RSA implementation in MIPS Assembler Language

Jannik Franz, Tobias Ruck

May 20, 2015

1.1 Algorithm

1.1.1 Introduction

Generally the algorithm implemented follows along the suggested exercise and approach. This section will focus on explaining how the different parts were implemented. Later overall constraints created by using such methods will be explained.

1.1.2 Implementation

Prime Generation

To generate prime numbers the "sieve of eratosthenes" is used. Generally this algorithm says whether any number up to n is a prime or not. The limit n has to be predefined before sieving. For the implementation $n = 1500$. This means that 1500 bytes have to be allocated which mark whether or not a certain number is a prime or not.

```
1 signed char primes[MAX_NUMBERS];
2 memset(primes, 1, sizeof(primes));
3 for (i = 2; i < MAX_NUMBERS; i++)
4     if (primes[i])
5         for (j = i; j * i < MAX_NUMBERS; j++)
6             primes[j * i] ^= primes[j * i];
```

The first line allocates the space for up to MAX_NUMBERS bytes. MAX_NUMBERS is 1500 in this case. The second line sets the value of all the adresses reserved to 1. The first loops goes over every number from 2 to MAX_NUMBERS. 0 and 1 are skipped since they are both not considered prime and not plausible for the algorithm because it strokes out multiples of previously processed numbers from the array marking them as non-primes as they are multiples. Next up is making sure to only take multiples of numbers currently considered prime to optimize. The inner loop calculates multiples of the current number and crosses them out from the array via an exclusive-OR operation. Eventually we are left with an array of zeroes and ones marking non-primes and primes, respectively.

Totient

The calculation of the totient is exactly the formular from the book: $N = p \cdot q$ and $\varphi = (p-1)(q-1)$, where p and q are arbitary (and large enough) primes.

Public Key

In the exercise it is stated that the user has given a public key e which is coprime to φ. In our code the public key is generated with φ by finding a small prime coprime to φ. So we are looking for the GCD[1] of a prime and φ with

[1] Greates Common Divisor

```
while (b != 0) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
return a;
```

where a is the prime and $b = \varphi$.

Private Key

Now we create the private key with the help of the public key e and the totient φ.

```
x0 = 0, x1 = 1;
while (e > 1) {
    q = e / phi.phi;
    t = phi.phi, phi.phi = e % phi.phi, e = t;
    t = x0, x0 = x1 - q * x0, x1 = t;
}
if (x1 < 0)
    x1 += b0;
return x1;
```

Encryption/Decryption

Encryption and decryption follow a simple pattern. Every character of a message is calculated with $m^e \% N$. To prevent overflows from happening for large e's a special algorithm is used to keep modulating before that happens.

```
x = 1, y = m;
while(e > 0) {
    if(e % 2) {   // (e & 1)
        x *= y;
        x %= N;
    }
    y = y * y;
    y %= N;
    b /= 2;
}
return x;
```

The only difference with decrypting to encryption is that the private instead of the public key is used.

1.2 Constraints

Prime Generation Although SPIM has no problems to generate the array of 1500 entries for the primes, finding a prime near n will fail for $n > 100$ for unknown reasons, as of now.

32-Bit Due to the variables being 32-bit it is constrained by the modulus, as you can't encrypt characters with a byte value bigger than the modulus. This would cause a loss of data.