissions, however using the standard encryption and decryption methods is extremely expensive. With Montgomery Modular Multiplication, we will be able to compute large plain and ciphertexts with large keys and prime numbers.

Contributions and Project Organization

Both parties performed primarily separate tasks, with review being done at the end of each task, or supervision while a task was being performed.

Adam Leung primarily focused on understanding how Montgomery Modular Multiplication works in terms of why it results into a scaled value, and how to effectively prescale the input to give a proper output. In addition, he generated multiple keys which are used in the implementation of the program. After the code was working properly to encrypt and decrypt, he focused on optimizing parts of the application with optimization techniques learned in class, as well as cleaning up the code. After the C code was finalized, he turned towards the assembly code, analyzing it for areas to improve, as well as testing the new assembly code. Additionally, he created UML diagrams during the implementation and testing of the code. Adam Leung wrote in his respective sections according to the task done prior, and reviewed the report prior to submission.

David Mah primarily focused on understanding how the RSA Cryptography works with Montgomery Modular Multiplication. He initially started with simply implementing a skeleton for the code, as rough design on how the code will be laid out, as well as the inputs and outputs of the code. Additionally, he focused on getting the basics of the program to work, by implementing the Montgomery Modular Multiplication algorithm from the slides. After the code was working, he as well focused on optimizing the code. Additionally for each stage of the optimization steps, he tested each iteration, gathering performance metrics and ensuring no bugs were introduced. Additionally, he focused on cache misses and declaring register declaratives for specific variables. David Mah wrote in his respective sections according to the task done prior, and reviewed the report prior to submission.

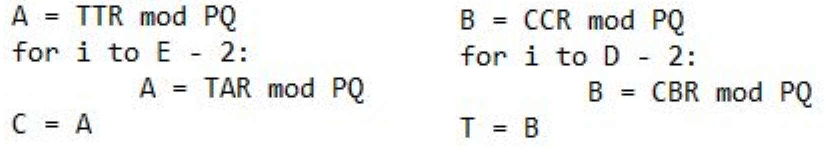
Theoretical Background

RSA Cryptography utilizes large prime numbers to generate keys to encrypt and decrypt messages and data between parties. In order to generate the keys, 2 large distinct prime numbers are chosen, P and Q. A public key is generated by multiplying P and Q together, generating PQ. The 2 large prime numbers P and Q should be destroyed after the key generation. After, a separate public key E must be chosen such that E > 1, E < PQ, and E and (P - 1)(Q - 1) are relatively prime. With E, a private key D will be generated such that $D = (X(P-1)(Q-1)+1)/E$ where X is an integer that causes D to also be an integer. With PQ, E, and D, a message can be encrypted using PQ and E, and decrypted using PQ and D. The encryption function is $Ciphertext = T^E mod\,PQ$ where T is the plain text. The decryption function is $Plaintext = C^D mod\,PQ$ where C is the ciphertext from the encryption function. [2]

While the above method is simple and easy to follow, the encryption and decryption methods are notoriously slow, as T^E and C^D may be extremely large that may result in integer overflows, and slow modulus operations. Such expensive modular and multiplication operations can be resolved using Montgomery Modular Multiplication, which is able to calculate $Z = XYR^{-1}\ mod\,M$ effectively replacing the basic encryption and decryption methods. [2]

Montgomery Modular Multiplication results in a scale factor of the X and Y by R^-1. When Montgomery Modular Multiplication is performed once, the pre scaling required to counter the scaled result is rather inefficient and expensive. On the other hand, when this operation is performed for a large number of times, similar to Modular Exponentiation where large modular operations are separated and performed in chunks, this expensive scaling method becomes rather negligible, and fairly efficient compared to the standard modulus operation mentioned in the first paragraph of this section. [2] [3]

Additionally, since Montgomery Modular Multiplication only performs the modulus operation on the plaintext of ciphertext twice, $TT\ mod\,PQ$ where T is the plaintext, we can take the remainder or result from the prior operation, and perform the Montgomery Modular Multiplication again with another plaintext or ciphertext achieving the $T^E\ mod\,PQ$. A pseudo code is shown below.



```
A = TTR mod PQ
for i to E - 2:
        A = TAR mod PQ
C = A
```

```
B = CCR mod PQ
for i to D - 2:
        B = CBR mod PQ
T = B
```

Figure 1: Pseudo code of modified encryption on the left and decryption method on the right using Montgomery Modular Multiplication for performing $T^E mod\,PQ$ and $C^D mod\,PQ$

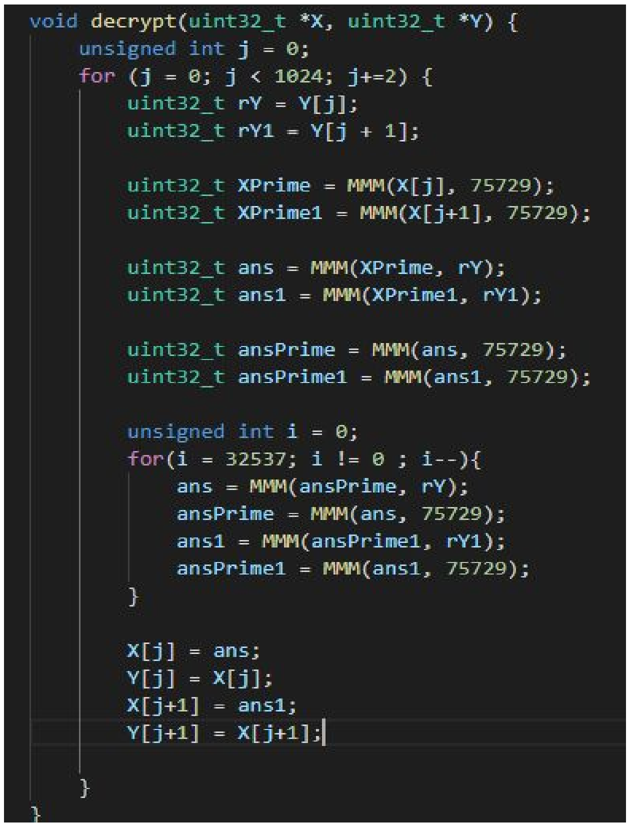
A or B represents the remainder from the previous iteration of the Montgomery Modular Multiplication. The final result represents the ciphertext or plaintext respectively.

Design Process

Implementation Design

As the Montgomery Modular Multiplication scales the answer by a factor of R^{-1} every time it is used, the inputs have to be prescaled by a factor of R beforehand. R is calculated by raising 2 to the power of m where m is the bit-length of PQ. This R is chosen because when you calculate $R*R^{-1}$mod PQ, you get 1, essentially cancelling out R^{-1}. However, in order to prescale the initial input, Montgomery Modular Multiplication has to be used, causing the final answer to be scaled by a total factor of R^{-2}. Thus, in order to properly cancel out the R^{-1} scale introduced by Montgomery Modular Multiplication, the input had to be prescaled by R^2.

In the original encryption method, as we are performing $Ciphertext\ = T^E mod\ PQ$, this can be rewritten as $Ciphertext\ = (TT\ ....\ TT)\ mod\ PQ$. From here, we can change this to $A = TTR\ mod\ PQ$, and then perform $A\ =\ TAR\ mod\ PQ$ repeatedly E - 2 times to achieve the ciphertext. The original decryption method will be changed in a similar manner, however performing D - 2 times, using the ciphertext instead of the plaintext. Please refer to the figure 1 above as it shows pseudo code on how the code is structured to perform $T^E mod\ PQ$ and $C^D mod\ PQ$. The figure below is a snippet of the code implemented with optimization techniques. The prescaling value R is 75729 in this application.



```
void decrypt(uint32_t *X, uint32_t *Y) {
    unsigned int j = 0;
    for (j = 0; j < 1024; j+=2) {
        uint32_t rY = Y[j];
        uint32_t rY1 = Y[j + 1];

        uint32_t XPrime = MMM(X[j], 75729);
        uint32_t XPrime1 = MMM(X[j+1], 75729);

        uint32_t ans = MMM(XPrime, rY);
        uint32_t ans1 = MMM(XPrime1, rY1);

        uint32_t ansPrime = MMM(ans, 75729);
        uint32_t ansPrime1 = MMM(ans1, 75729);

        unsigned int i = 0;
        for(i = 32537; i != 0 ; i--){
            ans = MMM(ansPrime, rY);
            ansPrime = MMM(ans, 75729);
            ans1 = MMM(ansPrime1, rY1);
            ansPrime1 = MMM(ans1, 75729);
        }

        X[j] = ans;
        Y[j] = X[j];
        X[j+1] = ans1;
        Y[j+1] = X[j+1];

    }
}
```

Figure 2: Code snippet of the decryption method

To accept large inputs, since we are unable to store large messages into a single integer variable, we will be storing it into an array of size 1024 of 32 bit integers. The array size can be increased if need be up to the limit of the system. Since our modulus key is only 17 bits, we will only be using 16 bits of the 32 bit integers. During the encryption or decryption operations, we will only perform the operation in chunks of 16 bits because of the key size restriction. The input message, and keys are all hard coded into the application, generated outside of the application.

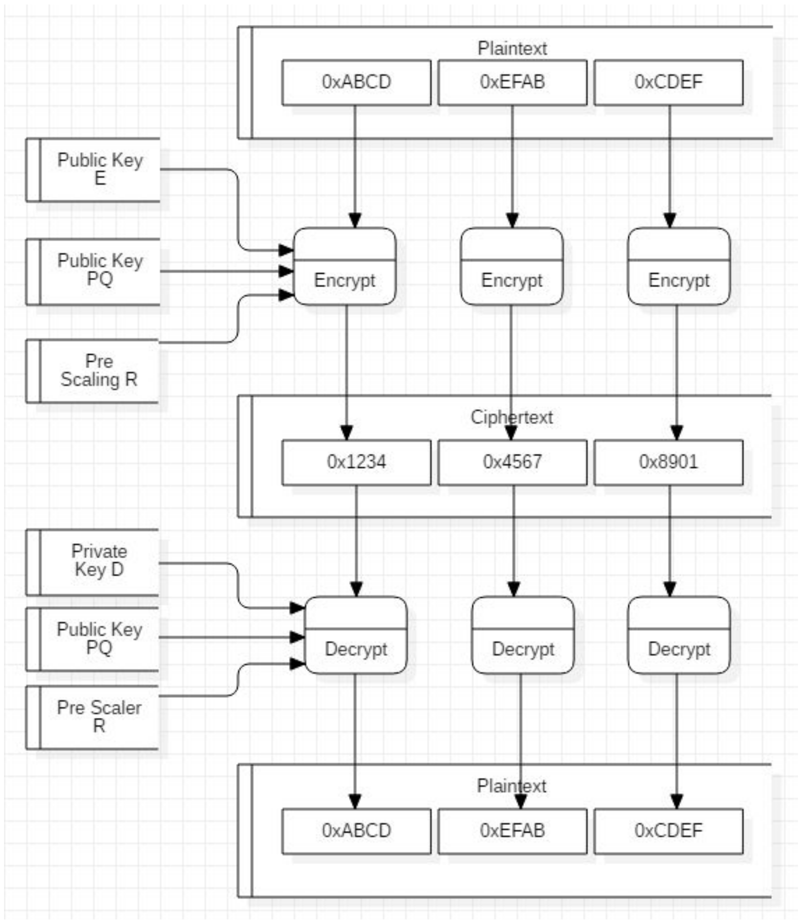


Figure 3: Encryption and decryption design for large plaintext input

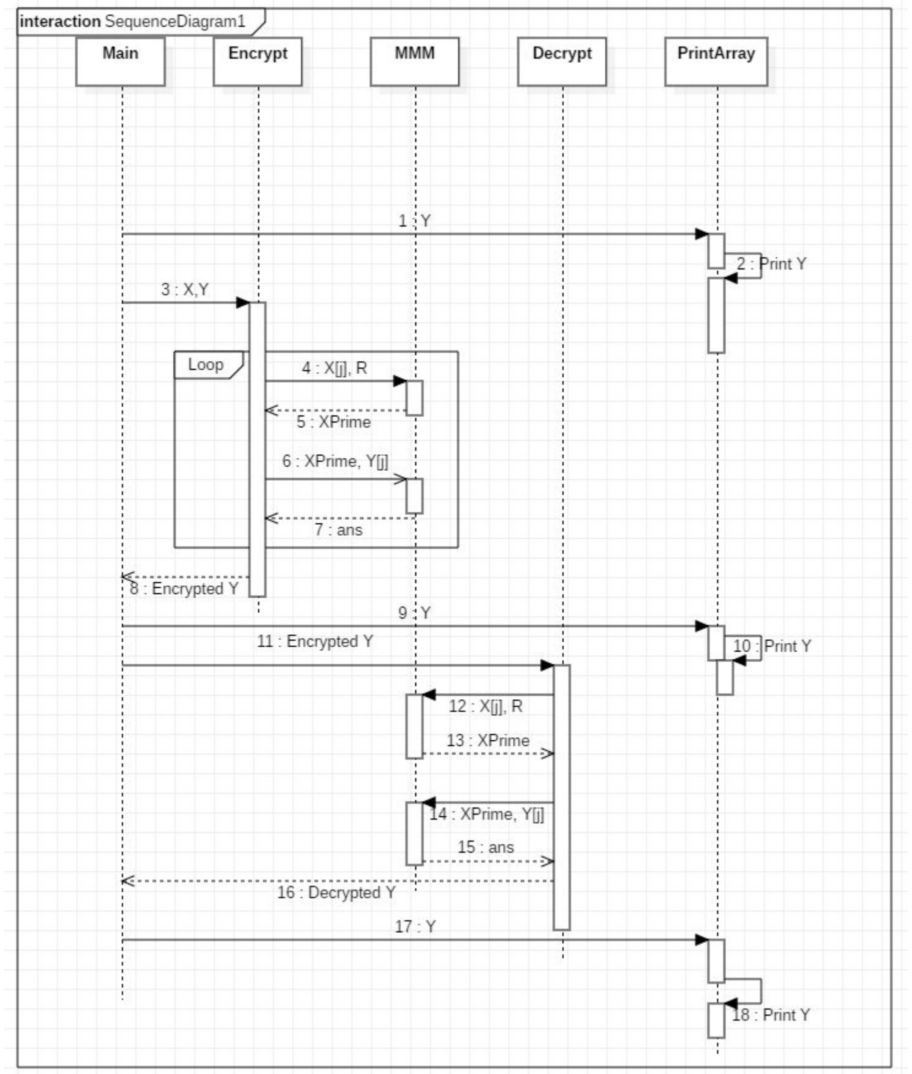
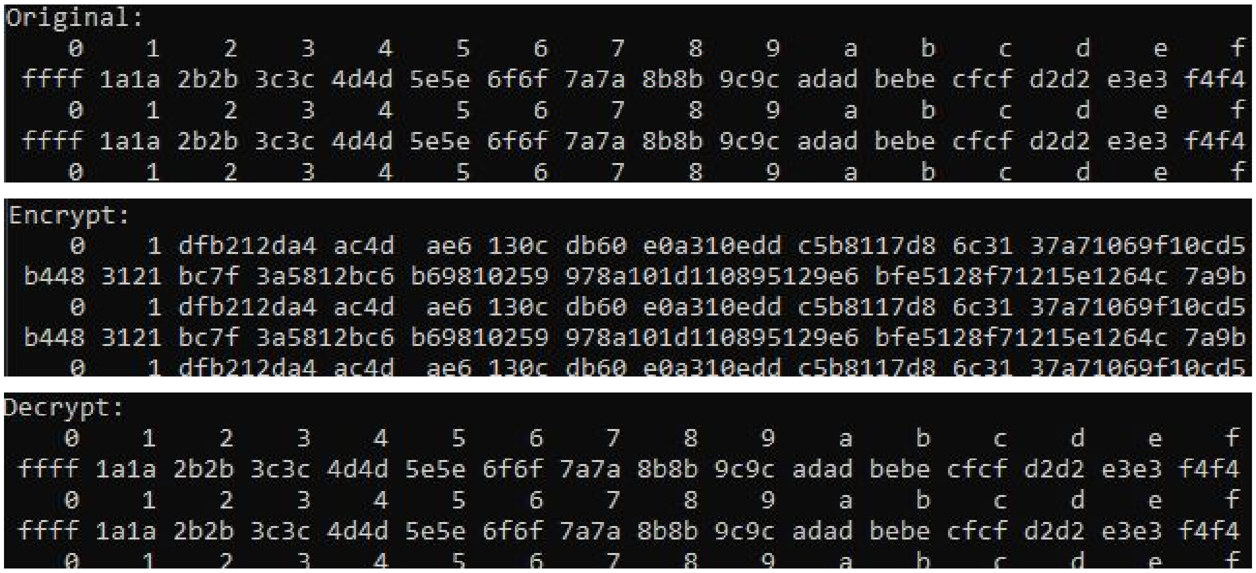


Figure 4: Sequence diagram of encrypting and decrypting a message

From the data flow diagram and the sequence diagram above, figure 3 and figure 4, the application begins with encrypting the message. The message is separated into 16 bit chunks, that occurs over 1024 times. Each chunk is used to calculate a scale value that is applied to the chunk prior to performing Montgomery Modular Multiplication to counter the scaled result. Montgomery Modular Multiplication is performed on the chunk E or D times, the public or private key respectively. After the chunk is converted to a ciphertext, it is then stored back into the message replacing the plaintext chunk with a ciphertext chunk. The exact same method is performed for decryption. For demonstration purposes, in figure 5. below shows the data values stored in the message array, encrypted array, and decrypted array.



```
Original:
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
 ffff 1a1a 2b2b 3c3c 4d4d 5e5e 6f6f 7a7a 8b8b 9c9c adad bebe cfcf d2d2 e3e3 f4f4
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
 ffff 1a1a 2b2b 3c3c 4d4d 5e5e 6f6f 7a7a 8b8b 9c9c adad bebe cfcf d2d2 e3e3 f4f4
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
```

```
Encrypt:
   0    1 dfb212da4 ac4d  ae6 130c db60 e0a310edd c5b8117d8 6c31 37a71069f10cd5
 b448 3121 bc7f 3a5812bc6 b69810259 978a101d110895129e6 bfe5128f71215e1264c 7a9b
   0    1 dfb212da4 ac4d  ae6 130c db60 e0a310edd c5b8117d8 6c31 37a71069f10cd5
 b448 3121 bc7f 3a5812bc6 b69810259 978a101d110895129e6 bfe5128f71215e1264c 7a9b
   0    1 dfb212da4 ac4d  ae6 130c db60 e0a310edd c5b8117d8 6c31 37a71069f10cd5
```

```
Decrypt:
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
 ffff 1a1a 2b2b 3c3c 4d4d 5e5e 6f6f 7a7a 8b8b 9c9c adad bebe cfcf d2d2 e3e3 f4f4
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
 ffff 1a1a 2b2b 3c3c 4d4d 5e5e 6f6f 7a7a 8b8b 9c9c adad bebe cfcf d2d2 e3e3 f4f4
   0    1    2    3    4    5    6    7    8    9    a    b    c    d    e    f
```

Figure 5: Console output of the RSA Cryptography implementation

Because of the public key choice of being 19, the private key is 32539, resulting in the decryption method to be extremely slow compared to the encryption method, effectively taking at least 97% of the execution time of the application.

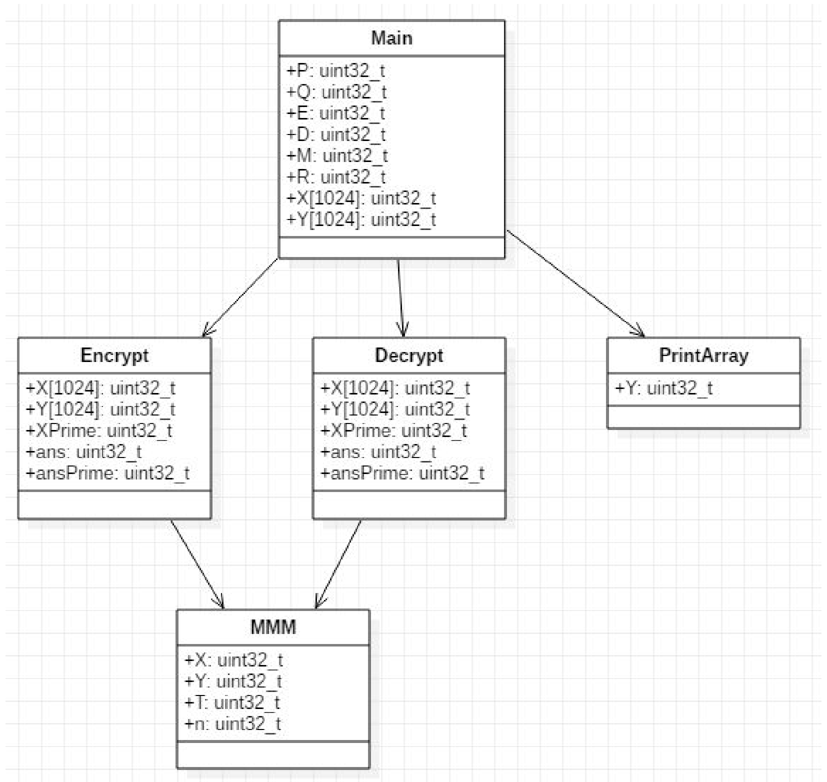


Figure 6: Class diagram of all the functions used with their respective variables

Optimization and Performance Testing

Each section below represents optimization attempts to optimize the code. Each attempt is done sequentially in the order of the sections building on top of the previous optimization, each containing a description, and performance metrics of the optimized code. Prior to optimization, only the algorithm as seen below was implemented with some optimizations implemented already such as direct indexing to an array, and passing a pointer to a variable instead of instantiating a new variable. All code is tested with a 16384 bit long input. The 2 large prime numbers are 277 for P, and 281 for Q. Multiplying these 2 together equals 77837 for PQ. Choosing an E of 19 for our public key, our private key E effectively becomes 32539 for our private key. After each test, the execution time is collected using gprof, and the instruction count and cache misses were collected from the tool valgrind.

Optimization Method	**Average Execution Time (s)**
Original Code	8.66
Increment and Exit Condition	8.58
Constants	8.53
Removing Unused Variables	8.51
Loop Unroll	8.51
Pipeline	8.47
Loop Fusion	8.49
Reduce Operator Strength	330.67
Pipeline with 2 Unrolls	8.46
Registers Variable Declaration	4.12
Reduce Operator Strength for Constants	4.9475
Removing Unnecessary Operations	3.675
Percent Difference Between Original and Best (C Code Only)	57.61%
Original Assembly Code	3.66
Optimized Assembly Code	3.31
Assembly Code Percent Difference	9.67%

Percent Difference Between Original and Best	61.86%

Figure 7: Average execution times of each optimization method

In the above figure shows the average execution time for each optimization method. Each optimization method builds on top of each other, except for the Reduce Operator Strength method, as this method did not make the RSA Cryptography more efficient. The last row indicates the percent difference between the final optimized code, and the original code to start with. Please refer to Appendix A for the full test results

Increment and Exit Condition

Where possible, we changed the for loop counters to decrement instead of increment, and the exit conditions to be != 0. The for loops that were unchanged were ones that relied on the values of the counter to access certain values. While it is possible to have them decrement and have a conversion method to reuse the counter, the conversion method would result in degraded performance. These changes were implemented to reduce both the latency between calls as each loop would perform a simple check against the 0 flag instead of doing a range comparison, as well as the potential need to perform memory access calls. With the optimization implemented, it improved the runtime of the RSA Cryptography algorithm by 1%.

Constants

The next code optimization we did was to replace variables with its values when the values were known beforehand.These values were the prime numbers chosen at the beginning of the algorithm as well as the calculated R, E and D values pertaining the encrypt and decrypt functions. This optimization did two things. The first was that it eliminated the subexpressions needed to calculate R, E and D. The second was that it reduces the need, as well as the cost to repeatedly access variables from memory. With constants, it took an average of 8.53 seconds, 0.58% improvement from the previous optimization.

Removing Unused Variables

Following the replacement of known variables with their respective values, the variables declarations themselves were removed as they were no longer needed. This optimization allowed us to reduce the cost of storing variables. This resulted in a 0.23% improvement, a small increment however effectively cleaning up the code base, as well as useless instruction cycles.

Loop Unroll

The next code optimization performed was loop unrolling. Since we are performing encryption and decryption in chunks, with each chunk being independent from each other, we are able to perform 4 encryption or decryption operations in 1 iteration of the loop. This effectively reduces the loop overhead, and allows for the potential of parallel operations as seen from pipelining and loop fusion. With only loop unrolling, the encryption and decryption methods were improved by

0.04%, an extremely small increment however enables us to further improve the code in later optimization methods.

Pipeline

Following loop unrolling, we rearranged the order of which some code was run in order to apply software pipelining. By rearranging the code, we allow independent operations to run in parallel with each other, without having them to wait to fetch data when the previous operation is currently running, effectively increasing the overall performance of the program by 0.5%.

Loop fusion

After pipelining we noticed that there were multiple for loops that ran the same number of times, with the same exit and increment conditions. We performed loop fusion on them by combining the contents of all the similar for loops and placing them all within one for loop, cleaning up the code base and reducing the amount of loop comparisons to be performed. However in the end, this reduced our code performance by 0.25%. This is because we are constantly switching between 4 independent operations, resulting in new data to be loaded into the caches, and other data being removed increasing the number of cache misses.

Reduce operator Strength

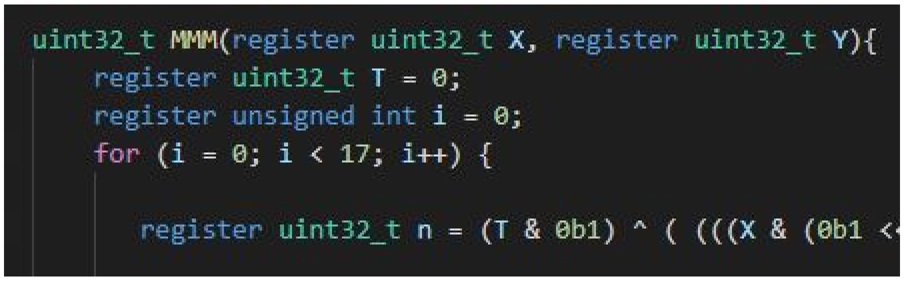
An alternate multiply method was created using for loops and bit shifts to replace the * operator. However, since products being multiplied were not always powers of two, the method was placed in a separate function to with additional logic to handle inputs that were not powers of 2. When we implemented this optimization, the additional function call and logic ended up costing more than the original * operator, decreasing the overall performance instead of increasing it. This optimization method will not be implemented.

Pipeline With 2 Loop Unrolls Instead of 4

Previously, when we performed loop fusion, we fused 4 for loops together since we also loop unrolled by 4. However, instead of increasing performance, this change ended up increasing the run-time instead. This may have been the result of reduced data locality within the for loop after combining 4 loops together. With this observation, we decided to loop unroll by 2 instead of 4. This improved data locality, reducing the number of cache misses, and increasing the performance of the code by 0.29%

Registers

All previous operations were mostly performed on not the Montgomery Modular Multiplication as the algorithm implemented was already fairly efficient. Since Montgomery Modular Multiplication is called and performed many number of times, we found that it would be effective to store all the variables used in the Montgomery Modular Multiplication in registers. By declaring a variable with a register attribute, this suggests to the compiler to store this variable in a register, granting faster read and write access to the processor. This resulted in a 51% increase in performance compared to the previous optimization.



```
uint32_t MMM(register uint32_t X, register uint32_t Y){
    register uint32_t T = 0;
    register unsigned int i = 0;
    for (i = 0; i < 17; i++) {

       register uint32_t n = (T & 0b1) ^ ( (((X & (0b1 <
```

Figure 8: Register declaratives in the Montgomery Modular Multiplication method

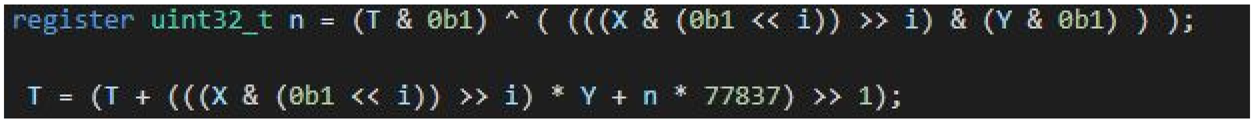
We attempted to add the register attribute to other variables in the code however this did not change the performance of the system by any measurable amount, and therefore was only applied in the Montgomery Modular Multiplication algorithm itself. This could of been that the register declaration was ignored by the compiler as the Montgomery Modular Multiplication algorithm is a separate function call. Additionally, the Montgomery Modular Multiplication algorithm is performed more than 553,129 times where as other functions and variables would only be accessed less than 40,000 times, therefore if they were stored in the register, the performance difference is negligible.

Reduce Operator Strength for Constants

Instead of replacing all multiply functions, we only focused on changing the multiply function with constants. By doing this, there will be less strain on multiplying and instead being replaced with a series of addition, however in the end, this actually reduced the performance by 20% compared to the previous optimization. This could be that it requires many more cycles to perform the modified multiplication operation with only shifts and additions, rather than simply performing the multiplication instruction.

Remove Unnecessary Operations

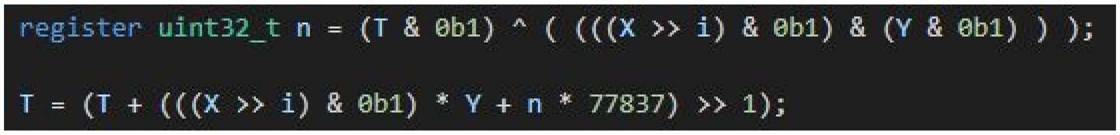
Furthermore, we removed unnecessary shift operations from our C code inside the Montgomery Modular Multiplication increasing the efficiency by 15.37%.



```
register uint32_t n = (T & 0b1) ^ ( (((X & (0b1 << i)) >> i) & (Y & 0b1) ) );

 T = (T + (((X & (0b1 << i)) >> i) * Y + n * 77837) >> 1);
```

Figure 9: Unoptimized C Code



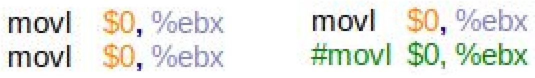
```c
register uint32_t n = (T & 0b1) ^ ( (((X >> i) & 0b1) & (Y & 0b1) ) );

T = (T + (((X >> i) & 0b1) * Y + n * 77837) >> 1);
```

Figure 10: Optimized C Code for removing unnecessary operations

Assembly Code Optimization

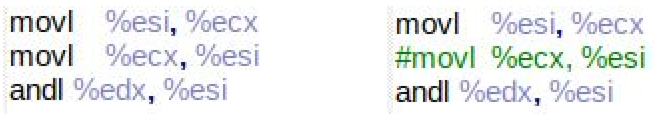
After optimizing the C code, we generated the assembly equivalent to see what further optimizations could be done. Before optimizing, we profiled the assembly code and got an average 3.66s total execution time. As the program was already optimized in C previously, the only optimizations done to the assembly code was removing dead and useless code. There were three common types of dead code found in the program. The first was duplicate movl calls that appeared right after the other. In order to fix this, we commented out the second call of the duplicate movl as seen in Figure 11.



```
movl   $0, %ebx        movl   $0, %ebx
movl   $0, %ebx        #movl  $0, %ebx
```

Figure 11: Original and Optimized Assembly Code

The second most common type of dead code was redundant movl calls. There were two types of this, one where register A is moved to register B, and then register B is moved to register A, and the other where register A is moved to register A. One example of the redundant calls as well as our fix is shown in Figure 12.



```
movl   %esi, %ecx      movl   %esi, %ecx
movl   %ecx, %esi      #movl  %ecx, %esi
andl %edx, %esi        andl %edx, %esi
```

Figure 12: Original and Optimized Assembly Code

The final type of dead code were push and pop instructions on a register where the registers were not used in between the push and pop calls.

The result of removing all the dead code led to an average of 3.30625 seconds total execution time. This was a 9.665% increase in performance.

Conclusion

From the original RSA Cryptography method, to using Montgomery Modular Multiplication to perform the modulus and large power operations, the code was able to run in 8.67 seconds without any optimizations on an input of 16384 bits using a 9 bit private key. After performing a series of optimization to the C code, the average execution time was reduced to 3.675 seconds, a 57.61% in improvement.

Furthermore, analyzing the assembly code, the execution time was able to be reduced by 9.665% from removing duplicate or unnecessary operations achieving an overall efficiency improvement of 61.86%.

In the future, it would be highly recommended to expand the current implementation in accepting larger private keys, as it would increase the security of the system. This will however require many implementation changes as C does not natively support 512 bit integer storage or operations.

References

[1] University of Victoria. “CourseSpaces.” [Online] Available: https://coursespaces.uvic.ca/my/ [Accessed: July 30th, 2019]

[2] M, Sima. SENG 440. Class Slides. “Lesson 108: RSA Cryptography.” University of Victoria, 2019

[3] J. GroBschadl; Z, Liu. “New Speed Records for Montgomery Modular Multiplication on 8-bit AVR Microcontrollers.” University of Luxembourg. [Online]. Available: https://eprint.iacr.org/2013/882.pdf [Accessed: July 30th, 2019]

Appendix A: Performance Results

Optimization Method	Original	Increment and Exit Condition	Constants	Removing Unused Variables	Loop Unroll	Pipeline
Test 1 (s)	8.79	8.52	8.35	8.56	8.57	8.31
Test 2 (s)	8.58	8.65	8.36	8.52	8.44	8.38
Test 3 (s)	8.63	8.56	8.64	8.52	8.42	8.58
Test 4 (s)	8.75	8.54	8.47	8.4	8.66	8.5
Test 5 (s)	8.58	8.48	8.55	8.43	8.56	8.37
Test 6 (s)	8.69	8.64	8.66	8.58	8.47	8.61
Test 7 (s)	8.65	8.67	8.59	8.51	8.41	8.55
Test 8 (s)	8.68	8.59	8.63	8.57	8.53	8.43
Average (s)	8.66875	8.58125	8.53125	8.51125	8.5075	8.46625
Difference (s)	--	0.0874999999999986	0.0500000000000007	0.0199999999999996	0.0037500000000000014	0.04125000000000016
Ratio	--	0.01009372	0.00582665	0.00234432	0.0004405	0.0048486

		74693582	695557182	23443223	93332354253	6294446096
Percent Difference (%)	--	1.00937274693582	0.582665695557182	0.23443223443223	0.0440593332354253	0.484866294446096

Optimization Method	Loop Fusion	Pipeline with 2 Unrolls	Registers	Reduce Operator Strength for Constants	Remove Unnecessary Operations	Assembly Code Original	Assembly Code Optimized
Test 1 (s)	8.6	8.6	3.81	4.96	3.76	3.66	3.31
Test 2 (s)	8.28	8.28	4.29	5.01	3.66	3.66	3.30
Test 3 (s)	8.43	8.43	4.26	5.06	3.60	3.65	3.30
Test 4 (s)	8.46	8.46	4.19	4.97	3.72	3.67	3.31
Test 5 (s)	8.46	8.46	4.16	4.98	3.73	3.66	3.31
Test 6 (s)	8.77	8.57	4.15	4.89	3.60	3.66	3.31
Test 7 (s)	8.61	8.61	4.11	4.72	3.62	3.66	3.31
Test 8 (s)	8.29	8.29	3.99	4.99	3.71	3.66	3.30
Average (s)	8.4875	8.4625	4.12	4.9475	3.675	3.66	3.30625
Difference (s)	-0.021250000000002	0.025000000000021	4.3425	-0.8275	0.6675	--	0.35375

Ratio	-0.00250996604163614	0.0029455081001 4752	0.51314623338257	-0.2008495145631	0.153713298791	--	0.09665300546448
Percent Difference From Previous (%)	-0.250996604163614	0.29455081001 4752	51.314623338257	-20.08495145631	15.371398791	--	9.665300546448