实验指导书 4

复习RSA算法

ALGORITHM 12.25
RSA key generation GenRSA

Input: Security parameter 1^n
Output: N, e, d as described in the text

$(N, p, q) \leftarrow \mathsf{GenModulus}(1^n)$
$\phi(N) := (p-1) \cdot (q-1)$
choose $e > 1$ such that $\gcd(e, \phi(N)) = 1$
compute $d := [e^{-1} \bmod \phi(N)]$
return N, e, d

CONSTRUCTION 12.26

Let GenRSA be as in the text. Define a public-key encryption scheme as follows:

- Gen: on input 1^n run $\mathsf{GenRSA}(1^n)$ to obtain $N, e,$ and d. The public key is $\langle N, e\rangle$ and the private key is $\langle N, d\rangle$.
- Enc: on input a public key $pk = \langle N, e\rangle$ and a message $m \in \mathbb{Z}_N^*$, compute the ciphertext
$$c := [m^e \bmod N].$$
- Dec: on input a private key $sk = \langle N, d\rangle$ and a ciphertext $c \in \mathbb{Z}_N^*$, compute the message
$$m := [c^d \bmod N].$$

RSA 加密

这里因为采用cryptopp进行加密1G文件会耗时72分钟，并且加密后文件扩大了500MB，怀疑算法框架优化问题，因此此处换成openSSL框架进行RSA加密，虽然消耗时间也很长但是减少了30分钟的时间，且文件仅仅增大了几十k，相对来说更加合理。

RSA加密算法主要分为两步，产生密钥以及加密，但是需要注意的是加密密钥的大小决定了加密的文字长度，所以大文件只能分块来加密。

首先是RSA的密钥生成：

```
#define KEY_LENGTH 2048
#define PUB_KEY_FILE "E:\\CQUverify\\CQUVerify\\pubkey.pem"    // 公钥路径
#define PRI_KEY_FILE "E:\\CQUverify\\CQUVerify\\prikey.pem"    // 私钥路径
void GenerateRSAKey(std::string& out_pub_key, std::string& out_pri_key)
{
    size_t pri_len = 0; // 私钥长度
    size_t pub_len = 0; // 公钥长度
    char* pri_key = nullptr; // 私钥
    char* pub_key = nullptr; // 公钥

    // 生成密钥对
    RSA* keypair = RSA_generate_key(KEY_LENGTH, RSA_3, NULL, NULL);
```

```
    BIO* pri = BIO_new(BIO_s_mem());
    BIO* pub = BIO_new(BIO_s_mem());

    // 生成私钥
    PEM_write_bio_RSAPrivateKey(pri, keypair, NULL, NULL, 0, NULL, NULL);
    //生成公钥
    PEM_write_bio_RSA_PUBKEY(pub, keypair);

    // 获取长度
    pri_len = BIO_pending(pri);
    pub_len = BIO_pending(pub);

    // 密钥对读取到字符串
    pri_key = (char*)malloc(pri_len + 1);
    pub_key = (char*)malloc(pub_len + 1);

    BIO_read(pri, pri_key, pri_len);
    BIO_read(pub, pub_key, pub_len);

    pri_key[pri_len] = '\0';
    pub_key[pub_len] = '\0';

    out_pub_key = pub_key;
    out_pri_key = pri_key;

    // 将公钥写入文件
    std::ofstream pub_file(PUB_KEY_FILE, std::ios::out);
    if (!pub_file.is_open())
    {
        perror("pub key file open fail:");
        return;
    }
    pub_file << pub_key;
    pub_file.close();

    // 将私钥写入文件
    std::ofstream pri_file(PRI_KEY_FILE, std::ios::out);
    if (!pri_file.is_open())
    {
        perror("pri key file open fail:");
        return;
    }
    pri_file << pri_key;
    pri_file.close();

    // 释放内存
    RSA_free(keypair);
    BIO_free_all(pub);
    BIO_free_all(pri);

    free(pri_key);
    free(pub_key);
}
```

为了减少IO操作，采用每次从文件读取1mb到缓冲区，对这1mb每次截取一部分进行加密。

下面成为为加密一段超过密钥长度的文件程序：

```
std::string RsaPriEncrypt(const std::string& clear_text, std::string& pri_key)
{
    std::string encrypt_text;
    BIO* keybio = BIO_new_mem_buf((unsigned char*)pri_key.c_str(), -1);
    RSA* rsa = RSA_new();
    rsa = PEM_read_bio_RSAPrivateKey(keybio, &rsa, NULL, NULL);
    if (!rsa)
    {
        BIO_free_all(keybio);
        return std::string("");
    }

    // 获取RSA单次可以处理的数据块的最大长度
    int key_len = RSA_size(rsa);
    int block_len = key_len - 11;    // 因为填充方式为RSA_PKCS1_PADDING, 所以要在
key_len基础上减去11

    // 申请内存：存贮加密后的密文数据
    char* sub_text = new char[key_len + 1];
    memset(sub_text, 0, key_len + 1);
    int ret = 0;
    int pos = 0;
    std::string sub_str;
    // 对数据进行分段加密（返回值是加密后数据的长度）
    while (pos < clear_text.length()) {
        sub_str = clear_text.substr(pos, block_len);
        memset(sub_text, 0, key_len + 1);
        ret = RSA_private_encrypt(sub_str.length(), (const unsigned
char*)sub_str.c_str(), (unsigned char*)sub_text, rsa, RSA_PKCS1_PADDING);
        if (ret >= 0) {
            encrypt_text.append(std::string(sub_text, ret));
        }
        pos += block_len;
    }

    // 释放内存
    delete sub_text;
    BIO_free_all(keybio);
    RSA_free(rsa);
    return encrypt_text;
}
```

每次读取的1mb文字就采用上面的算法进行加密，在此之前每次还需要对文件进行读取到缓冲区的操作，其中最后一次需要特殊处理，因为最后一次不一定还剩下1mb可以读取。

程序如下：

```
void RSA_encrypt_file() {
    string public_key;
    string private_key;
    GenerateRSAKey(public_key, private_key);

    clock_t start, end;
    //明文文件
    char path_in[100] = "E:\\CQUverify\\CQUVerify\\1gb.data";
```

```
    //密文文件
char path_out_en[100] = "E:\\CQUVerify\\CQUVerify\\1gb_out_en_openssl.data";
    int size = 1048576;


    ifstream in(path_in, ios::binary);
    in.seekg(0, in.end);
    int filesize = in.tellg();//获取文件长度
    cout << "file size:" << filesize << endl;
    int readsize = size;//读取文件时的读取长度
    in.seekg(0, in.beg);//将明文文件指针再指向开头
    ofstream out(path_out_en, ios::binary | ios::app);

    bool flag = true;
    start = clock();
    while (flag) {
        char* temp = new char[size];
        int true_size = 0;
        //如果文件剩余未读取大小小于每次读入的大小，则temp指针只分配filesize的大小，并标记
flag表示这次加密后就不再运行。
        if (filesize <= readsize) {
            in.read(temp, filesize);
            true_size = filesize;
            flag = false;
        }
        else {
            in.read(temp, readsize);
            true_size = readsize;
        }
        string res=RsaPriEncrypt(string(temp,true_size),private_key);
        out.write(res.c_str(), res.length());//密文写入文件
        filesize -= readsize;
    }
    end = clock();
    cout << "cost time:" << (end - start) << endl;
    return;
}
```

DES加密文件:

因为DES不需要文件切割，所以采用cryptopp进行尝试加密，cryptopp提供文件加密的API，因此DES加密文件十分简洁。由于只是进行测试加密速度，所以密钥采用之前的弱密钥进行测试。

```
void DES_encrypt(unsigned char key[DES::DEFAULT_KEYLENGTH],const string&
file_in,const string& file_out) {
    ECB_Mode<DES>::Encryption cipher{};
    cipher.SetKey(key,DES::DEFAULT_KEYLENGTH);
    ifstream in(file_in, ios::binary);
    ofstream out(file_out, ios::binary);
    FileSource(in,true,new StreamTransformationFilter(cipher,new
FileSink(out)));//文件的加密API
    return;
}

void DES_File_Test() {
```

```
    char path_in[100] = "E:\\CQUverify\\CQUVerify\\1gb.data";
    char path_out_en[100] = "E:\\CQUverify\\CQUVerify\\1gb_out_en_DES.data";
    clock_t start = clock();
    DES_encrypt(weakKetSets[0], path_in, path_out_en);
    clock_t end = clock();
    cout << "cost time of encrypting a file by DES alg:" << (end - start) <<
endl;
    return;
}
```

大数运算

使用Cryptopp框架下的大整数进行实验。

首先需要生成n

代码如下：

```
AutoSeededRandomPool rng;//随机数种子
Integer p,q;
//生成大素数
do
{
    p.Randomize(rng, 1024);
} while (!IsPrime(p));
do {
    q.Randomize(rng, 1024);
} while (!IsPrime(q));
Integer phi = (p - 1) * (q - 1);//n的欧拉数
Integer n = p*q;
```

采用do-while循环来检查生成的随机大整数是否是素数。

然后需要生成e，使得它和n的欧拉数互素

```
Integer e;//0-2^1024范围生成数
Integer min = 1;
do {
    e = Integer(rng, min, phi);
} while (!(GCD(e,phi)==1));//0-phi的与phi互素的数
```

同样使用do-while循环，使用cryptopp的GCD API判断是否互素。

最后是随机生成x,并进行运算

```
Integer x = Integer(rng, 1024);
Integer res=ModularExponentiation(x,e,n);
```

这里直接调ModularExponentiation的API运算即可。

完整源码如下:

```
void TestInteger() {
    AutoSeededRandomPool rng;
    clock_t start, end, middle;
    start = clock();
    Integer p,q;
    //生成大素数
    do
    {
        p.Randomize(rng, 1024);
    } while (!IsPrime(p));
    do {
        q.Randomize(rng, 1024);
    } while (!IsPrime(q));
    Integer phi = (p - 1) * (q - 1);//n的欧拉数
    Integer e;//0-2^1024范围生成数
    Integer min = 1;
    do {
        e = Integer(rng, min, phi);
    } while (!(GCD(e,phi)==1));//0-phi的与phi互素的数
    Integer x = Integer(rng, 1024);
    Integer n = p*q;
    middle = clock();
    Integer res=ModularExponentiation(x,e,n);
    end = clock();
    cout << "------------------" << endl;
    cout << "x:" << x << endl;
    cout << "the bits of x:" << x.BitCount() << endl;
    cout << "------------------" << endl;
    cout << "e:" << e << endl;
    cout << "the bits of e:" << e.BitCount() << endl;
    cout << "------------------" << endl;
    cout << "n:" << n << endl;
    cout << "the bits of n:" << n.BitCount() << endl;
    cout << "------------------" << endl;
    cout << "res:" << res << endl;
    cout << "the bits of res:" << res.BitCount() << endl;
    cout << "------------------" << endl;
    cout << "the time costed by generating nums is:" << middle - start << endl;
    cout << "the time costed by calculating is:" << end - middle << endl;
    cout << "the whole time is:" << end - start << endl;
    return;
}
```

素数判定

1. 试除法
2. AKS素性测试
3. Miller-Rabin算法
4. n Solovay-Strassen算法