SECURE COMMUNICATION: AN RSA IMPLEMENTATION ON C, FOR MSP430 WITH KEY SIZE OF 64-BIT

by

"İbrahim Bali"

Engineering Project Report

Faculty of Engineering

Department of Electrical and Electronics Engineering

Istanbul, *2017*

SECURE COMMUNICATION: AN RSA IMPLEMENTATION ON C, FOR MSP430 WITH KEY SIZE OF 64-BIT

APPROVED BY:

Name of supervisor (Supervisor) .....

Name of Jury .....

Name of Jury .....

Date of Approval:

ACKNOWLEDGEMENTS

I would like to thank my supervisor Engin Maşazade, Assistant Professor for his encouragement and guidance through the course of this study.

I would like to thank to wolframalpha.com for without it, it would have been hard to verify the arithmetic operations' correctness.

I would also like to thank stackoverflow.com, Internet community as a whole because without such entity being existed, every piece of information I used in this project would have been much more difficult to acquire.

I would like to thank, David Eisenstat from stackoverflow.com for without his help the already very limited key size would have been halved.

I would like to thank graduate students Veysel Yaman Akgün, for providing the msp430g2553 mediums; and Volkan Talha Doğukan, for providing the information and the work for the communication part for demonstrative and testing purposes.

I would like to thank undergraduate student Buğra Altuğ for reviewing, criticizing and pointing out some optimization suggestions in the implemented C code.

ABSTRACT

We're living in the information age, and thus the information security is very important for without it, one could launch nuclear weapons, decide wars or even start one. Its failure had given the upper hand before, in World War II. Information security however is especially of uttermost importance in a country that jails journalists on a world record basis, meaning that one could get jailed by the mere opinions on what's going on; and investigate social media and internet users without court orders.

The problem with communication and data storage is that they don't by themselves hide information that they carry, but rather, apart from non-security related protocols, they store or send the information as it is. This makes the information available to anyone with certain backgrounds, whereas in most cases it needs to be secret in between certain parties.

In this study a set of security procedures called RSA was implemented on C, to be specifically used in msp430g2553. It however, as it can be used for both communication and storage; was mostly written in a modules basis, as in it can be re-written for other platforms with small differences. Which also means that this study focuses the cryptography of a large system, and cryptography alone, communication and storage modules needs to be studied separately.

After the C implementation was written, field tests with a Wireless Sensor Network application were made, compatibility issues were fixed and other minor changes were made in the C code of the implementation. It was then observed that the data was transmitted the way it was expected to have been. Data was first encrypted with RSA, sent, and got received by the other end, and then finally decrypted to have the raw data that was sent.

Now obviously this study doesn't protect journalists from being jailed for commenting on the matters, but approaches like this made the internet, or our other daily life environment a little more secure, and private.

ÖZET

Bilgi çağında yaşamakta olduğumuz için bilgi güvenliği çok önemlidir çünkü eğer olmasaydı herhangi bir insan nükleer füzelerle bir yere saldırabilir, savaşın gidişatını değiştirebilir veya hatta bir savaş bile başlatabilirdi. Üstelik daha önce böyle bir şey yaşandı, bilgi güvenliği yitirilince İkinci Dünya Savaşı'nın seyri değişti. Bilgi güvenliği hele de gazetecilerini dünya rekorları düzeyinde hapseden, ki gazetecilerin hapsedilmesi demek gündemle ilgili fikir beyan etmenin hapsedilmenize sebep olabileceği anlamına gelir; veya sosyal medya veya internet kullanıcılarıyla ilgili mahkeme kararı olmaksızın soruşturma açan bir ülkede belki de en önemli şeylerden bridir.

Komünikasyon ve bilgi depolama sistemleri taşıdıkları bilgileri, güvenlikle alakasız protokoller haricinde, gizlemezler. Bu da bu bilgilerin, belirli bir bilgi birikime sahip herkesçe erişilebilir olmasına neden olur. Oysa bir çok durumda istenen, bunun tersine, bilgilere yalnızca belirli bir kesimin erişebilmesidir.

Bu çalışmada RSA adı verilen bir dizi güvenlik prosedürü C dilinde, ve msp430g2553 için gerçekleştirildi. Her ne kadar belirli bir platform için yazılmışsa da, tıpkı hem iletişim amaçlı hem de veri depolama amacıyla kullanılabileceği gibi, genel olarak modüler bir kavrayışla yazıldığından, başka bir platformda da oldukça benzer bir kodla aynı prosedürler gerçekleştirilebilir. Yine modüler kavrayıştan dolayı, bu çalışma yalnızca kriptografi üzerinedir ve iletişim ve depolama modülleri ayrıca çalışılmalıdır.

C kodları yazıldıktan sonra bir kablosuz haberleşme uygulamasında denendi, ve koddaki uyumluluk sorunları giderildi ve bazı küçük değişiklikler yapıldı. Daha sonra verilerin beklendiği gibi iletildiği gözlemlendi.Veriler önce RSA ile şifrelendi, sonra yollandı, karşı taraf bu verileri alıp deşifreledi, ve en nihayetinde ilk yollanan verilere erişildi.

Şurası kesin ki, bu çalışma gazetecilerin bazı konular hakkında yorum yaptıkları için hapsedilmelerini engellemiyor. Ama bu çalışmaya benzer yaklaşımlar, interneti veya günlük yaşantımızdaki diğer ortamları birazcık daha güvenli ve mahremiyetlerine uygun hale getirdi.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS / ABBREVIATIONS

MAC	Message Authentication Code
gcd	Greatest Common Divisor
RNG	Random Number Generation

1. INTRODUCTION

The purpose of this document is to explain a secure communication implementation on programming language C, and for micro-controller MSP430G2553S using RSA algorithm.

This document consists of three parts. In the first part the basic terminology, definition cryptography is given.

In the second part the definition of RSA is given and the proof that it works is shown.

Finally in the third part, how the RSA algorithm is implemented in C, and how MSP430G2553S' specific conditions affected the implementation is explained.

2. SECURE COMMUNICATION

To communicate securely means that the communication between the desired parties can not be eavesdropped by an undesired party, and one of the types of security to achieve that is encrypting the “message” that is being used to communicate. This study focuses on Encryption type of security.

2.1. CRYPTOGRAPHY AND ENCRYPTION

“Cryptography is the art and science of encryption.”[1]Its fundamental purpose is to hide information from everyone except for certain allowed parties in certain points in time. In a way it is to hide information in plain sight. Roughly, it is to “write” in such a “secret” way that only the individuals with specific information, would understand what's written. This process is called encryption.

2.1.1. TERMINOLOGY

In order to easily understand further cryptosystems like RSA it is good to set some definitions that will be used on a regular basis. Some of those are:

- **message,** is the information that is to be stored or transmitted securely
- **channel,** refers to the medium where transmissions are being carried out
- **plaintext,** refers to the message that is either just decrypted or yet to be encrypted
- **ciphertext,** refers to the encrypted plaintext
- **to encrypt,** is to use an encryption function on plaintext in order to make the plaintext look meaningless
- **to decrypt,** is to obtain plaintext by using a corresponding decryption function on ciphertext

- **Alice and Bob,** usually refer to the two parties that want to communicate securely
- **Eve,** usually refers to a third party that wants to disrupt and/or eavesdrop the communication at hand
- **key,** refers to the specific information that must be known in order to either encrypt or decrypt, or do both

2.2. AUTHENTICATION

Authentication is the process to ensure that the message received is genuinely, from whom it claimed to have came from; from the time it was sent at; in the order that it claimed to be. In other words authentication checks whether the message is genuine or not. After all encrypting a message doesn't necessarily mean that the message came from Bob or Alice, and sent on time it claimed to have or it is in the claimed order. Eve can disrupt the message in above mentioned ways without knowing the encryption. She can disrupt the message by manipulating the channel alone unless certain precautions are made.

2.3. PUBLIC-KEY ENCRYPTION

Public-key encryption usually refers to a cryptographic system that has a pair of keys, public-key and secret-key. It can also be referred to as asymmetric key encryption. The public-key is used to encrypt the message where secret-key is used to decrypt the encrypted message, hence the name asymmetric-key. The public-key is made public, and assumed to be known by everyone, including Eve, Alice and Bob. Whereas secret-key is known only to Alice or Bob, as in whomever that will decrypt the encrypted message. The security is based on the secrecy of the secret-key alone.

Assume that Alice wants to send Bob a secret message, she first needs Bob to publish his public-key. She then encrypts the message using Bob's public-key and when Bob receives the ciphertext, he decrypts it using his secret-key, and reach the message. Eve can not decrypt the ciphertext since she doesn't know the secret-key. She can however send encrypted message

to Bob just like Alice did, and this is where Bob needs to rule Eve's message out using the authentication process.

RSA is a public-key algorithm and can be used as both for encrypting and digital signatures, but also mostly for key exchange.

Most classic ciphers are private-key or symmetric key encryption. AES, DES, Twofish, Caesar, Vigenére are some examples to secret-key encryption. In these scenarios Alice and Bob shares the same key, and the key is secret. The security of the message is based on the fact that Eve doesn't know what the key is.

3. RSA

RSA is a public-key cryptosystem, and can be used both on digital signatures and encryption. It's based on the difficulty of divison, and factorization of a number.[1]

There is a very smart analogy about how RSA works. Imagine that Bob wants to send a secret message to Alice. For this to happen Alice sends everyone a box and an unlocked lock. Bob then puts his message in the box and locks the lock, and sends it back to Alice. Alice then opens the box Bob sent, with her key. Eve can't learn Bob's message because only Alice has the key that opens her lock. Eve, too, however can send a secret message to Alice just like Bob did.

3.1. RSA ENCRYPTION

In order to encrypt a message *m,* with RSA, one needs to know the public-key pair (*e, n*) which Bob published. She then calculates $c :\equiv m^e \bmod n$, then sends *c* to Bob for him to decrypt. For decrypting the ciphertext Bob needs to calculate $m :\equiv c^d \bmod n$ with decryption key *d* that only he knows.

In order to find the public-key pair (*e, n*), Bob either; first generates two randomly selected primes *p* and *q* that aren't equal to one another then calculate $n := pq$ and then he picks an encryption exponent *e,* such that $gcd(e, (p-1)(q-1)) = 1;$ or he could do the vice versa, as in he could first pick *e,* and then generates primes in the form that is limited by *e.*

"Given the publicly known information *n* and *e,* it is easy to compute $m^e \bmod n$ from *m,* but not the other way around. However, if you know the factorization of *n,* then it is easy to do the inverse computation."[1]

Finding the decryption key *d,* is computationally much harder. *d* is the exponential inverse of encryption key *e* in mod *n,* as in that $m^{ed} \equiv m \bmod n$. In order to find *d* Euler's Theorem in a special form $ed \equiv 1 \bmod (p-1)(q-1)$ and Extended Euclidean Algorithm can be used.

3.2. RSA PROOF OF CORRECTNESS

It needs to be proven that $m^{ed} \equiv m \bmod (n)$, where $n = pq$ and e and d are positive integers that satisfy $ed \equiv 1 \bmod (p-1)(q-1)$. Since e and d are positive it can be written that $ed = 1 + k(p-1)(q-1)$ for some non-negative integer k. Assuming that m is relatively prime to n, as in $\gcd(m, n) = 1$ then it can be written that

$$m^{ed} = m^{1+k(p-1)(q-1)} = m(m^{(p-1)(q-1)})^k \equiv m(1)^k \equiv m \bmod n$$

3.2.1. RSA ALGORITHM

1. Bob chooses secret primes p and q and computes $n = pq$.
2. Bob chooses e with $gcd(e, (p-1)(q-1)) = 1$.
3. Bob computes d with $de \equiv 1 \bmod (p-1)(q-1)$.
4. Bob makes n and e public, and keeps p, q, d secret.
5. Alice encrypts m as $c \equiv m^e \bmod n$ and sends c to Bob.
6. Bob decrypts by computing $m \equiv c^d \bmod n$. [2]

Explanations of these steps can be made as follows.

1. 1st step is there because of achieving the mathematical difficulty of divisibility, n is equal to two prime factors and two prime factors alone, for it's the worst case scenario as one needs to know exactly at least one of the two numbers p or q in order to find the factorization, as if p or q were composite their corresponding prime factors would have reduced the difficulty of divisibility of n
2. 2nd step is there because in modular arithmetic one can work with exponents and modulus' totient, which is $(p-1)(q-1)$ in our specific case, according to

Euler's Theorem; and then in order for e to have a multiplicative inverse in modulus $(p-1)(q-1)$, $gcd(e, (p-1)(q-1)) = 1$ needs to be true. [2]

3. 3rd step is the calculation of d using Extended Euclidean Algorithm while working on the exponents. [2]
4. 4th step explains which information is made public and which information is kept secret.
5. 5th step explains the actual encryption process.
6. 6th step explains the actual decryption process.

3.2.1.1. EULER'S THEOREM'S BASIC PRINCIPLE

Let a, n, x, y be integers with $n \geq 1$ and $gcd(a, n) = 1$.

If $x \equiv y \bmod \varphi(n)$, $a^x \equiv a^y \bmod n$, where $\varphi(n)$ denotes the totient for n.

In other words, if you want to work mod n, you should work mod $\varphi(n)$ in the exponent. [2]

Now let's consider it for our specific case:

$a = m;$

$n = pq;$

$x = de;$

$y = 1;$

$n \geq 1$ is always true since $n = pq$, and even the smallest prime, 2, is greater than 1, but $gcd(m, pq)$ is *assumed* to be true, but not necessarily true, and this is one part this study is inadequate. For it to be always true, m needs to be smaller than both p and q. So for our specific case, principle becomes: If $de \equiv 1 \bmod (p-1)(q-1)$, $m^{ed} \equiv m \bmod pq$. d now needs to be calculated using Extended Euclidean Algorithm. In order to do that one needs to know p and q since the modulus requires $p-1$ and $q-1$ and there's no easy way to find it only using the value n.

3.2.1.2. EXTENDED EUCLIDEAN ALGORITHM

Extended Euclidean Algorithm is used to find the inverse of a number in a modulo, as in $ed \equiv 1 \bmod (p-1)(q-1)$ means that e and d are inverses of each other in modulus $(p-1)(q-1)$.

In a division operation there are dividend, quotient, divisor and remainder. Dividend = quotient x divisor + remainder. For example in the operation 17/6, 17 = 2 x 6 + 5, 17 is called the dividend, 2 is called the quotient, 6 is called the divisor and finally 5 is called the remainder of the division.

The Euclidean algorithm, the algorithm that finds gcd of two numbers, does this division operation for multiple times in which the last remainder becomes the new divisor, the last divisor becomes the new dividend, and finally the last dividend is discarded. This is done until the divisor is equal to 1.

dividend = quotient x divisor + remainder

Explaining how the algorithm works is much simpler by showing an example:

Let's consider that gcd(1337, 7331) needs to be known:

7331 = 5 x 1337 + 646

1337 = 2 x 646 + 45

646 = 14 x 45 + 16

45 = 2 x 16 + 13

16 = 1 x 13 + 3

13 = 4 x 3 + 1

3 = 3 x 1 + 0

The last division's, the division with remainder equals to 0, divisor is the gcd. So gcd$(1337, 7331) = 1$

Now assume that the quotients of this algorithm is numbered such that:

$quot_1 = 5$

$quot_2 = 2$

$quot_3 = 14$

$quot_4 = 2$

$quot_5 = 1$

$quot_6 = 4$

$quot_7 = 3$

The Extended Euclidean Algorithm computes $ax_i + by_i = \gcd(a, b)$ and if $\gcd(a, b) = 1$ then it means that the number a has an inverse under mod b or vice versa. Extended Euclidean Algorithm computes x_i and y_i with the formula below:[2]

$x_0 = 0, x_1 = 1, x_j = -quot_{j-1}x_{j-1} + x_{j-2}$

$y_0 = 1, y_1 = 0, y_j = -quot_{j-1}y_{j-1} + y_{j-2}$

Let's try this for 1337 and 7331:

$x_0 = 0, \ x_1 = 1$

$x_2 = -5x_1 + x_0 \quad = -5$

$x_3 = -2x_2 + x_1 \quad = 11$

$x_4 = -14x_3 + x_2 \quad = -159$

$x_5 = -2x_4 + x_3 \quad = 329$

$x_6 = -x_5 + x_4 \quad = -488$

$x_7 = -4x_6 + x_5 \quad = 2281$

Which means that 1337 x 2281 $\equiv$ 1 mod 7331, which is true. Now in the RSA $a = e$ and $b = (p-1)(q-1)$ and d would be what 2281 represents.

4. RSA IMPLEMENTATION ON MSP430G2553

This section explains the code pieces in order to achieve each step in the RSA Algorithm. In the implementation however the order of tasks are changed to 2,1,3,4,5,6.

4.1. IMPLEMENTATION OF 2nd STEP

This step is rather trivial if done first. In this implementation *e* was predefined as *3*.[4] "Choosing a short public exponent makes RSA more efficient, as fewer computations are needed to raise a number to the power *e*."[1] Picking a small value for *e* limitates the length of the message *m* since if it's too small the modulus wouldn't do any work.[3] The following code piece can be found in main.c.

```
#define E 3
```

4.2. IMPLEMENTATION OF 1st STEP

This step is rather complex compared to 2nd so it was implemented in modules, and will be studied in sub-steps.

This step is about generating two random primes and multiplying them. Obviously multiplication is rather trivial, but generating random primes can be difficult. In order to generate random primes, one needs to generate random numbers to begin with and then put them in a primality test in order to see if they're primes or not. This step is part of rsa_init function under /src/rsa/rsa.c in [4].

```
unsigned long long rsa_init(unsigned char e, unsigned long long *d) {//  returns d and n
 unsigned long p = rsae3prime_gen();
 unsigned long q = rsae3prime_gen();
```

```
  while (p == q) {  //  In order to prevent having both primes equal, this is very unlikely but possible
    p = rsae3prime_gen();   //  Change either one of the primes, p in this case
  }


  unsigned long long n = (unsigned long long) p * q;  //  casting may be unnecessary
```

4.2.1. RANDOM NUMBER GENERATION

For random number generation the time difference between the VLO and DCO was used.[5] Some minor changes were made to the code piece provided by MIT, it generated 16-bit random numbers but 32-bit random numbers were required for a key size of 64-bits.

In cryptosystems, any leak of information in the processes is dangerous, and generating random numbers is no exception, the randomness of the numbers shouldn't fit to a pattern, the numbers should be uniformly distributed. The RNG here isn't tested for such measures.

The header files used to achieve this sub-step can be found under /src/bali_rand in [3].

4.2.2. PRIMALITY TEST

There are two types of primality tests, deterministic and probabilistic. In deterministic tests it is for certain if a number is prime or composite. In probabilistic tests however it is determined if a number is composite or probable prime. The number may pass the test but still be a composite number, if it is a prime however it certainly passes the test.

In this study Miller-Rabin and trial division tests were used but only a deterministic version of Miller-Rabin was implemented in MSP430G2553. The header files used in order to implement this sub-step is located under /src/32-bit_det_MR and the function names are miller_rabin and try_div in [3].

4.2.2.1. MILLER-RABIN PRIMALITY TEST

Miller-Rabin is a probabilistic primality test that is based on the unproven Extended Riemann hypothesis. It checks whether a number is composite or not, and if not the number is said to be strong pseudoprime. A pseudoprime can either be a composite or a prime. The test is probabilistic because all primes are pseudoprimes but not the other way around, as in not all the pseudoprimes are primes.[6] The code can be found in /src/32-bit_det_MR and the function name is miller_rabin.

4.2.2.1.1. ALGORITHM

The algorithm used was in pseudocode as follows:[7]

Input #1: $n > 3$, an odd integer to be tested for primality;

Input #2: k, a parameter that determines the accuracy of the test

Output: *composite* if n is composite, otherwise *probably prime*

Write $n - 1$ as $2^r d$ with d odd by factoring powers of 2 from $n - 1$

WittnessLoop: **repeat** k times:

pick a random integer a in the range $[2, n - 2]$

$x \leftarrow a^d \bmod n$

if $x = 1$ or $x = n - 1$ **then**

continue WitnessLoop

repeat $r - 1$ times:

$x \leftarrow x^2 \bmod n$

if $x = n - 1$ **then**

continue WitnessLoop

return *composite*

return *probably prime*

4.2.2.1.2. IMPLEMENTATION

Some changes had been made to this pseudocode, for numbers up to 4,759,123,141, a number that is greater than 2^{32} and thus a very good fit for numbers up to 32-bit, the test is deterministic under bases 2, 7, 61 [8]. With this information *k* is no longer needed and it can be considered as *k = 1, a* is no longer picked at random but it is now an array iterating through 2, 7 and 61 and for each number trial, the test is done for all three bases. It can be further optimized if needed for there are other deterministic bases for smaller values.

The test itself was tested in computer and compared to trial division for 32-bit values fit to the limitation of RSA with *e = 3*, and for some values of *n,* the number that is being tested, the result was a false positive, those *n* values were put to an exception list (26^{th} line), until fixing.

4.2.2.2. TRIAL DIVISION PRIMALITY TEST

This test is the regular trial division test which is to square root the number for trial and then test if the number is divisible by primes up to that point. It is based on the fact that when a number is factorized, its factors are inversely proportional to one another. In worst case scenario an integer is a multiple of two close sized integers.

An integer can be factorized as the multiplication of; a large integer and a small integer, or a medium integer and another medium integer, or the numbers in between. For example if *34* is being tested for primality $\sqrt{34} \approx 5.8$ so if it is divisible by *2, 3* or *5* then it is a composite, which is true. Notice that *34* can be factorized as *2 x 17* the test won't check if it *34* is divisible by *17* but it will check for *2,* and that's sufficient because the number could've been at worst a prime squared which the test will definitely check..

This test wasn't implemented in MSP430G2553 because it required a list of primes up to 2^{16} which doesn't at all fit in the microprocessor's memory. It was used for testing purposes for other test(s) in computer medium. Still the codes can be found in /src/32-bit_det_MR and the function name is try_div.

4.2.3. 32-BIT RANDOM PRIME GENERATION

The above code piece, /src/rsa/rsa.c rsae3prime, first generates random integers of the form $2^{31} \leq 6n - 1 < 2^{32}$ and then checks for primality and if the number passes the test it returns the number, if not repeats the process.

The reason of the range limitation is in order to have RSA's public modulus *n* precisely as a 64-bit number; and the reason behind the form *6n – 1* is that apart from *2* and *3* every prime number is in the form of *6n -1* or *6n + 1*. *6n+1* is excluded because every integer with that form fail to satistfy RSA Algorithm's 2nd step's condition of *gcd(e, (p-1)(q-1)) = 1* for *e = 3*. On the other hand every integer of the form *6n -1* satisfy that same condition, which reduces the generated prime candidates to a very limited pool.

```
unsigned long rsae3prime_gen(void) {
  unsigned long n = 0;
  while (!(miller_rabin(n))) {//Miller-Rabin test
     //while  (!(try_div(n)))  {//Trial  Division  test  with
predetermined primes < 16 bit DOESN'T FIT
    n = 6 * (357913942 + (prand(rand()) % 357913941)) - 1; //
357913941 = 715827883 - 357913942, the values to fit 32-bit
exactly
  }
  return n;
}
```

If the *e* were to change to a different value than *3,* this function shouldn't be used. Although a very similar function can be written.

4.2.4. COMPUTATION OF MODULUS N

The primes *p* and *q* is found by using above mentioned sub-steps. There is however one small, but a vital step before calculating the modulus *n.* The primes *p* and *q* are not supposed to be equal to one another. This is obviously very unlikely but nonetheless, possible.

To address that, a while loop, which verifies that $p \neq q$ and if not, re-generate either one of the primes, is put before finding n.

4.3. IMPLEMENTATION OF 3rd STEP

In this step the exponential inverse d of e in mod n is implemented. In order to do that Extended Euclidean Algorithm was implemented.[9] This is the remaining part of the /src/rsa/rsa.c rsa_init function.

```
  unsigned long long totn = (unsigned long long) (p - 1) * (q -
1);  //  casting may be unnecessary
  long long s, t;
  *d = gcdExtended(e, totn, &s, &t);

  if (s < 0) {
    *d = s + totn;
  }

  else  {
    *d = s;
  }
  return n;
}
```

4.4. IMPLEMENTATION OF 4th STEP

This step, making n and e public, although some compatibility functions in parsing were written, was primarly done by communcation module and won't be covered here.

4.5. IMPLEMENTATION OF 5th STEP

This is the step where the actual encrypting happens. Mathematically it looks rather trivial but when implementing it wasn't at all an easy task. This is mostly because most variables were used to the limit, which required the attention for overflows at all times. For example the function to take an integer's power in modulus used to overflow, and that demanded for key size to be dropped to half of its current value, thankfully another function was found to address this issue.[10] Also in this step, it wasn't mentioned in RSA Algorithm but, an encoding scheme was required for regulating the message size and uniformity. This will be covered as a sub-step. This step covers /src/rsa/rsa.c rsa_e function.

```
void rsa_e(unsigned long long e, unsigned long long n, unsigned
long long *outblock, unsigned char blocksiz, unsigned char
*outmsg) {// returns outblock
  encode(outmsg, outblock, blocksiz);
  unsigned char i = 0;
  for (i=0; i<blocksiz; i++) {
    outblock[i] = PowMod(outblock[i], e, n);
  }
}
```

4.5.1. ENCODING

When communicating repeating messages may be sent, in our regular life we use some phrases much more regularly than the others such as “What's up?” or “Good morning”. If the same numbers were to appear when encrypting these messages, it would leak information.

The message may not always be the size the encryption process requires, this could also cause leaks and may prevent the message from being encrypted at all. This may also leak information.

When communicating, some blocks, or some whole messages even, may get lost, disrupted or corrupted. This would disrupt the communication between the parties, a numbering scheme needs to be present for understanding which block belongs to which

message and in what order. Also Eve may use this information to cause confusion between Alice and Bob.

In order to address all of the above mentioned problems an encoding scheme needs to be present. The encoding is done before encryption. The function encode in /src/parsing/parsing.c covers the implementation to the solutions of above mentioned problems.

```
void encode(unsigned char *outmsg, unsigned long long
*outblock, unsigned char blocksiz) { //  returns block


    //    ENCODING
    unsigned char r1 = 0xFF;
    unsigned char r2 = 0xFF;
    //  0x00 28 51 46 <= block[i] < 0x40 00 00 57 00 00 16 AC


    rng_2_uc(&r1, &r2);
    outblock[0] = (unsigned long long) r1 << (8 * 3); //  01 +
should be random [1]
    outblock[0] |= (unsigned long long) (2 ^ r2) << (8 * 2); //
# START XOR with [2]
    outblock[0] |= (unsigned long long) (0 ^ r1) << (8 * 1); //
block # XOR with [1]
      outblock[0] |= (unsigned long long) r2 << (8 * 0); //
should be random [2]


    rng_2_uc(&r1, &r2);
    outblock[1] = (unsigned long long) r1 << (8 * 3); //  01 +
should be random [1]
    outblock[1] |= (unsigned long long) (blocksiz ^ r2) << (8 *
2); //    # of blocks to come XOR with [2]
    outblock[1] |= (unsigned long long) (1 ^ r1) << (8 * 1); //
block # XOR with [1]
      outblock[1] |= (unsigned long long) r2 << (8 * 0); //
should be random [2]
```

```
    unsigned char i;
    for (i = 2; (i < blocksiz); i++) {
      rng_2_uc(&r1, &r2);
       outblock[i] = (unsigned long long) r1 << (8 * 3); //  01
+ should be random [1]
      outblock[i] |= (unsigned long long) (outmsg[i-2] ^ r2) <<
(8 * 2); //    # of blocks to come XOR with [2]
        outblock[i] |= (unsigned long long) (i ^ r1) << (8 *
1); // block # XOR with [1]
       outblock[i] |= (unsigned long long) r2 << (8 * 0); //
should be random [2]


    }
}
```

In the above code an 8-bit information, a char, is taken as input and it was transformed to a 64-bit, 4 chars, number. Two 8-bit random integers are generated and then put as the 1st and the 4th char of a 32-bit block, and then r1 was xored with the block number and written to 3rd position, and then r2 was xored with the actual message's char and was written to 2nd position, thus achieved the diffusion. This is crucial, this makes it possible to send the same message over and over again and get different blocks with each.

The char in the 1st position needs to be greater than 1, in order to make the message, or rather the plaintext definitely greater than 0x00285146 which is required for smallest modulus *n* possible for *e=3* in 64-bit. The upper limit of the message is irrelevant as it is a 64-bit number whereas we are working up to 32-bits with the message.

First two blocks of the packet involve the “start of the message” and how many blocks the message has. One can use one more block in order to id the messages sent as well. Even without that, when Alice sends this to Bob, it allows him to understand if the message was disrupted or not. This part may be optimized for more accurate inspections.

4.6. IMPLEMENTATION OF 6th STEP

This is the step where the actual decryption happens. It is very similar to encryption function, as mathematically they're almost identical. This step covers /src/rsa/rsa.c rsa_d function.

4.6.1. DECODING

The receiving side needs to decode the encrypted message received after it has been decrypted. decode, the corresponding function to encode can be found in /src/parsing/parsing.c. This code, besides returning the message, returns 0 as a warning if the block mismatches with its block number and returns 1. This step covers the function decode in /src/parsing/parsing.c covers the implementation.

5. CONCLUSION

5.1. IMPLEMENTING FOR MSP430G2553

Implementing for msp430g2553 is easy if you're comfortable with C language skills for the most part. But when the variables are filled to their limit it is like carrying two buckets of water, filled to the brim, with your eyes closed. One needs to be very careful in order to avoid overflow issues and be as efficient as possible while doing that since the actual required key size for RSA for it to be secure is around 6800-bit for the time being.[1] This begs for multiprecision integer libraries, the libraries that focus on arithmetic operations with integers of arbitrary bit size, but unfortunately in my research phase I was unable to find such library for msp430g2553 medium until after I was very close to finishing it. Even now I'm very skeptical about it that I won't include it in this study. As a result I ended up working with largest standard variable size possible in C, which is 64-bit. Though this helped in other means, like generating primes and random numbers were much simpler, where arithmetic operations were probably much harder I guess. On the other hand I'm not sure if I'd be able to test the primality of very very large numbers if I were to use msp430g2553 to the limit. So everything being taken into account, I believe the key size is in a well place for demonstrative purposes of RSA in msp430g2553.

5.2. TEST RESULTS FOR RSA FUNCTIONS

In this section how long rsa_init, rsa_e, rsa_d functions take in time will be measured.

It will be in the format below:

function name: [measurement1, measurement2, measurement3, …] (all in seconds)

rsa_init: [7.00, 4.47, 1.72, 3.09, 2.97, 5.38, 4.99, 5.53, 5.53, 9.75]

average = 5.043 seconds

rsa_e: [2.61, 2.75, 2.58, 1.94, 2.54, 2.63, 2.56, 2.58, 2.59, 2.55]

average = 2.533 seconds

rsa_d: [4.56, 7.78, 6.67, 7.68, 6.70, 10.75, 12.82, 8.73, 12.83, 6.69]

average = 8.521 seconds

5.3. GENERIC ATTACKS

5.3.1. EXHAUSTIVE SEARCH

RSA's security is based on the fact that public-key *n*'s factorization, or in other words knowing what *p* or *q* is. Whomever knows either *p* or *q* can decrypt the encrypted messages encrypted with corresponding public-keys. An exhaustive search could be thus be defined as trying to find *p* or *q* by trial division of a given *n*. Since we know that *n* is a 64-bit number, its square root is a 32-bit number. But we also know that in our special case, both *p* and *q* are in fact 32-bit primes, so one only needs to go through primes that are greater than 2^31 and less than 2^32. One can decrease this range even further by discarding the primes that doesn't satisfy the rule involving *e*, which was in our case equal to 3. In the test for Miller-Rabin primality test's correctness, it was observed that there are in fact 49091942 primes in 32-bit range given above, and satisfy the condition in 1st step of RSA Algorithm for *e=3*. In order to do an exhaustive search attack on this specific system one needs to calculate $\sqrt{n}$ and then test against primes that are less than $\sqrt{n}$ in that specific range above. It can be said that one need roughly 49091942 division processes in order to find *p* or *q*. Assuming another MSP430G2553 is used with 16MHz clock speed and the primes above are in an array, which is in fact untrue because it takes roughly 187 MB space w/o compression but we can assume one may procedurally generate primes on the go and it takes no clock cycle, If each modulus operation takes 1 clock cycle then it should take around 2.93 seconds to break. For testing purposes the below code was written to test if that was true:

```
unsigned long rsae3_exh(unsigned long long n) { //  returns the
small factor of n

  unsigned long p = 2147483693; //  smallest 32-bit number of
the form 6n-1
```

```
  while (1) { //  loops until p divides n
    if (!(n % p)) { //  returns true only if p divides n, as in
p is one of the two factors of n
      return p;
    }
     p += 6; //  increase p by 6 as successive numbers of the
form 6n-1 are separated by 6, for example 5, 11, 17, 23...
  }
}
```

In the above code the while loop did around 1666 loops in 1 second for a random *n* with clock adjusted to 16MHz, but around 64827446 loops were required in order to reach the small factor of *n*, and that meant roughly 10 hours 48 minutes were needed in order to find *p*.

But if *n* has 2147483693, smallest 32-bit prime number of the form 6k-1, as one of its factors then this function returns *p* almost immediately. Or if *n = 4294967279 x 4294967291,* the largest two distinct 32-bit prime numbers of the form 6k-1, then it would perhaps take much longer.

Perhaps a better search could be made with first calculating $\sqrt{n}$ and then calculate both $\sqrt{n}$ - 2147483693 and 4294967291 - $\sqrt{n}$ and if the first calculation's result is less than second start from the beginning 2147483693 or if the second calculation's result is less than the first start from the end. Although it should be in consideration that as numbers get larger the prime number frequency decreases, so perhaps the second calculation should be chosen in some cases even though if first calculation's result is less than the second calculation's result.

5.3.2. MEET-IN-THE-MIDDLE ATTACK

Assuming that the encoding scheme works as it supposed to be then this attack is not possible, as it is based on the duplicates of the ciphertexts. With a working encoding scheme whether two ciphertexts are equal or not is irrelevant.

5.4. FUTURE WORK

RSA is primarily used for key exchange of symmetric key cryptosystems and as digital signatures, so there's definitely room for improvement. Even though it is secure than bare communication, this implementation is still pretty insecure to put to field as it is. The ideas of improvement could be in the fields of increasing the key size, using a verified uniformly random number generator, a verified 'padding' scheme, implementing authentication, implementing a symmetric key cryptosystem like AES for efficiency, implementing hash functions, and using one of the standard modes.

6. REFERENCES

[1] N.Ferguson & B. Schneier & T. Kohno, Cryptography Engineering: Design Principles and Practical Applications, Indianapolis: Wiley Publishing, Inc., 2014

[2] W. Trappe & L. C. Washington, Introduction to Cryptography with Coding Theory: Second Edition, Upper Saddle River: Pearson Education, Inc., 2006

[3] Stack Exchange, "Lower Limit on RSA message length" Internet: http://stackoverflow.com/questions/12223325/lower-limit-on-rsa-message-length-in-m2crypto/41372986#41372986

[4] Github, "The Repository of the codes written" Internet: https://github.com/Naeus/msp430/tree/master/rsa_bali

[5] Github, "The repository of the codes for RNG in MSP430" Internet: https://github.com/0/msp430-rng

[6] University of Waterloo, "Miller-Rabin Primality test" Internet: http://cacr.uwaterloo.ca/hac/about/chap4.pdf

[7] Wikipedia, "Miller-Rabin Primality Test" Internet: https://en.wikipedia.org/wiki/Miller%E2%80%93Rabin_primality_test#Computational_complexity

[8] American Mathematical Society, "Deterministic Variant of Miller-Rabin Primality test for numbers under 32-bit" Internet: http://www.ams.org/journals/mcom/1993-61-204/S0025-5718-1993-1192971-8/S0025-5718-1993-1192971-8.pdf

[9] Geeks for Geeks, “Extended Euclidean Algorithm in C” Internet: http://www.geeksforgeeks.org/basic-and-extended-euclidean-algorithms/

[10]David Eisenstat, Stack Exchange, “Non-overflowing modular exponentiation” Internet: http://stackoverflow.com/questions/41397158/is-there-a-way-to-do-modular-exponentiation-that-requires-at-most-double-the-byt/41398685#41398685

7. APPENDIX