Secure Communication: An RSA Implementation on C, for MSP430 with key size of 64-bit

İbrahim Bali

Introduction

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.

Overview

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.

What RSA Is

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.

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

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.

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 $m \equiv c^d \bmod n$.

Bob (Decrypting side)

As suggested Bob does first four steps and these are in C are as follows:

Steps 1 through 4 can be done using rsa_init function as follows:

```
#define E 3
unsigned long long d = 0;
unsigned long long n = 0;
n = rsa_init(E, &d);    //   returns n and d for a given e
```

In this case $e = 3$ is a given. Let's say Bob generated $n = 1460383995585742 9723$, which is factorized as $3561039371 \times 4101004913$, he then calculated $d = 973589329879692 3627$ from the $p = 3561039371$, and $q = 4101004913$. He then publishes ***n*** and ***e*** using communication module and then waits for a response from Alice.

```
unsigned long rsae3prime_gen(void) {  // This function may take significant time, thus timer freq.
  unsigned int BCSCTL1_old = BCSCTL1; //BALI
  unsigned int DCOCTL_old = DCOCTL; //BALI
  DCOCTL = CALDCO_16MHZ;  // BALI increasing clk speed
  BCSCTL1 = CALBC1_16MHZ; // BALI increasing clk speed
  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,
  }
  DCOCTL = DCOCTL_old;  // BALI
  BCSCTL1 = BCSCTL1_old;  // BALI
  return n;
}
```

How Primes Are Generated

After Alice sends her encrypted message **0x5F089997DBA2E9CA** Bob first decrypts it with rsa_d which is a very similar function to rsa_e. Decryption of that message gives **0x5D485F09** as expected. He then decodes it to have the message “A”.

Alice (Encrypting side)

In order to do do 5th step Alice first receives ***n*** and ***e*** that Bob sent and she then uses the function rsa_e which has its own encoding function and encrypts the encoded blocks:

```
    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);
          }
}
```

For simplicity let's say that Alice wants to send “A” which is 0x41 in hexadecimal. Encode function then generates two 8-bit numbers R1 and R2, puts them in a 32-bit memory space in 0xR1 00 00 R2 and then xors R2 with our char(8-bit) “A” and places it in 2nd place of 8-bits our 32-bit memory then becomes 0xR1 M2 00 R2 and finally it xors R2 with the block number, which is 0 in this case since we're sending the very first char of the message, finally to have the 32-bit memory in the form of **0xR1 M2 B1 R2**. Let's say that **R1 = 0xB8**, and **R2 = 0xFF** then outblock before it's encrypted is **0x5D485F09**, and after encryption it is **0x5F089997DBA2E9CA**. This is then sent through communication module to be received on Bob's end.

Sample Results

On average of 10 rsa_init function takes 5.043 seconds in between breakpoints, where as rsa_e, and rsa_d takes 2.533 and 8.521 seconds respectively.

Also for an exhaustive search it takes roughly to 0.0006 to 214834.2923 seconds to crack ***n*** to its factors ***p*** and ***q***.

The Function that was Used for Exhaustive Search:

```
unsigned long rsae3_exh(unsigned long long n) {
  unsigned long p = 2147483693;
  while (1) {
    if (!(n % p)) {
      return p;
    }
    p += 6;
  }
}
```