SENG 440
RSA Cryptography

University of Victoria

Instructor: Dr. Mihai Sima
Date: August 17, 2020

Group 19

Jonah Koebernick	V00897331
Scott Theriault	V00808918

Table of Contents

Table of Figures

Introduction

With the rapidly increasing reliance on digital communication of sensitive data, most notably through the internet and internet of things (IoT) enabled devices, people and companies are becoming increasingly susceptible to hackers and packet capturing of data. This data can include a wide range of sensitive data from smart meter readings, autonomous car commands, or critical operation instruction to a multi-million dollar piece of IoT machinery. Therefore, information security is becoming an increasingly important field. A notable technique to combat packet capturing and hackers is encryption, which scrambles the message to render it useless to attacks.

The project we selected for the summer 2020 session of SENG 440 was Rivest-Shamir-Adleman (RSA) encryption, which dates back to 1977 and is widely used in cryptosystems today [1]. The goal of the project was to reduce the run-time and memory usage of the encryption and decryption on an embedded system. The solution system will be able to encrypt and decrypt the full set of ASCII characters. The ASCII set was specifically chosen as it can convey any text-based message necessary and is universal between machines.

The solution system is written in C code with assembly code optimization in the bottleneck areas that were discovered. Optimization was specific to the ***S3C2440A 32-Bit CMOS Microcontroller*** that is available over the University of Victoria (UVic) engineering network. This machine was chosen to allow for remote access during the COVID-19 pandemic.

The contribution was split evenly among the team members to ensure that the time spent was equal. Having a remote team required collaboration tools where both members could work on the project simultaneously. These tools included GitHub and Repl.it for version control/source code management, Google Docs for real-time collaborative reports, and the UVic intranet to use the physical embedded microcontroller.

In this report, the requirements set forth by Dr. Sima and through discussion prior to starting the project will be introduced. Background information will be presented that is necessary for an understanding of the material - namely, RSA cryptography and the tables of powers method for computing exponentials. The design process used to develop the software and optimization will be discussed, stating the approach to meet the various requirements. Furthermore, performance analysis will be presented, which will describe which techniques worked best and what significant limitations we encountered. Finally, a brief discussion on possible future work that could be pursued if in-person collaboration was feasible or a larger timeline for project delivery was available.

Requirements

The requirements for various aspects of the project are presented below. This section details the considered specifications given at the start of the project, metrics to analyze the performance of the optimizations, and the hardware limitations of the embedded system.

Specification Requirements

R1: Implement the Table of Powers method rather than the Montgomery Modular Exponentiation (MME) method. Calculate the memory required for the tables.

R2: Use 16-bit integers.

R3: M = P*Q is a 16-bit integer.

R4: Define a new instruction to activate a hardware assist that stores all tables of powers.

Design Metrics

During the course of executing multiple optimizations, various metrics were required to provide insight into whether an optimization was successful or not. These metrics included:

- Execution time of a custom test suite of performance tests
- Detailed analytics using Gprof, which provided valuable metrics of time spent in each function, allowing bottlenecks to be discovered

Hardware Requirements

During the course of this project, access to the compiler and the ARM processor was exclusively limited to remote connections. A GCC compiler over the UVic Engineering intranet and a remote ***S3C2440A Samsung*** ARM machine was utilized. The ARM machine has the following hardware constraints:

- Clock Speed: 1.30V for 400MHz on the ARM9TDMI
- Cache size: 64-way set-associative cache with I-Cache (16KB) and D-Cache (16KB)
- RAM: 128M-bytes for each bank (total 1G-byte)
- System memory: NAND Flash
- Instruction set: Reduced Instruction Set Computer (RISC)

Background

RSA encryption uses a public key infrastructure (PKI). A PKI consists of a public key that is available to the sender via plain text and a private key that is only known by the receiver. The public key is used to encrypt the message, while the private key is used to decrypt the message. The keys are chosen in a way that makes it virtually impossible to compute the private key given only the public key, resulting in an extremely secure encryption. Key creation will be covered in the theoretical background section. The flow of this cryptography exchange can be seen in Figure 1.

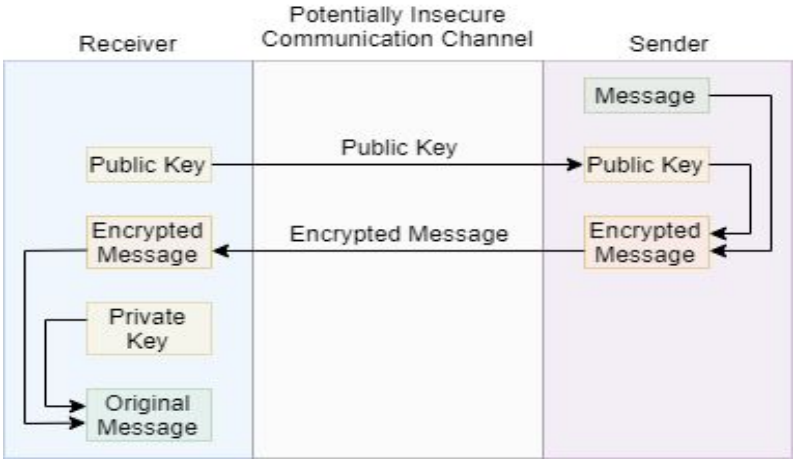

Figure 1: RSA encryption flow [2]

When utilised properly, RSA encryption is extremely secure. It relies heavily on two aspects: choosing the keys (public and private) and efficiently computing large exponents. The method for choosing the public and private keys is described below.

1. Select P and Q, such that P and Q are two large prime numbers.
2. Select E such that $E > 1$ and $E < PQ$
    a. E and $(P-1)(Q-1)$ are relatively prime
    (i.e. E and $(P-1)(Q-1)$ have no prime factors in common)
    b. E must be odd
3. Compute D such that $(DE-1)$ is evenly divisible by $(P-1)(Q-1)$
    a. To do so, find an integer X which causes
    $D = (X(P-1)(Q-1) - 1)/E$
    to be an integer.

Once these values are calculated, the public key is given by the pair (PQ, E), while the private key is D. Additionally, the encryption and decryption functions can be determined using these calculated values. The encryption function is given by

$$C = T^E mod\ (PQ) \tag{1}$$

where C and T are the ciphertext and plaintext, respectively. The decryption function is given by

$$T = C^D mod\ (PQ) \tag{2}$$

where C and T are the ciphertext and plaintext, respectively.

As seen in equations (1) and (2), encryption and decryption are both fairly rudimentary and rely on the choice of the keys to be effective and functional. The most notable challenge for these functions is the computation of large exponentials in a timely manner. Both computations become very complex with large values of E and D. To overcome this issue, the Montgomery Modular Expansion (MME) or the Tables of Powers method is utilized to compute the exponential.

Montgomery Modular Expansion (MME): MME is a common and dynamic way to efficiently compute large exponents. It will be discussed for completeness but it was not utilized in the source code of this project. MME uses the *multiply and square algorithm*. To perform the algorithm, the first exponent is converted to a binary representation where it can be split into sub-problems and solved individually. This process is shown below with the value of 13 for the exponent as an example.

This example shows how to compute $5^{13} mod\ 10$ using MME. The exponent, 13, can be represented in binary as 1101. Therefore, we need to compute three numbers in total.

$$A = 5^0 mod\ 10$$
$$B = (5^2 \times A)\ mod\ 10$$
$$Result = (5^3 \times B)\ mod\ 10$$

Tables of Powers: The Tables of Powers method is used less frequently due to the memory required to store many tables of powers, but is a useful option if the keys don't change often. This method was used in the development of the source code in the project as per requirement R1. The Tables of Powers method utilizes the same logic as MME, except the intermediate values are pre-computed in a table. This process is shown below with the value of 13 for the exponent as an example.

This example shows how to compute $5^{13} mod\ 10$ using this method. The exponent, 13, can be represented in binary as 1101. Therefore, we need to compute three numbers in total.

$$\begin{array}{rl} & 5^0 \quad 5^1 \quad 5^2 \quad 5^3 \\ table & = [1, \quad 5, \quad 25, \quad 125] \end{array}$$

$$A = table[0] mod\ 10$$
$$B = (table[2] \times A)\ mod\ 10$$
$$Result = (table[3] \times B)\ mod\ 10$$

As seen above, the table of powers method is based on MME but it requires less computation in real time, at the cost of memory. The table that stores the powers can quickly become too large and impractical to compute if the keys change often, resulting in this method being less commonly used. However, it is useful with a static key and large amounts of memory or a truncated set of values that are trivial to compute.

Design

During the course of the project, many design decisions were required in order to fulfill the requirements listed above. The following section describes the approaches that were used and calculations were necessary to satisfy the various requirements.

Specification Requirements

R1: Implement the Table of Powers method rather than the Montgomery Modular Exponentiation (MME) method. Calculate the memory required.

Approach: The ASCII character set was chosen as the range of values to encrypt and decrypt using the Table of Powers method of calculating exponentials. The tables of powers for all the ASCII characters require two 2-dimensional arrays, each containing 128 by 16 entries to store all the pre-computed exponents.

Memory required: The maximum memory required for each table is *8192 bytes* as outlined below.

$$\begin{aligned} ASCII\_TABLE\_SIZE &= 128 \\ ENCRYPTION\_TABLE\_ENTRY\_SIZE &= \lceil log_2((max(P) \times max(Q) - 1) \rceil = 2^0 + 2^1 + 2^2 + 2^3 + \ ... \ + 2^{15} \\ &= 16\ powers \ \rightarrow 16 \\ DECRYPTION\_TABLE\_ENTRY\_SIZE &= \lceil log_2(65,536) \rceil = 2^0 + 2^1 + 2^2 + 2^3 + \ ... \ + 2^{15} \\ &= 16\ powers \ \rightarrow 16 \end{aligned}$$

$$encrypt\_table[ASCII\_TABLE\_SIZE][ENCRYPTION\_TABLE\_ENTRY\_SIZE] = \{\{\}, ..., \{\}\}$$
$$decrypt\_table[ASCII\_TABLE\_SIZE][ENCRYPTION\_TABLE\_ENTRY\_SIZE] = \{\{\}, ..., \{\}\}$$

$$max(encrypt\_table\_size) = 16 \times 128 \times 32\ bits = 65536\ bits = 8192\ bytes$$
$$max(decrypt\_table\_size) = 16 \times 128 \times 32\ bits = 65536\ bits = 8192\ bytes$$

Calculation of Tables: The tables being $128 \times 16 = 2048$ entries in total made it impractical to calculate all the exponents by hand. Two approaches were evaluated to dynamically create these tables for us:

- Approach 1: Implement the MME method to create the tables on initialization
- Approach 2: Pre-compute the tables with a Python script utilizing the WolframAlpha API

Approach 2 was chosen since the tables would only have to be computed once and it reduced programmer error. The Python script used to interface with the API and generate the tables can be found in Appendix B.

R2: Use 16-bit integers.
Approach: Fulfilled by using the built-in C-type *uint16_t* to limit the executable to using 16-bit unsigned variables (where possible - see Issue 1).

R3: M = P*Q is a 16-bit integer.
Approach: Fulfilled by picking a P and Q such that $P \times Q$ is a 16-bit integer.

R4: Define a new instruction to activate a hardware assist which stores all tables of powers.
Approach: We started with designing a separate hardware unit (the Modulus and Multiplication Unit) that would calculate the modulus and multiplication operations done in the encryption and decryption functions. A new assembly instruction and opcode would then be used to replace the 39 assembly instructions that were required to compute each modulus and multiplication operation.

Code Structure

main.c

Contains the following functions that are necessary to perform the RSA cryptography.

```
encrypt(): Uses RSA encrypt a ASCII char using the table method
decrypt(): Uses RSA decrypt a ASCII char using the table method
find_index(): Used to find the table index needed to decrypt - O(n)
binarysearch_table(): Used to find the table index needed to decrypt - O(log(n))
hashFunctionSearch(): Used to find the table index needed to decrypt - O(1)
```

Figure 2: Functions of main.c

main.h

Contains all the constants and power tables needed for the encryption and decryption.

```
P = 167      // Prime #1
Q = 269      // Prime #2
M = 44923    // P*Q
E = 16969    // Coprime with (P-1)(Q-1) = 44488, 1 < E < 44923
D = 55785    // (DE-1) is evenly divisible by (P-1)(Q-1) = 44488
gen_table[997] = {...}
encrypt_table[ASCII_TABLE_SIZE][ENCRYPTION_TABLE_ENTRY_SIZE] = {...}
decrypt_table[ASCII_TABLE_SIZE][DECRYPTION_TABLE_ENTRY_SIZE] = {...}
```

Figure 3: Constants and tables of main.h

Solutions Evaluated

In the following section, the different solutions that were evaluated will be introduced. We explored five solutions in total, each building on the previous solution in order to deliver the optimal performance.

Solution 1: Naive Implementation

The naive implementation was the first attempt. It set the groundwork for the code and is an essential part of the developmental process. It relied on the code structure presented above and a simplistic linear search algorithm in the decrypt function to locate the index needed to decrypt the value.

```
int find_index(uint16_t c)
{
    uint16_t i;
    for(i = 0; i < 128; i++)
    {
        if(decrypt_table[i][0]==(int)c)
        {
            return i;
        }
    }
    return -1;
}
```

Figure 4: Linear search algorithm

Optimization Areas Discovered

Area 1: Decryption speed was significantly slower than encryption. Using GPROF, as seen in Figure 5, we determined that the cause of the reduction in speed was from the linear search that is run on the decryption table.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 60.62    334.57   334.57 128000000     0.00     0.00  find_index
 17.98    433.81    99.24 128000000     0.00     0.00  decrypt
  8.99    483.41    49.60 128000000     0.00     0.00  encrypt
```

Figure 5: GPROF summary for linear search

Solution 2: Binary Search Implementation

In Solution 1, the major bottleneck discovered was the linear search function *find_index()*. Being an $O(n)$ algorithm, this outcome was predicted. The decision was made to implement a binary search ($O(log(n))$) to improve decryption times and reduce time spent searching for indices.

```c
int binarySearch_table(const uint16_t arr[][16], uint16_t left, uint16_t right, uint16_t x)
{
    if (right >= left)
    {
        uint16_t mid = left + (right - left) / 2;
        if (arr[mid][0] == x)
        {
            return mid;
        }
        if (arr[mid][0] > x)
        {
            return binarySearch_table(arr, left, mid - 1, x);
        }
        return binarySearch_table(arr, mid + 1, right, x);
    }
    return -1;
}
```

Figure 6: Binary search algorithm

Optimization Areas Discovered

Area 1: Decryption speed was significantly slower than encryption. Using GPROF, as seen in Figure 7, we determined that the cause of the reduction in speed was from the binary search that is run on the decryption table.

Area 2: Computation of an exponential using the Table of Powers method takes a significant amount of temporary memory and modular instructions. We wish to reduce the amount of temporary variables and modular instructions if possible. One solution we investigated was using longs to reduce modular instructions alongside temporary variables but this may have caused greater performance issues.

Area 3: Fetching the tables from memory is the most frequent operation and, with the tables being so large (at 8192 bytes) and the D-cache having a size of 16 KB, we encounter many cache misses. We wish to investigate reducing the table sizes by using shorts where possible and improving efficiency through software pipelining.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 27.37    120.35   120.35                             mcount_internal
 27.20    239.98   119.63 128000000     0.00     0.00  binarySearch_table
 23.20    341.99   102.01 128000000     0.00     0.00  decrypt
 11.35    391.90    49.91 128000000     0.00     0.00  encrypt
```

Figure 7: GPROF summary for binary search

Solution 3: Hash table Implementation

This implementation builds off Solution 2; the major bottleneck discovered was in the binary search function, even after reducing it from a linear search. This solution investigates replacing the binary search, which is $O(log(n))$ with a hash table that is designed to provide an $O(1)$ time to find the index of values in the decrypt table. The hash chosen was a $C mod 997$, which resulted in 3 collisions. Instead of being resolved dynamically, the cache misses were resolved before that hash key was calculated. This greatly improved the decryption algorithm execution time. We investigated setting the memory locations of each value of C aside, but this wasn't possible due to the reserved memory locations.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 51.93    110.16   110.16 128000000     0.00     0.00  decrypt
 24.25    161.61    51.45 128000000     0.00     0.00  encrypt
```

Figure 8: GPROF summary for hash function utilization

Solution 4: Hash Table with Optimized C Code Assist Implementation

Building off of the adequate retrieval of the correct index for the decryption algorithm in Solution 3, the code was then optimized to reduce the instruction count and run-time length. These changes are outlined below:

Looking at the encryption table, it can be noticed that $encrypt\_table[T][0]$ is equivalent to $T^1 = T$. Therefore, a memory fetch can be avoided by simply using T.

```
//Before Optimizing
uint32_t a = encrypt_table[T][0] * encrypt_table[T][3];
//After
uint32_t a = T * encrypt_table[T][3];
```

Figure 9: Optimization of the first calculation in the encrypt function

A small change that reduced the number of LDR and STR operations on the temporary variables in encrypt and decrypt was the inclusion of the *register* keyword which hints at the compiler to not store the specified value in memory.

```
//Before
uint32_t b = a % 44923 * encrypt_table[T][6];
//After Optimizing
register uint32_t b = a % 44923 * encrypt_table[T][6];
```

Figure 10: Optimization of temporary variables

Two changes took place in the following section of code (Figure 11):

Firstly, a *goto* statement was added in order to reduce the number of instructions. If the value for C was greater than 25714, two extra *if* statements (consisting of 8 instructions each) would be executed. In total, this addition saved $47 \times 16 - (78 \times 4) = 440$ instructions over the entire ASCII set.

Secondly, removing the temporary variable *num* and integrating it into the array index reduced the variable count for this section.

```
//Before
   int index;
    if(i == 5777)
        index = 18;
    else if(i == 5182)
        index = 15;
    else if(i == 25714)
        index = 77;
    else{
        int num = (((int)i) % 997);
        index = gen_table[num];}
// After
    int index;
    if(C > 25714)
        goto skip;
    if(C == 5777)
        index = 18;
    else if(C == 5182)
        index = 15;
    else if(C == 25714)
        index = 77;
    else
     skip:index = gen_table[(((int)C) % 997)];
```

Figure 11: Optimization of the hash function logic

Solution 5: Hash Table with Hardware Assist Implementation

A major component of the execution time in Solution 4 was the calculation of the modulus and multiplication of the exponents. With the RISC architecture these actions take a total of 39 instructions. With a new hardware assist, 39 instructions can be reduced to 4, spanning 4 clock cycles: 2 to load the two registers into the Modulus and Multiplication Unit (MAMU), 1 to send the OpCode, and 1to load the result into the register. Theoretically, this would reduce the run-time greatly.

The interface of the MAMU with the rest of the system is as follows:

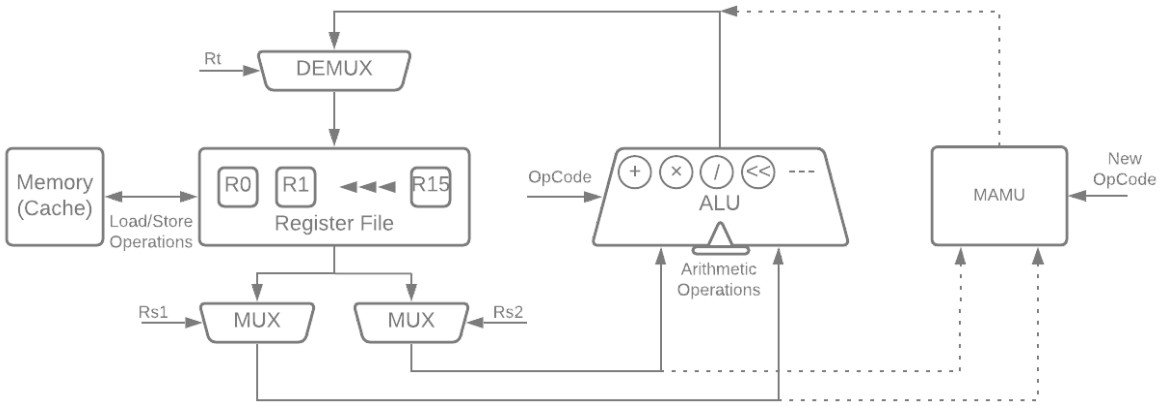

Figure 12: Modulus and Multiplication Unit (MAMU) integration

The MAMU consists of the following components:

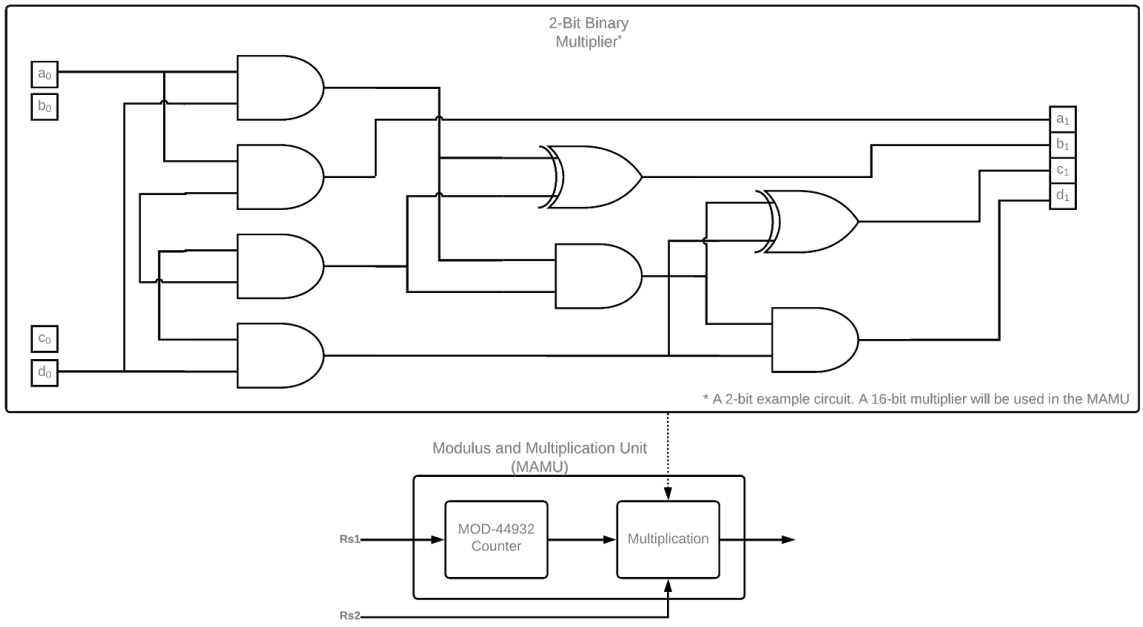

Figure 13: Optimization of temporary variables

A 2-bit multiplication is shown in Figure 13 for simplicity, but in actualization, all the modules would be implemented with 32-bit components.

Discussion

The performance of the application throughout the different stages of the optimization was measured using an in-house performance testing suite. The testing suite consists of running the encryption and decryption of 128 ASCII characters 10,000 times repeated over 100 trials to create a reasonable sample size. The ASCII characters were run in sequential order from 1 to 128 to simulate worst-case performance and spread out the cache hits. The results of the trials were output to a .txt file where they could be evaluated. The mean, median, maximum, and minimum times were analyzed throughout all stages of the optimization.

GPROF was another tool used to generate analytics of our algorithms. It was invented by Susan L. Graham and is a performance analysis tool for Unix applications [3]. It is capable of call graph collecting, which is useful when optimizing. It provides detailed analytics on where the program was bottlenecked, allowing areas of improvement to be easily identified.

Solution 1: Naive Implementation

This solution performed the worst out of all the solutions implemented. This was foreseen, but it served as a benchmark of the lowest performance if no optimizations were implemented. The results of the performance can be seen in Figure 12, where the average times for encryption and decryption of 1,280,000 ASCII characters were 0.8 seconds and 6.5284 seconds, respectively.

Solution 2: Binary Search Implementation

This solution saw significant gains from the naive implementation in Solution 1 and was the largest leap in performance improvement. The results of the performance are shown below in Figure 12, where the average times for encryption and decryption of 1,280,000 ASCII characters were 0.8411 seconds and 3.3559 seconds, respectively. Compared to the previous solution, there was little change in the encryption performance and a 195% increase in the decryption performance.

Solution 3: Hash Table Implementation

This solution built upon the underlying structure of Solution 3, with a hash table implementation to map ciphered values (C) with the correct column of the decryption table. The results of the performance are shown below in Figure 12, where the average times for encryption and decryption of 1,280,000 ASCII characters were 0.8424 seconds and 1.7275 seconds, respectively. Compared to the naive solution, there was little change in the encryption performance and a 378% increase in the decryption performance.

Solution 4: Hash Table with Optimized C code Assist Implementation

This solution was simply an optimization to improve the running time of Solution 3 and relied on using concepts taught in SENG 440 to make the code more efficient on the microprocessor. It included looking at the assembly code and finding areas for improvement through manual

inspection in order to further reduce the running time. The results of the performance are shown below in Figure 12, where the average times for encryption and decryption of 1,280,000 ASCII characters were 0.7778 seconds and 1.651 seconds, respectively. This translates to an 8% increase in the encryption performance, and a 5% increase in the decryption performance compared to the naive solution. This optimized solution also reduced the instruction count by 30 lines, even after adding the new *if* and *goto* statement.

Solution 5: Hash Table with Hardware Assist Implementation

Due to the limitations of working remotely and not having physical access to the hardware, this solution is strictly theoretical and the actual run-time may vary. Based on the previous run-times and the amount of time saved by reducing the instruction, a rough estimate time was calculated and would be classified as the ideal solution if it could be properly tested. The estimated times may be different than observed real-world times due to the following factors:

- Under/Over clocked CPU
- Thermal throttling
- Operating system overhead
- Other users on the ARM

The calculations are presented below.

Encrypt:

The encryption function takes 125 instructions to execute the series of modulus and multiplication operations necessary, with 4 unique sets of modulus and multiplication. Reducing the clock cycles necessary to compute those numbers from 125 to $(4 \times 4) = 16$ would result in roughly the following performance:

$$Number\ of\ encrypt\ instructions\ without\ a\ hardware\ assist\ = 154$$
$$Number\ of\ encrypt\ instructions\ with\ a\ hardware\ assist\ = 45$$
$$CLOCKS\_PER\_SEC\ = 400\ Mhz\ = 400,000,00\ hz$$
$$Number\ of\ ASCII\ characters\ = 128$$
$$Number\ of\ iterations\ of\ the\ ASCII\ set = 10,000$$

$$Estimated\ time\ of\ encrypt\ without\ a\ hardware\ assist\ = \frac{10,000 \times 128 \times 154}{400,000,000} = 0.4928\ seconds$$
$$Estimated\ time\ of\ encrypt\ with\ a\ hardware\ assist = \frac{10,000 \times 128 \times 45}{400,000,000} = 0.144\ seconds$$

Decrypt:

The decryption function takes 289 instructions to execute the series of modulus and multiplication operations necessary, with 10 unique sets of modulus and multiplication. Reducing the clock cycles necessary to compute those numbers from 289 to $(10 \times 4) = 40$ would result in roughly the following performance:

$Number\ of\ decrypt\ instructions\ without\ a\ hardware\ assist\ =\ 356$
$Number\ of\ decrypt\ instructions\ with\ a\ hardware\ assist\ = 107$
$CLOCKS\_PER\_SEC\ =\ 400\ Mhz\ =\ 400,000,00\ hz$
$Number\ of\ ASCII\ characters$ = 128
$Number\ of\ iterations\ of\ the\ ASCII\ set = 10,000$

$Estimated\ time\ of\ encrypt\ without\ a\ hardware\ assist\ = \frac{10,000\times128\times356}{400,000,000} = 1.1392\ seconds$
$Estimated\ time\ of\ encrypt\ with\ a\ hardware\ assist = \frac{10,000\times128\times107}{400,000,000} = 0.3424\ seconds$

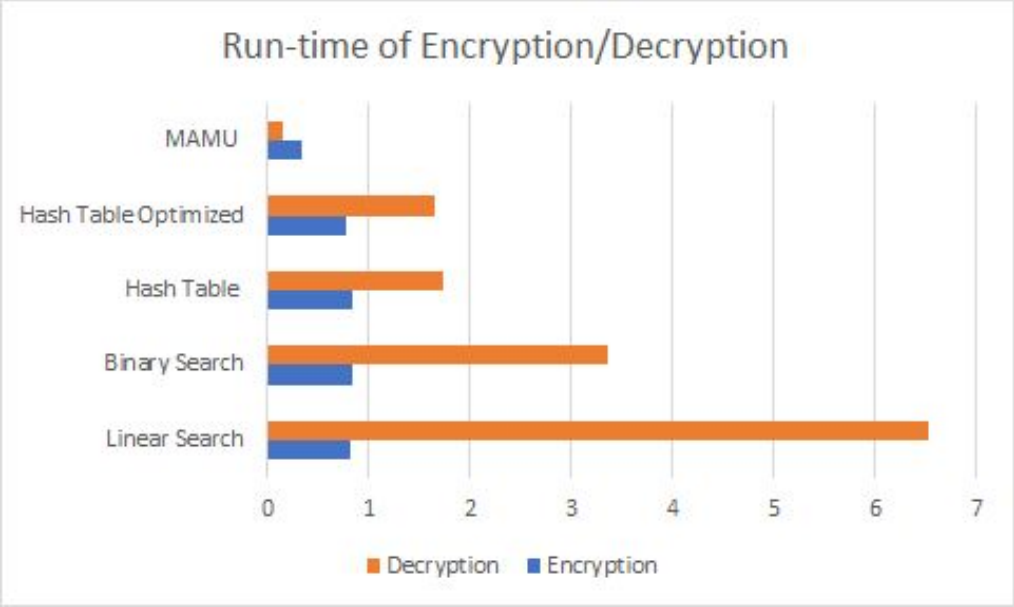

Figure 14: Comparison of the run-times of each solution

In summary, the encryption function could see up to a 342% increase in performance and decryption function could see up to a 332% increase in performance from using a hardware assist.

Issues Encountered

There were numerous issues encountered during the design and optimization process; the major ones are listed below:

Issue 1: Avoiding overflow on the exponential computation.
Solution: When computing exponentials, unsigned 32-bit integers were used to avoid any 16-bit number multiplied with each other exceeding 32 bits. The modular of every multiplication instruction was taken directly after to reduce it back to 16 bits maximum.

Issue 2: Efficiently computing modulus operations.

Solution: Enabling a hardware assist that is able to compute the modulus (and multiplication) as described in Solution 5 above. Conversely, after designing the system, an easier solution is to select a value M where bit masking could work, which would greatly reduce the time needed to compute the modulus operations..

Issue 3: Optimizing the assembly code.
Solution: Some optimizing of the assembly code didn't function as planned and resulted in an decrease in performance. The .s file was examined in order to locate the expensive instructions that we're being executed and the change was either reverted or fixed by manual inspection.

Cost Issues

An ever-present consideration we were presented with was the decision between sacrificing memory or run-speed to increase performance. They were both important aspects to keep balanced as a reduced memory size increases cache hits, which would lead to lower run-speeds. An example of this was the sacrifice of memory to implement a hash table large enough that it would encounter very few collisions; the size of the hash table required to do so ended up being 997. This table uses a significant amount of memory and leads to fewer cache hits from the encryption and decryption lookup tables, but it increases the run-time significantly; the hash function is $O(1)$ compared to the $O(log(n))$ binary search.

Major Limitations

The major limitation encountered was that the RISC instruction set has no efficient way to compute the modulus of numbers, which results in creating large overhead for the instruction. As this needs to be done extremely often for encryption and decryption, it presents an extremely large time increase for RSA encryption using the Tables of Powers method. No solution to fix this issue in software was ever encountered. The solution that was chosen, designed, and analyzed was entirely theoretical with no actualization or testing involved.

For this semester, the largest limitation was simply not having physical access to the lab or physical hardware. This made the solution difficult to actualize and led to a more theory based understanding of some of the concepts.

Conclusion

Using various techniques taught in SENG 440, the run-time of RSA encryption implemented using the Tables of Powers method was reduced significantly. The design process involved five iterations in total, each building off the strength of its predecessors and improving upon the bottlenecks. This required high level C code at first, but later required searching through assembly code and writing assembly optimizations into the software to further improve efficiency. The solution that utilized the least amount of software modification required the implementation of a piece of hardware, the Modulus and Multiplication Unit (MAMU), to improve performance. Solution 4 is the most optimized software-only solution that provided a 395% increase in performance from the naive solution. Solution 5 is the most optimized hardware-assisted solution that could provide a theoretical 4,534% increase in performance. Both hardware-assisted and software-only solution systems were investigated and each saw greatly increased performances. Unfortunately, the hardware-assisted solution is only theoretical and cannot be validated as the physical hardware was not constructed and couldn't account for the numerous variables at run-time. Therefore, the RSA encryption/decryption was increased by 395% using a software approach.

There are many future works that would be interesting to pursue and test. For Solution 5, future work would include actualizing the hardware mentioned and performing testing. After researching the topic and having completed the solution systems above, many new ideas and implementations could be valuable to pursue. Namely, the use of a cryptic co-processor to do specific tasks quickly. This is much like Solution 5, but is actually supported by ARMv4. Another idea that was pursued but was ultimately aborted due to the processor's physical limitations, was parallelization. This would significantly speed up performance by creating a buffer and computing two (or more) values at once. Ultimately, there are many future works that are worth pursuing, whether they include keeping the current hardware or software.

References

[1] R. Rivest, A. Shamir, L. Adleman, "*Cryptographic communications system and method,"* United States Patent 4405829, Dec. 14, 1977.

[2] J. K. Lim, "Algorithms Explained: RSA Encryption," Medium, 20-Feb-2019. [Online]. Available: https://medium.com/@jinkyulim96/algorithms-explained-rsa-encryption-9a37083aaa62. [Accessed: 20-Jul-2020].

[3] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. gprof: a Call Graph Execution Profiler // Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No 6, pp. 120-126; doi: 10.1145/800230.806987

Appendix A: Software Source Code

main.c

```
/* ------------------------------------------------------------------------ */
/*                          RSA Encryption/Decryption                       */
/*                            SENG 440 Summer 2020                          */
/* ------------------------------------------------------------------------ */
/*
  RSA Cryptography Algorithm

   Public  key-piar(E, PQ), used to encrypt data
   Private key-pair(D, PQ), used to decrypt data

   1. Find P and Q, two large prime numbers

   2. Choose E such that E > 1, E < PQ and
      E and (P-1)(Q-1) have no prime factors in common
      E doesn't have to prime but it must be odd

   3. Compute D such that (DE-1) is evenly divisible by (P-1)(Q-1)
      D = (X(P-1)(Q-1)+1)/E ,find an int X that causes D to be an int

   4. Encryption function is the following
      C = T^E mod PQ where C=ciphertext and T=Plain text

   5. decryption function is the following
      T = C^D mod PQ where C=ciphertext and T=Plain text

  List of the first 60 primes
     1     2     3     4     5     6     7     8     9    10
  ---------------------------------------------------------
     2,    3,    5,    7,   11,   13,   17,   19,   23,   29,  // 1  - 10
    31,   37,   41,   43,   47,   53,   59,   61,   67,   71,  // 11 - 20
    73,   79,   83,   89,   97,  101,  103,  107,  109,  113,  // 21 - 30
    27,  131,  137,  139,  149,  151,  157,  163,  167,  173,  // 31 - 40
   179,  181,  191,  193,  197,  199,  211,  223,  227,  229,  // 41 - 50
   233,  239,  241,  251,  257,  263,  269,  271,  277,  281   // 51 - 60
```

```
ASCII CHAR to DEC
Dec  Char                           Dec  Char     Dec  Char     Dec  Char
---------                           ---------     ---------     ----------
  0  NUL (null)                      32  SPACE     64  @         96  `
  1  SOH (start of heading)          33  !         65  A         97  a
  2  STX (start of text)             34  "         66  B         98  b
  3  ETX (end of text)               35  #         67  C         99  c
  4  EOT (end of transmission)       36  $         68  D        100  d
  5  ENQ (enquiry)                   37  %         69  E        101  e
  6  ACK (acknowledge)               38  &         70  F        102  f
  7  BEL (bell)                      39  '         71  G        103  g
  8  BS  (backspace)                 40  (         72  H        104  h
  9  TAB (horizontal tab)            41  )         73  I        105  i
 10  LF  (NL line feed, new line)    42  *         74  J        106  j
 11  VT  (vertical tab)              43  +         75  K        107  k
 12  FF  (NP form feed, new page)    44  ,         76  L        108  l
 13  CR  (carriage return)           45  -         77  M        109  m
 14  SO  (shift out)                 46  .         78  N        110  n
 15  SI  (shift in)                  47  /         79  O        111  o
 16  DLE (data link escape)          48  0         80  P        112  p
 17  DC1 (device control 1)          49  1         81  Q        113  q
 18  DC2 (device control 2)          50  2         82  R        114  r
 19  DC3 (device control 3)          51  3         83  S        115  s
 20  DC4 (device control 4)          52  4         84  T        116  t
 21  NAK (negative acknowledge)      53  5         85  U        117  u
 22  SYN (synchronous idle)          54  6         86  V        118  v
 23  ETB (end of trans. block)       55  7         87  W        119  w
 24  CAN (cancel)                    56  8         88  X        120  x
 25  EM  (end of medium)             57  9         89  Y        121  y
 26  SUB (substitute)                58  :         90  Z        122  z
 27  ESC (escape)                    59  ;         91  [        123  {
 28  FS  (file separator)            60  <         92  \        124  |
 29  GS  (group separator)           61  =         93  ]        125  }
 30  RS  (record separator)          62  >         94  ^        126  ~
 31  US  (unit separator)            63  ?         95  _        127  DEL
*/
```

```
/* ------------------------------ Includes ------------------------------- */

#include <stdio.h>
#include <math.h>
#include <stdint.h>
#include "main.h"
#include "test.h"

/* ------------------------------ Functions ------------------------------ */

/* !!THIS FUNCTION WAS REPLACED BY binarySearch_table()!!
 *
 * int find_index(uint16_t c)
 * Inputs:  uint16_t C, where C is the value to be searched for
 * Outputs: int i,      where i is arrary index of the value
 *
 * Locates the index of decrypt_table[i][0] needed to compuet the power
 *
 * Time complexity: O(n), where n=i
 */
/*
int find_index(uint16_t c)
{
    uint16_t i;
    for(i = 0; i < 128; i++)
    {
        if(decrypt_table[i][0] == (int)c)
        {
            return i;
        }
    }
    return -1;
}

*/
/* int binarySearch_table(arr[][16], left, right, x)
```

```
 * Inputs: - const uint16_t decrypt_table[][]
           - uint16_t left , where left is the lefthand boundary for the search
           - uint16_t right, where right is the righthand boundary for the search
           - uint16_t x,     where x is the value to be searched for
 * Outputs: int mid,         where mid = array index of the value
 *
 * Locates the index of decrypt_table[i][0] needed to compute the power
 *
 * Time complexity: O(log(n)), where n=i
 */
/*
int binarySearch_table(const uint16_t arr[][16],
                       uint16_t left,
                       uint16_t right,
                       uint16_t x)
{
    if(right >= left)
    {
        uint16_t mid = left + (right - left) / 2;
        if(arr[mid][0] == x)
        {
            return mid;
        }
        if(arr[mid][0] > x)
        {
            return binarySearch_table(arr, left, mid - 1, x);
        }
        return binarySearch_table(arr, mid + 1, right, x);
    }
    return -1;
}
*/
/* int hashFunctionSearch(uint16_t i)
 * Inputs:  uint16_t i, where i is the value to be searched for
 * Outputs: int index,  where index is the array index of the value
 *
 * Locates the index of decrypt_table[i][0] needed to compute the power
```

```
 * by utilizing a hash function
 *
 * Time complexity: O(1)
 */
/*
int hashFunctionSearch(uint16_t i)
{
    int index;
    if(i == 5777)
    {
        index = 18;
    }
    else if(i == 5182)
    {
        index = 15;
    }
    else if(i == 25714)
    {
        index = 77;
    }
    else
    {
        int num = (((int)i) % 997);
        index = gen_table[num];
    }
    return index;
}

*/
/* uint16_t encrypt(uint16_t T)
 * Inputs:  uint16_t T, Where T = Plaintext
 * Outputs: uint16_t C, Where C = Ciphertext
 *
 * Computes encryption of of T using the RSA formula below
 * and table method for computing exponents.
 *
 * C = T^E mod PQ where C=ciphertext and T=Plain text
```

```
 *
 * E = 16969 = 0100 0010 0100 1001
 * E = 2^0 + 2^3 + 2^6 + 2^6 + 2^9 + 2^14
 */
uint16_t encrypt(uint16_t T)
{
    register uint32_t a = T     * encrypt_table[T][3];
    register uint32_t b = a % 44923 * encrypt_table[T][6];
    register uint32_t c = b % 44923 * encrypt_table[T][9];
    register uint32_t d = c % 44923 * encrypt_table[T][14];
    register uint32_t C = d % 44923;
    return (uint16_t)C;
}


/* uint16_t decrypt(uint16_t C)
 * Inputs:  uint16_t C, Where C = Ciphertext
 * Outputs: uint16_t T, Where T = Plaintext
 *
 * Computes decryption of of C using the RSA formula below
 * and table method for computing expoents.
 *
 * T = C^D mod PQ where C=ciphertext and T=Plain text
 *
 * D = 55785 = 1101 1001 1110 1001
 * D = 2^0 + 2^3 + 2^5 + 2^6 + 2^7 + 2^8 + 2^11 + 2^12 + 2^14 + 2^15
 */
uint16_t decrypt(uint16_t C)
{
    int index;
    if(C > 25714)
    {
        goto skip;
    }
    if(C == 5777)
    {
        index = 18;
```

```
    }
    else if(C == 5182)
    {
        index = 15;
    }
    else if(C == 25714)
    {
        index = 77;
    }
    else
    {
     skip:index = gen_table[(((int)C) % 997)];
    }
    register uint32_t a = decrypt_table[index][3] * decrypt_table[index][5];
    register uint32_t b = a % 44923 * decrypt_table[index][6];
    register uint32_t c = b % 44923 * decrypt_table[index][7];
    register uint32_t d = c % 44923 * decrypt_table[index][8];
    register uint32_t e = d % 44923 * decrypt_table[index][11];
    register uint32_t f = e % 44923 * decrypt_table[index][12];
    register uint32_t g = f % 44923 * decrypt_table[index][14];
    register uint32_t h = g % 44923 * decrypt_table[index][15];
    register uint32_t i = h % 44923 * decrypt_table[index][0];
    register uint32_t T = i % 44923;
    return (uint16_t)T;
}


/* int main()
 *
 */
int main(void)
{
    // user_input();
    // test();
    test_performance_CVS();
    return 0;
}
```

main.h

```
#ifndef __MAIN_H__
#define __MAIN_H__

/* ------------------------------- Defines ------------------------------- */

#define ASCII_TABLE_SIZE            128
#define ENCRYPTION_TABLE_ENTRY_SIZE 15
#define DECRYPTION_TABLE_ENTRY_SIZE 16

/* ------------------------------ Constants ------------------------------ */

static const uint16_t P = 167;   // Prime #1
static const uint16_t Q = 269;   // Prime #2
static const uint16_t M = 44923; // P*Q
static const uint16_t E = 16969; // Coprime with (P-1)(Q-1) = 44488, 1<E<44923
                                 //  0100 0010 0100 1001
                                 //   = 1 + 8 + 64 + 512 + 16384
static const uint16_t D = 55785; // (DE-1) is evenly divisible by (P-1)(Q-1) =
                                 // 44488
                                 //  1101 1001 1110 1001
                                 //  = 1 + 8 + 32 + 64 + 128 + 256 + 2048 + 4096
                                 //    + 16384 + 32768

static const int gen_table[997] = {0,   1,   73,  18,  15,  77,  0,   0,  ... };

static const uint16_t
encrypt_table[ASCII_TABLE_SIZE][ENCRYPTION_TABLE_ENTRY_SIZE] = {
  {0,   0,     0,     0,     0,     0,     ... },
  {1,   1,     1,     1,     1,     1,     ... },
  {2,   4,     16,    256,   20613, 14035, ... },
  {3,   9,     81,    6561,  10487, 5665,  ... },
  {4,   16,    256,   20613, 14035, 38793, ... },
  {5,   25,    625,   31241, 2983,  3535,  ... },
  {6,   36,    1296,  17465, 43978, 39488, ... },
   ...
  {127, 16129, 40471, 9261,  8114,  24801, ... }};
```

```
static const uint16_t
decrypt_table[ASCII_TABLE_SIZE][DECRYPTION_TABLE_ENTRY_SIZE] = {
  {0,     0,     0,     0,     0,     0,     … },
  {1,     1,     1,     1,     1,     1,     … },
  {206,   42436, 30718, 32832, 12839, 17434, … },
  {313,   8123,  36165, 19003, 22935, 10818, … },
   …
  {44769, 23716, 12696, 4692,  2594,  35309, … }};

/* ------------------------- Function Prototypes ---------------------------- */

int find_index(uint16_t);
int binarySearch_table(const uint16_t [][DECRYPTION_TABLE_ENTRY_SIZE],
                       uint16_t,
                       uint16_t,
                       uint16_t);
int hashFunctionSearch(uint16_t);
uint16_t encrypt(uint16_t);
uint16_t decrypt(uint16_t);

#endif // __MAIN_H__
```

test.c

```c
/* ------------------------------ Includes ------------------------------ */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdint.h>
#include <time.h>
#include "main.h"
#include "test.h"

/* ------------------------------ Functions ------------------------------ */

void print_array(uint16_t input[])
{
    int i = 0;
    while(input[i] != '\0')
    {
        printf("%-7d ",input[i]);
        i++;
    }
}

void user_input()
{
    char message[100];
    uint16_t encrypted_message[100];
    uint16_t decrypted_message[100];
    printf("Input a message: ");
    scanf("%s", message);
    int i = 0;
    printf("%-15s", "Sent");
    while(1)
    {
        printf("%-7c ", (int)message[i]);
        encrypted_message[i] = encrypt((uint16_t)message[i]);
        i++;
```

```
        if(message[i] == '\0')
        {
            encrypted_message[i] = '\0';
            break;
        }
    }

    i = 0;
    printf("\n%-15s", "ASCII Dec");
    while(1)
    {
        printf("%-7d ", (int)message[i]);
        encrypted_message[i] = encrypt((uint16_t)message[i]);
        i++;
        if(message[i] == '\0')
        {
            encrypted_message[i] = '\0';
            break;
        }
    }

    printf("\n%-15s","Encrypted");
    print_array(encrypted_message);
    printf("\n%-15s","Decrypted");

    i = 0;
    while(1)
    {
        decrypted_message[i] = decrypt(encrypted_message[i]);
        i++;
        if(encrypted_message[i] == '\0')
        {
            decrypted_message[i] = '\0';
            break;
        }
    }
```

```
    print_array(decrypted_message);
    printf("\n");
    i = 0;
    printf("%-15s", "Received");
    while(1)
    {
        printf("%-7c ", decrypted_message[i]);
        encrypted_message[i] = encrypt((uint16_t)message[i]);
        i++;
        if(message[i] == '\0')
        {
            encrypted_message[i] = '\0';
            break;
        }
    }
}


/* void test_ASCII()
 *
 *
 *
 */
int test_ASCII()
{

printf("%-9s|%-9s|%-9s|%-9s|%-9s\n","Char","Orginial","Encrypted","Decrypted","PA
SS/FAIL");
    printf("---------+---------+---------+---------+---------\n");
    uint16_t encrypted = 0;
    uint16_t decrypted = 0;
    int cases_passed = 0;
    int i;
    for(i = 0; i < 128; i++)
    {
        encrypted = encrypt((uint16_t)i);
        decrypted = decrypt(encrypted);
```

```
        if(i > 31 && i < 127)
        {
            printf("%-9c|", i);
            printf("%-9d|", i);
            printf("%-9d|", encrypted);
            printf("%-9d|", decrypted);
            if(i == (int)decrypted)
            {
                printf("%-9s|", "PASS");
                cases_passed = cases_passed + 1;
            }
            else
            {
                printf("%-9s|", "FAIL");
            }
            printf("\n");
        }
        else
        {
            if(i == (int)decrypted)
            {
                cases_passed = cases_passed + 1;
            }
        }
    }
    if(cases_passed == 128)
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

/* int test_sanity()
 *
```

```
 *
 *
 */
int test_sanity()
{
    uint16_t encrypted = encrypt(33);
    uint16_t decrypted = decrypt(encrypted);

    if(33 == (int)decrypted)
    {
        return 1;
    }
    else
    {
        return -1;
    }
}


/* int test_encrpytion_performance()
 *
 *
 *
 */
double test_encrpytion_performance()
{
    clock_t start = clock();
    int i, j;
    for(i = 0; i < 10000; i++)
    {
        for(j = 0; j < 128; j++)
        {
            uint16_t encrypted = encrypt((uint16_t)j);
            if(expected_encrypted_ASCII[j] != (int)encrypted)
            {
                return -1;
            }
```

```
        }
    }
    clock_t diff = clock() - start;
    double time_taken = ((double)diff) / CLOCKS_PER_SEC; // in seconds
    return time_taken;
}

/* int test_encrpytion_performance()
 *
 *
 *
 */
double test_decrpytion_performance()
{
    clock_t start = clock();
    int i, j;
    for(i = 0; i < 10000; i++)
    {
        for(j = 0;j<128;j++)
        {
            uint16_t decrypted = decrypt((uint16_t)expected_encrypted_ASCII[j]);
            if(j != (int)decrypted)
            {
                return -1;
            }
        }
    }
    clock_t diff = clock() - start;
    double time_taken = ((double)diff) / CLOCKS_PER_SEC; // in seconds
    return time_taken;
}

/* void test()
 *
 *
 *
 */
```

```
void test()
{
    int ASCII_result = test_ASCII();
    int santiy_result = test_sanity();
    double encryption_performance = test_encrpytion_performance();
    double decryption_performance = test_decrpytion_performance();
    printf("+---------------+----------+----------+\n");
    printf("|%-15s|%-10s|%-10s|\n","Test","PASS/FAIL","Time");
    printf("|---------------+----------+----------| \n");
    if(ASCII_result)
    {
        printf("|%-15s|%-10s|%-10s|\n","ASCII Test","PASS","N/A");
    }
    else
    {
        printf("|%-15s|%-10s|%-10s|\n","ASCII Test","FAIL","N/A");
    }

    printf("|---------------+----------+----------| \n");

    if(santiy_result)
    {
        printf("|%-15s|%-10s|%-10s|\n","Sanity Test","PASS","N/A");
    }
    else
    {
        printf("|%-15s|%-10s|%-10s|\n","Sanity Test","FAIL","N/A");
    }

    printf("|---------------+----------+----------| \n");

    if((int)encryption_performance != -1)
    {
        printf("|%-15s|%-10s|%-10f|\n","Encryption
Perf","PASS",encryption_performance);
    }
    else
```

```
    {
        printf("|%-15s|%-10s|%-10s|\n","Encryption Perf","FAIL","N/A");
    }

    printf("|---------------+----------+----------| \n");

    if((int)decryption_performance != -1)
    {
        printf("|%-15s|%-10s|%-10f|\n","Decryption
Perf","PASS",decryption_performance);
    }
    else
    {
        printf("|%-15s|%-10s|%-10s|\n","Decryption Perf","FAIL","N/A");
    }

    printf("+---------------+----------+----------+ \n");
}

/* void test()
 *
 *
 *
 */
void test_performance_CVS()
{
    time_t t = time(NULL);
    struct tm tm = *localtime(&t);
    // Allocates storage
    char *file_name = (char*)malloc(100 * sizeof(char));
    // Prints "Hello world!" on hello_world
    sprintf(file_name, "performance_results_%d-%02d-%02d %02d:%02d:%02d.txt",
tm.tm_year + 1900, tm.tm_mon + 1, tm.tm_mday, tm.tm_hour, tm.tm_min, tm.tm_sec);
    FILE * fp;
    fp = fopen (file_name, "w");
    fprintf(fp, "%-10s %-10s\n", "Encryption", "Decryption");
    double encryption_performance = 0.0;
```

```
    double decryption_performance = 0.0;
    int i;
    for(i = 0; i < 100; i++)
    {
        encryption_performance = test_encrpytion_performance();
        decryption_performance = test_decrpytion_performance();
        fprintf(fp, "%-10f %-10f\n",encryption_performance,decryption_performance
);
    }
    fclose(fp);
}
```

test.h

```
#ifndef TEST_H
#define TEST_H

/* --------------------------- Function Prototypes --------------------------- */

void print_array(uint16_t[]);
void user_input();
int test_ASCII();
int test_sanity();
void test();
void test_performance_CVS();
double test_encrpytion_performance();
double test_decrpytion_performance();

static const int expected_encrypted_ASCII[ASCII_TABLE_SIZE] = {
  0,     1,     41139, 33080, 33142, 24541, ...,
  24814, 34599, 33550, 34324, 6907,  718,   ...,
  37817, 23909, 27929, 41292, 44121, 10878, ...,
  14064, 40863, 19847, 31649, 11666, 6633,  ...,
  ...
  1185,  2576,  40461, 1694,  3922,  22738, ... };
#endif // __TEST_H__
```

Appendix B: WolframAlpha API Python Code

```
import wolframalpha
client = wolframalpha.Client('demo')

# Generated by encrypting the ASCII table
encrytped_ascii = [0, 1, 41139, 33080, 33142, 24541, 25681, 5777, 15688, 7043, 37620,
                   18873, 36468, 38146, 17333, 12747, 24814, 34599, 33550, 34324, 6907,
                   718, 12138, 9182, 8544, 22943, 38058, 11762, 44431, 43977, 12654,
                   10320, 37817, 23909, 27929, 41292, 44121, 10878, 35300, 27533, 9098,
                   44769, 23391, 11936, 26037, 23482, 25714, 38125, 14064, 40863, 19847,
                   31649, 11666, 6633, 11285, 6163, 19885, 9095, 30747, 206, 5182, 10695,
                   32130, 32096, 25150, 35512, 3266, 313, 20483, 16157, 38189, 38195, 24927,
                   12404, 32039, 25278, 25802, 1200, 36488, 5364, 29109, 8857, 43660, 17242,
                   31689, 4436, 26714, 17651, 37054, 18789, 1806, 22127, 1442, 15723, 27676,
                   39034, 15579, 10909, 44297, 40305, 10208, 7923, 4902, 10421, 15165, 10622,
                   12685, 32727, 19333, 29540, 39168, 11010, 1185, 2576, 40461, 1694, 3922,
                   22738, 29110, 15996, 22663, 40585, 5743, 26902, 26641, 24204, 20528, 35992]


def generate_encrypt_table():
  # a^(2^b) modulo 44923, a: 0 - 127, b: 0 - 14
  for a in range(0, 128):
      print('{', end='')
      for b in range(0, 15):
          query = str(a) + '^(2^' + str(b) + ') modulo 44923'
          res = client.query(query)
          print(next(res.results).text, end='')
          if(b != 15):
              print(', ', end='')
      print('}')

def generate_decrypt_table():
  # a^(2^b) modulo 44923, a: 0 - 127, b: 0 - 14
  for a in encrytped_ascii:
      print('{', end='')
      for b in range(0, 16):
          query = str(a) + '^(2^' + str(b) + ') modulo 44923'
          res = client.query(query)
          print(next(res.results).text, end='')
          if(b != 15):
              print(', ', end='')
      print('}')

generate_decrypt_table()
```