CSE5014 · Cryptography and Network Security

Plain RSA Implementation

Liu Leqi / 12011327

June 16, 2023

1 Descryption

RSA (Rivest–Shamir–Adleman) is one of the oldest public-key cryptosystem, and is widely used for secure data transmission. In a public-key cryptosystem, the encryption key is public and distinct from the decryption key, which is kept secret (private). An RSA user creates and publishes a public key based on two large prime numbers, along with an auxiliary value. The prime numbers are kept secret (or deprecated). Messages can be encrypted by anyone, via the public key, but can only be decrypted by someone who knows the private key.
The security of RSA relies on the practical difficulty of factoring the product of two large prime numbers, the "factoring problem". Breaking RSA encryption is known as the RSA problem. Whether it is as difficult as the factoring problem is an open question. There are no published methods to defeat the system if a large enough key is used.
In this project, I have implemented a plain RSA public-key cryptiosystem using *rust*. We can use the system to encrypt the message and decrypt the corresponding ciphertext. Besides, the public key and private key can be ported into a *.key* file so that we can exchange the public key to others.

2 Steps

Suppose that Alice and Bob want to communicate with each other through a reliable but not secret channel, i.e., the message transmitted will not lose in channel but may be eavesdropped by malicious attacker.

2.1 Key generation

1. Choose two distinct large prime numbers p and q.
    - To make factoring harder, p and q should be chosen at random, be both large and have a large difference. We can continuously choose random integers and then utilize primality test until two primes are found.
2. Compute $n = p \cdot q$ and $\phi(n) = (p-1)(q-1)$, then destroy p and p.
    - n is released as part of the public key and private key. The bit length of it is the key length.
3. Choose an integer e, which is coprime with $\phi(n)$. Commonly $2^{16} + 1 = 65537$.
4. Compute integer d such that $ed \equiv 1 (mod\ \phi(n))$.
    - It can be computed by using the extended Euclidean algorithm.

2.2 Key distribution

Alice and Bob will send their public key (n_i, e_i) $(i = 1, 2)$ to each other through the channel and keep their private key (n_i, d_i) $(i = 1, 2)$ secretly.

2.2.1 Encryption

To encrypt a message m, we can using the public key (n, e) as below.

$$c \equiv m^e (mod\ n)$$

2.3 Decryption

To decrypt a ciphertext c, we can using the private key (n, d) (the private key is corresponding to the public key) as below.

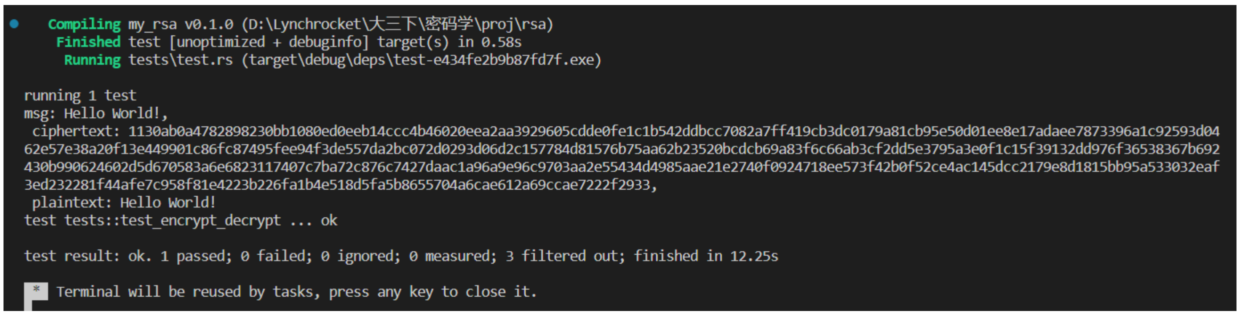
$$m \equiv c^d \equiv m^{ed} (mod\ n)$$

3 How to use

Here is an example for encryption and decryption.

```rust
#[test]
fn test_encrypt_decrypt() {
    // initialize the enc() and dec()
    let gen = generator::Generator::new(1024).unwrap();
    let enc = plain_rsa::PublicKey::from(&gen);
    let dec = plain_rsa::PrivateKey::from(&gen);

    assert_eq!(enc.n, dec.n);

    let msg = String::from("Hello World!");
    let ciphertext = enc.encrypt(&msg[..]).unwrap();
    let plaintext = dec.decrypt(&ciphertext[..]).unwrap();

    assert_eq!(&msg[..], &plaintext[..]);
    println!(
        "msg: {}, \nciphertext: {}, \nplaintext: {}",
        msg, ciphertext, plaintext
    );
}
```

The running result of the test shows that the implementation works well.



```
   Compiling my_rsa v0.1.0 (D:\Lynchrocket\大三下\密码学\proj\rsa)
    Finished test [unoptimized + debuginfo] target(s) in 0.58s
     Running tests\test.rs (target\debug\deps\test-e434fe2b9b87fd7f.exe)

running 1 test
msg: Hello World!,
 ciphertext: 1130ab0a4782898230bb1080ed0eeb14ccc4b46020eea2aa3929605cdde0fe1c1b542ddbcc7082a7ff419cb3dc0179a81cb95e50d01ee8e17adaee7873396a1c92593d04
62e57e38a20f13e449901c86fc87495fee94f3de557da2bc072d0293d06d2c157784d81576b75aa62b23520bcdcb69a83f6c66ab3cf2dd5e3795a3e0f1c15f39132dd976f36538367b692
430b990624602d5d670583a6e6823117407c7ba72c876c7427daac1a96a9e96c9703aa2e55434d4985aae21e2740f0924718ee573f42b0f52ce4ac145dcc2179e8d1815bb95a533032eaf
3ed232281f44afe7c958f81e4223b226fa1b4e518d5fa5b8655704a6cae612a69ccae7222f2933,
 plaintext: Hello World!
test tests::test_encrypt_decrypt ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out; finished in 12.25s

 *  Terminal will be reused by tasks, press any key to close it.
```

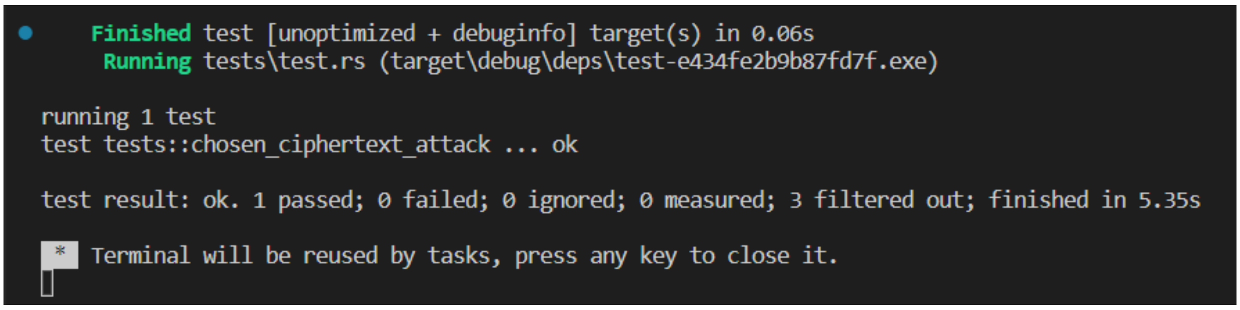
Figure 1: Test encryption and decryption

4 Chosen ciphertext attack

Plain RSA is not secure under the chosen ciphertext attack, where the attacker can query the encryption oracle and the decryption oracle. Here I also reveal an example to show that plain RSA is not ind-CCA. The main idea is that if the attacker wants to know the decryption of the ciphertext $c \equiv m^e (mod\ n)$, it can ask the decryption oracle with the suspicious ciphertext $c' \equiv cr (mod\ n)$ with some random value r. After retrieving the decryption $m' \equiv mr (mod\ n)$, the attacker can reveal the message by multiplying m' with the modular inverse of r modulo n.

```rust
#[test]
fn chosen_ciphertext_attack() {
    // initialize the oracle
    let gen = generator::Generator::new(1024).unwrap();
    let enc_oracle = plain_rsa::PublicKey::from(&gen);
    let dec_oracle = plain_rsa::PrivateKey::from(&gen);

    // secret_msg is not visible by the attacker but the attacker wants to reveal it
        from the ciphertext
    let secret_msg = BigUint::from_bytes_be("I am secret msg".as_bytes());
    let ciphertext = secret_msg.modpow(&enc_oracle.e, &enc_oracle.n);

    // the random multiplier
    let adder = BigUint::from_bytes_be("msg adder".as_bytes());
    let forge_ciphertext = ciphertext * adder.modpow(&enc_oracle.e, &enc_oracle.n);
    let forge_msg = forge_ciphertext.modpow(&dec_oracle.d, &dec_oracle.n);

    // compute the modular inverse of r modulo n
    let (mut adder_inv_module_n, _, _) = algorithms::ext_euc(
        &BigInt::from_biguint(num_bigint::Sign::Plus, adder.clone()),
        &BigInt::from_biguint(num_bigint::Sign::Plus, enc_oracle.n.clone()),
    );
    while adder_inv_module_n.is_negative() {
        adder_inv_module_n += BigInt::from_biguint(Sign::Plus, enc_oracle.n.clone());
    }

    // the secret_msg is revealed
    let adder_inv = adder_inv_module_n.to_biguint().unwrap();
    let retrieve_msg = forge_msg * adder_inv % enc_oracle.n;

    assert_eq!(retrieve_msg, secret_msg);
}
```

The running result of the test shows that the attack works well.



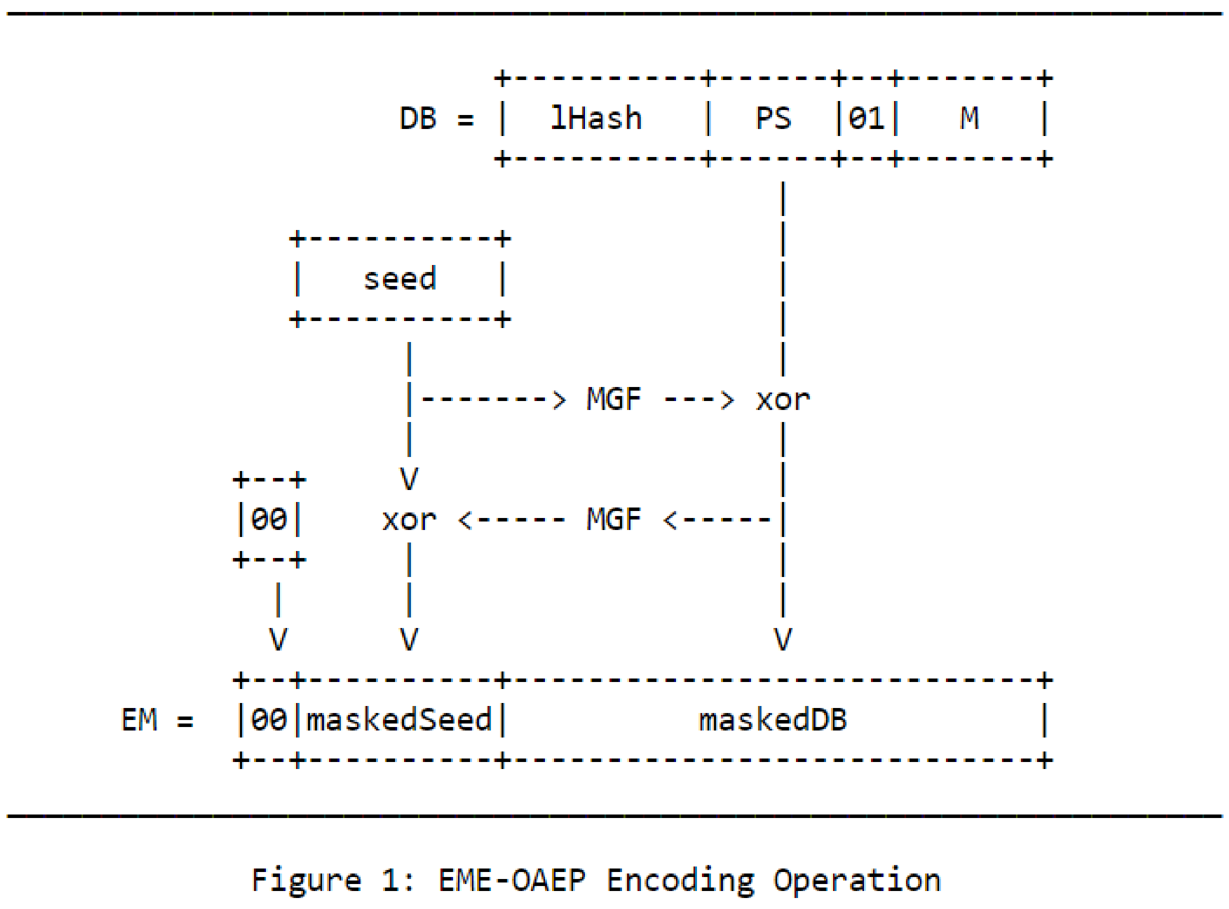
```
●    Finished test [unoptimized + debuginfo] target(s) in 0.06s
      Running tests\test.rs (target\debug\deps\test-e434fe2b9b87fd7f.exe)

   running 1 test
   test tests::chosen_ciphertext_attack ... ok

   test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 3 filtered out; finished in 5.35s

 *  Terminal will be reused by tasks, press any key to close it.
```

Figure 2: Chosen Ciphertext Attack

5 Further Work

The plain RSA is no longer to be recommended to use in a secure cryptosystem. One optimization (*PKCS #1 v1.5*) is that it uses a random padding string to pad on the message first then does encryption, which can improve the randomness. Further more, *PKCS #1 v2.0* proposed OAEP (Optimal Asymmetric Encryption

Padding), which makes RSA much stronger. However, since there was some bugs had happened on my program, the implementation of OAEP was in stillbirth. But you can still see it from *./deprecate/oaep.rs*.



```
                    +----------+------+--+-------+
               DB = |  lHash   |  PS  |01|   M   |
                    +----------+------+--+-------+
                                   |
          +----------+             |
          |   seed   |             |
          +----------+             |
                |                  |
                |-------> MGF ---> xor
                |                  |
       +--+     V                  |
       |00|    xor <----- MGF <-----|
       +--+     |                  |
         |      |                  |
         V      V                  V
       +--+----------+----------------------------+
 EM =  |00|maskedSeed|          maskedDB          |
       +--+----------+----------------------------+
```

Figure 1: EME-OAEP Encoding Operation

Figure 3: EME-OAEP Encoding Operation

6 Reference

[1] Rivest, R.; Shamir, A.; Adleman, L. (February 1978). "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (PDF). Communications of the ACM. 21 (2): 120–126. CiteSeerX 10.1.1.607.2677. doi:10.1145/359340.359342. S2CID 2873616.
[2] M. Bellare, P. Rogaway. Optimal Asymmetric Encryption – How to encrypt with RSA. Extended abstract in Advances in Cryptology - Eurocrypt '94 Proceedings, Lecture Notes in Computer Science Vol. 950, A. De Santis ed, Springer-Verlag, 1995.
[3] "Encryption Operation". PKCS #1: RSA Cryptography Specifications Version 2.2. IETF. November 2016. p. 22. sec. 7.1.1. doi:10.17487/RFC8017. RFC 8017. Retrieved 2022-06-04.
[4] Johnson, J.; Kaliski, B. (February 2003). Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1. Network Working Group. doi:10.17487/RFC3447. RFC 3447. Retrieved 9 March 2016.