实验二

（密码学与网络安全课程报告）

姓　　名：肖文韬

学　　号：2020214245

专 业 方 向：电子信息（计算机技术）

邮　　箱：xwt20@mails.tsinghua.edu.cn

二〇二一年四月十日

第 1 章　实验介绍

RSA 加密算法是应用最广泛的公钥加密算法，本次实验实现基于 RSA 算法的加解密以及数字签名功能，包含以下 4 种操作：生成密钥对、公钥加密、私钥解密和数字签名。本次实验中密钥长度为 2048 比特。实验使用 Rust 语言实现 RSA-2048 的密钥生成，加密解密，以及数字签名操作。

实验目的：

1. 熟悉 RSA 公钥加密算法的思路
2. 学习 RSA 实现上的技巧
3. 学习 rust 语言的基本用法

实验平台：

1. rust 语言

第 2 章　实验内容

2.1　生成密钥对

密钥对的生成过程包括选取随机数，对随机数进行素性测试，根据素数 p 和 q 计算 n，随机选择和 n 的欧拉函数互质的 e，计算 e 的逆元 d。选取的大素数 p 和 q 应当满足现有的安全性要求，且至少使用两种不同的算法进行素性测试，请在实验报告中说明你选择参数和算法的安全性以及效率。选取的 e 同样应当满足安全性要求，至少使用两种不同的算法进行计算逆元 d。

2.1.1　素性测试

本实现采用了两种最主流的概率素性测试算法：

1. **Miller-Rabin 测试**, 作为费马定理的扩展，每一轮 MR 测试的伪素数的可能性为 $\frac{1}{4}$，所以 k 轮通过仍然是伪素数的可能性为 4^{-k}。复杂度 $O(k \log^2 n)$ (k 为测试轮数)。
2. **Baillie−PSW 测试**, 结合了 Miller-Rabin 测试和强 Lucas 概率测试。复杂度为 $O(log^2 n)$，低于 MR 测试。

值得一提的是，素性测试的一些优化技巧：

1. 首先测试是否是常用的小素数的倍数，具体来说，RSA-2048 会首先判断随机数是否是自然数中前 384 个素数的倍数。该技巧和 384 的来源参考的是 OpenSSL 最新代码的 `bn_prime.c#L74:12`。
2. 如果当然随机数不是素数，会对随机数 +2 再判断是否是素数，因为按照素数定理，对于 RSA-2048，连续的数中出现素数的概率为 $\frac{1}{\log(2^{2048})} \approx \frac{1}{1418}$。最坏情况下尝试 700 多次就一定会遇到素数。

miller rabin 测试的代码：

```rust
fn miller_rabin_test(rnd: &Integer, iteration: u8, base_2: bool) {
    let n = Integer::from(rnd);
    let mut d: Integer = Integer::from(&n - 1);
    let mut r = 0;
    while !(&d.get_bit(0)) { d >>= 1; r += 1; }
    let mut rng = RandState::new();
    'witness_loop: for _ in 0..iteration {
        let mut n_sub = Integer::from(&n - 2u8);
        let a = if base_2 {
            Integer::from(2)
        } else {
            Integer::from(n_sub.random_below_ref(&mut rng)) + 2u8
        };
        let mut x = a.pow_mod(&d, &n).unwrap();
        n_sub += 1u8;
        if &x == &1 || &x == &n_sub {
            continue 'witness_loop;
        }
        for _ in 1..r {
            x = x.pow_mod(&Integer::from(2), &n).unwrap();
            if &x == &n_sub {
                continue 'witness_loop;
            }
        }
        return false;
    }
    true
}
```

因为囿于篇幅，具体的代码注释参见源码。

2.1.2　模逆运算

本实现中素数 p 和 q 均为 2048 位，是目前主流的 RSA-2048 实现，密钥长度符合安全要求。

对于 e 的选择，过小的 e（例如 3）会存在安全问题，同时短比特长度和小的 Hamming 权重能够使得加密的效率更加高，目前 OpenSSL 以及其他实现广泛采用的是 65537 (0x10001). 本实现参考主流实现，e 的选择也是 65537.

d 作为 e 的模 $\phi(n)$ 逆元，因为 n 为 4096 位，所以 d 的强度也能够得到保证。

本实现共有两种模逆的算法实现：

1. **扩展欧几里得算法**，算法效率 $O(2\log_{10}(\phi(n)))$（除法运算）。gcd 的计算复杂度推导很复杂，具体可以参考 TAOCP。
2. **平方幂算法（binary exponentiation）**，算法效率 $O(\log(\phi(n)))$。但是该算法只

适用于 $\phi(n)$ 为素数的情况。

两种模逆的算法实现代码：

```
fn modular_inverse(e: &Integer, phi_n: &Integer, method: &str) {
    if method == "extend_gcd" {
        let mut t = Integer::from(0);
        let mut newt = Integer::from(1);
        let mut r = Integer::from(phi_n);
        let mut newr = Integer::from(e);
        let mut quotient = Integer::new();
        let mut tmp = Integer::new();
        while newr.significant_bits() != 0 {
            quotient.assign(&r / &newr);
            tmp.assign(&quotient * &newt);
            tmp *= -1; tmp += &t; t.assign(&newt);
            newt.assign(&tmp);
            tmp.assign(&quotient * &newr);
            tmp *= -1; tmp += &r; r.assign(&newr);
            newr.assign(&tmp);
        }
        if r > 1 { panic!("e is not invertible!"); }
        if t < 0 { t += phi_n; }
        return t;
    } else if method == "binary_exp" {
        if baillie_psw(phi_n, true) {
            println!("phi_n is prime, use binary_exp");
            return Integer::from((&e).pow_mod_ref(
                    &Integer::from(phi_n - 2), &phi_n).unwrap())
        } else {
            return Integer::from((&e).invert_ref(&phi_n).unwrap());
        }
    }
}
```

2.1.3　密钥生成

密钥生成就是通过两个大随机素数 p, q 相乘计算出 n，然后再计算 $\phi(n) = (p-1)(q-1)$，然后通过 $e = 65537$ 计算它的模逆 $de = 1 (\mathrm{mod}\ \phi(n))$。代码如下：

```rust
fn rsa_key_phase1() -> (Integer, Integer, Integer) {
    let p = generate_prime("miller_rabin");
    let q = generate_prime("miller_rabin");
    let n = Integer::from(&p * &q);
    let phi_n = (p - 1) * (q - 1);
    let e = Integer::from(65537);
    (n, phi_n, e)
}

fn rsa_key_pair(method: &str) -> (Integer, Integer, Integer) {
    let (n, phi_n, e) = rsa_key_phase1();
    let d = modular_inverse(&e, &phi_n, method);
    (n, e, d)
}
```

2.1.4　运行结果

```
Generate RSA key pair:
n=330597022001488218028026386041089065564042348056390097411303640870564769395250555193704947392043686406505673104320320925477772952093994720955620631496350622361369496717653239758468804230972385780522088334959385925538728844672794325279251021396457595225249762833810592123362140151460864381407794837199369775614159633417487592812226133409415546889397853728597183424754542191780977581863746930272948892987064718896630991058890171931434736388635188678637929984476811191438763786966694566386535797175589189988347397630664595603127520265422270513807408423578454263464683693770324188957628139852492536462394158074081316016368692252130248271257752604903977763364881037032164783933750478843110695922005692302860698053154996438027678157551945587518630984976155662499534661014375733449949893674980135563247426495502409293711963159278912489161013239342419104333182822205884668786400307922953191392914281955170555083109857208003010345048119686910482469657045287461870725886749758201286981328337066136500724567232485291011914926933774600860891513215623446824864801860420575288765054468347417025052519122717051209560797433654996072906292045135140619438672322170369115487221616441670310883112997012440235016963196094549505459005986253591278746517 7,
e=65537,
d=322990016658273785985267886412635484977952416248876230940512394850258504655515171559542458100998904388698776922906538726786644557267290049713405151503423012941943139071587336748630530327981611748188790973024620007420529917383376533153869778125238928236381764597839477551713924152399236592271784939242013591349617814620883337778924923472346106858387254797119717194109600045771921530721811880614685818192881527818228459841783540318595350315436963083303333707520342057734811267912124523000925202642378473256690905217341932555941323526416333149019420259388330995931363997776587284662826795881199207259586284930241749329873002531271603980416300798851360906850057693434618505157491273461422736631527846270599332478421173482411933372216849656732524186924162772310232183974930342792075290926565045395809951381358241086072541599914237052793121267209943449333674041393872507713070346988734399229353047878438300605842753460069274888425511791216482233244197282218630838133798877496244151526927067322818968885989289753450176727111028137243325339730590616289394815520900927584042216888600489560101497789535388900010504659084587215141485207653123124090900397059063505566824885617666605541712302087595650864011232897874858813059685604611998726612349 3
```

图 2.1　密钥生成运行结果

RSA 的实现跟之前 AES 的实现都使用 cargo 组织项目代码。编译运行的代码：

```
cd target/release && cargo build --release && ./crypto
```

密钥生成的 n, e, d 如图 2.1 所示，其中 n 是两个 2048 位随机大素数 p, q 的乘积, e 则是固定的。

2.2　公钥加密

本题使用公钥对明文进行加密，明文为’Cryptography and Network Security; 学号; 姓名拼音’，使用自己的学号和姓名拼音替代对应位置字符串。加密过程包括

分块，字符串转数字，公钥加密。分块将待加密的字符串分解成多个块，对于每个块分别加密。每个块的长度取决于选择的大素数 p 和 q 的大小，具体来说，比特串的长度应当小于 log2n。块长度不足的部分需要使用填充字符补齐。字符串转数字可以先转化成比特串，再转化为数字，或者采用自行规定的转化规则。公钥加密部分请使用快速幂算法计算密文。输出每一步的结果。

2.2.1　快速幂算法

快速幂算法就是对计算 m^e 的优化，就是把 e 以二进制表示，复杂度 $O(2\log(n))$. 实现代码如下：

```
fn quick_pow_mod(mut m: Integer, e: &Integer, n: &Integer) -> Integer {
  m %= n;
  let mut ans = Integer::from(1);
  let mut e_curr = Integer::from(e);
  let two = Integer::from(2);
  while e_curr.significant_bits() != 0 {
      if e_curr.get_bit(0) {
          ans *= &m; ans %= n;
      }
      m.pow_mod_mut(&two, n).unwrap();
      e_curr >>= 1;
  }
  ans
}
```

2.2.2　加密填充字符

首先计算 $\log_2 n$ 作为加密块的大小，因为 n 为 4096 位，也就是每一块为 512 字节。对于明文的最后一块不足 512 字节的话，在最后面填充 0. 对于每一块的加密，就是把这一块当做一个大整数 m, 然后计算 $m^e \bmod n$. 其中 e, n 为公钥对。代码如下：

```rust
fn rsa_encrypt(key: (&Integer, &Integer), plaintext: &str) -> Vec<u8> {
    let (n, e)  = key;
    let block_size = (n.significant_bits() as f64 / 8.0).ceil() as usize;
    let mut padded_bytes: Vec<u8> = plaintext.to_string().into_bytes();
    let mut pad_size = padded_bytes.len() % block_size;
    if pad_size != 0 {
        pad_size = block_size - pad_size;
    }
    println!("padding_size={}", pad_size);
    padded_bytes.extend(vec![PAD_BYTE; pad_size]);
    for block in padded_bytes.chunks_mut(block_size) {
        let mut num_str = block.iter()
            .map(|x| format!("{:02X}", x))
            .fold(String::from(""),
                  |res: String, curr: String| res + &curr
            );
        let mut m = Integer::from(Integer::parse_radix(
            num_str, 16).unwrap());
        m = quick_pow_mod(m, e, n);
        let mut digits = m.to_string_radix(16);
        if digits.len() % 2 == 1 { digits.insert(0, '0'); }
        for idx in 0..block.len() {
            block[idx] = u8::from_str_radix(&digits[
              (idx*2)..(idx*2+2)], 16).unwrap();
        }
    }
    padded_bytes
}
```

2.2.3　运行结果

```
m=27516042870337128773354891295608776921451761309052707600405591559206497195216853334628701873239242646935967360557363612086787
6099070839678936358474685003138981656493805960724669896277505567425662084902592791338929590323768754746558448081735574910141345
7329707782241770392911200618878243489773546245779480553544596636948749886253948436687477111535694863180151270805324658702119023
8897855665699158102890608464554340661834901326561641494227368134973735732554957861179567961875395373140633077221389871906975382
2136241853893792050615850999056621419166059582088368251126581268175753432890213994787387079983691684410903634315860567663380813
8234644431738511957967233549269253585610202723294529434801730584232804019948317466784853448606906206054250410614825362965348590
4355908977770430819857726932333077722871639034789093458463211120528160182805162141412108324807676458275966251495900610216961082
0631306488228301888579587383567828096804264414776339179737806350364747667531037457741121749929895550860133859035917168974584435
8851865298094315815569430002328211943051419016419723136347438954896599778767616427060664746731065101598988857050152488193507484
8514091662081156882112689128263078283686738013490101447474452497419853240039695958860317081 6
RSA encryption of Cryptography and Network Security; 2020214245; 肖文韬 (Wentao Xiao) 🎉🚀
Block 1:
60 8E 07 D8 C2 D9 A3 9B 60 FF 77 78 E6 BD AF 7D AD AE 97 7A BE 4B 8E 65 41 81 59 B6 23 C7 AF E4 43 5B 42 7E B5 70 72 A0 2A 2D 2
B AC 75 FA 2B 70 AF F7 2F 72 90 FA 89 83 F2 CB C9 F3 11 CB E2 F8 EF F0 AD 13 29 78 A3 FE 1A 25 E1 1B 6D 24 D1 95 DC 01 30 44 73
 C0 95 6C D9 D6 57 2F 50 5C CA 57 BB 88 FD 3F 49 89 49 55 CC 8F D5 86 7E AC DA DE D0 04 43 94 3C 17 01 6F 05 A2 19 8C 7A 4D CE
0E 46 0A 19 3E D3 D8 B2 97 D6 99 5F 90 0B 17 6D A1 98 2F 0C 0F 9A F0 50 97 A7 5B 30 75 A4 35 22 C5 DD 90 A7 17 43 48 1C 2E 32 8
3 A4 17 00 ED 44 26 F4 B5 83 AA 65 3F 76 B4 85 F0 85 DA 9A 3C 3F 85 2D A0 61 E7 C7 71 43 33 F7 53 45 E7 FD 64 99 4F C6 F7 28 78
 CD A5 5A DA 3A 4B D5 0E B5 A4 89 E1 34 93 E3 13 39 1E 4A 02 44 E5 CB 81 FC 24 F1 E1 19 F2 63 44 77 20 79 3A DD 73 0E 1D 3F 83
7C A7 6D 88 DF 8D 83 CB 7D 1A 33 FB 0C 63 90 2E 91 13 C1 ED 61 CE 0D 7E 8C D9 61 AC 35 F3 89 DD B3 32 84 24 A9 65 68 65 55 86 6
7 63 63 7C 81 23 47 00 2A A9 E1 84 F5 12 AB A0 84 9F D7 04 4F B4 F6 3E A3 E6 BF 78 75 42 56 D3 34 2B 6F 27 EF BB 6F 3C 2B 9D 96
 CA 7F 52 A2 8D 7F F6 0E 0A F9 F3 C8 A6 D6 23 2E 39 43 57 D2 70 C3 93 6C 90 17 8B F1 ED DB 5E 3F 25 4B 0D 01 02 CB 0D BE C9 3C
0B D4 24 4E 95 BF 30 7E 34 69 5D 56 01 96 D5 60 D3 AB A7 A2 10 FB 58 E7 E9 7E 7D 36 ED FE 4C 80 C7 B5 57 70 B9 E0 26 38 3A 7C 3
8 36 34 31 B4 66 80 68 B9 3F AC FE 25 1D A8 B5 4F CF A9 6F 3D E9 7E 9C BD 18 5F C1 33 1A 96 03 DD 91 BF CB 38 AD 80 4B 36 5C 67
 02 B9 2B 41 FD B5 56 25 47 6A 79 1F 7D 1E 15 A2 11 1A 54 E4 03 80 8B 64 15 EE 5F 6D AF F5 E3 8B 0F CA F1 FA 0A DE 4E 4E 8B 0D
FD 97 66 74
```

图 2.2　公钥加密运行结果

运行结果如图 2.2所示，明文为“Cryptography and Network Security; 2020214245; 肖文韬 (Wentao Xiao) □□”，加密结果为一个加密块。值得一提的是，虽然明文对应的块大部分内容都是填充字符，但是加密块就是全满（没有填充字符）的数据。

2.3　私钥解密

私钥解密是公钥加密操作的逆操作，首先需要将比特串转化为数字，然后对数字使用私钥 d 进行解密，同样建议使用快速幂等算法计算，然后将数字还原成字符串，输出每一步的结果。

2.3.1　代码说明

因为 RSA 加密解密差不多一模一样，而且都能简单。解密过程就是就是计算 $c^d \bmod n$，其中 c, d, n 分别为密文（块），私钥对。去除填充字符的过程也比较简单，因为填充的字符就是 0，所以只需要把最后一块最后面连续的 0 去掉就好啦。

完整的代码如下：

```rust
fn rsa_decrypt(key: (&Integer, &Integer), cipher: &Vec<u8>) -> String {
    title("decryption");
    let (n, d) = key;
    let block_size = (n.significant_bits() as f64 / 8.0).ceil() as usize;
    let chunks = cipher.chunks(block_size);
    let num_chunks = chunks.len();
    let mut plain = String::new();
    for (idx, block) in chunks.enumerate() {
        let mut num_str = block.iter()
            .map(|x| format!("{:02X}", x))
            .fold(String::from(""),
                  |res: String, curr: String| res + &curr);
        if num_str.as_bytes()[0] == '0' as u8 {
            num_str.remove(0); }
        let mut c = Integer::from(Integer::parse_radix(
            num_str, 16).unwrap());
        println!("c={}", c);
        c = quick_pow_mod(c, d, n);
        let mut digits = c.to_string_radix(16);
        if digits.len() % 2 == 1 {
            digits.insert(0, '0'); }
        let mut c_block = vec![0u8; digits.len() / 2];
        for idx in 0..(digits.len() / 2) {
            c_block[idx] = u8::from_str_radix(&digits[(idx*2)..(idx*2+2)],
              16).unwrap(); }
        if idx == (num_chunks - 1) {
            while c_block[c_block.len() - 1] == PAD_BYTE {
                c_block.pop(); } }
        plain.push_str(&String::from_utf8(c_block).unwrap());
    }
    plain
}
```

2.3.2　运行结果

```
############################################################
#                    decryption                            #
############################################################

c=393909246560790521998242807810126023148309476704957678259765334730403419804632189253054712973767818843481982443846223708163500262882808922156857821437434463515121721184415869816990175148402563363678855040139553684595296374306576060564254801216995599518106811845521024908122572424765715137935609280623179453571552257107617530485021930065829690038929918559073939154922830824573814871413897522802389273715412753643217659298236853199848843109027140423517863173224256765700224769407708775614588022375032019409639766275717047149426434140738177119723574237159374405402812322618923486385007831878172018831481783477499508802748516983939806818582539773214361192434356958181474571093428528694735240386781959413120162764866965419691009971944930987538713303539607905013728235111700914673047404792777459193408020091951856010068014978475524518435533494054668519824559654099557453534907334197515962917103368387813327859466669438208599445858888145868365224856198684768805621510978736961557093473571457434626724463710511392929015925846163104998210221615110087257536526292769297010408656920365210270754973199886054705184935402460386337134087988990737979001979112651978259856804145314331775421874851238828681749284569752239077246006840838408536488280 20
unpad:
Block 1:
43 72 79 70 74 6F 67 72 61 70 68 79 20 61 6E 64 20 4E 65 74 77 6F 72 6B 20 53 65 63 75 72 69 74 79 3B 20 32 30 32 30 32 31 34 32 34 35 3B 20 E8 82 96 E6 96 87 E9 9F AC 20 28 57 65 6E 74 61 6F 20 58 69 61 6F 29 20 F0 9F 8E 89 F0 9F 9A 80

RSA decryption: Cryptography and Network Security; 2020214245; 肖文韬 (Wentao Xiao) 🎉🚀
```

图 2.3　私钥解密运行结果

运行结果如图 2.3所示，解密出来的明文为“Cryptography and Network Security; 2020214245; 肖文韬 (Wentao Xiao)”。解密出来的块当中大部分为填充字符。

2.4　数字签名

签名的内容可以和题目 2 中使用的字符串相同，也可以是从文件中导入的任意一个字符串。

签名过程主要包括消息散列，私钥加密，验证过程包括公钥解密，对比验证。首先需要将字符串压缩成哈希值，在这一环节中允许使用现有的哈希函数库，Python 中的哈希函数库为 hashlib。私钥加密和公钥解密的过程同题目 2 和 3，使用题目 1 中生成的密钥对，注意要使用私钥加密，公钥解密，与题目 2 和 3 中是相反的，输出每一步的结果。

2.4.1　代码说明

值得注意的是，RSA 在传输加密的数据的时候，发送者要用接受者的公钥加密消息，然后用发送者自己的私钥签名（加密）明文消息与密文一起发送给接受者。因为数字签名就是加密，只不过用的秘钥不一样，所以代码实现还是很简单的。代码如下：

```rust
fn rsa_sign(plaintext: &str) -> (Integer, Integer, Vec<u8>, String) {
    let (n, e, d) = rsa_key_pair("extend_gcd")
    let cipher = rsa_encrypt((&n, &d), plaintext);
    let mut hasher = DefaultHasher::new();
    plaintext.hash(&mut hasher);
    let hash: u64 = hasher.finish();
    let sign = quick_pow_mod(Integer::from(hash), &d, &n)
      .to_string_radix(16);
    (n, e, cipher, sign)
}

fn rsa_check_sign(cipher: &Vec<u8>, n: &Integer, e: &Integer,
                  sign: &str, plain: &str) -> bool {
    let decrypted = rsa_decrypt((n, e), cipher);
    let mut hasher = DefaultHasher::new();
    decrypted.hash(&mut hasher);
    let hash: u64 = hasher.finish();
    let hash_in_sign: u64 = quick_pow_mod(
        Integer::from(Integer::parse_radix(sign, 16).unwrap()),
        e, n).to_u64().unwrap();
    hash == hash_in_sign
}
```

2.4.2　运行结果

```
sign:
2e0e35f8dbd69483dba5bc72d003c90cf7fe664706d1fd13729633dff96e8bed0caee48a80e7fae82c032b478b0603c8c2f8286aa79b8f7a89be1929d0b91f5
3cab83a53cbf3b9db36d94e501ead61520e1679e96b24615a129a22d6b98fbcfc10567af6b639caad4659ec2983aca0dd3b9179530aa4b42121bda0b1cb045b
770ef6fcdbd5b7c0cf0986e295767d16f6575b5d6031898e8932892353e43422893892d7d75724c88c6fcdfcc4b5816c80b5d7dd8729dccaa5741a9c478c056
509bac9bd17b7bc608c9b7db93fca7b4ee5fa107d32f3d7de144c528d1ed6f02c56ca34552ad5f2b33719f457ef33d828bd2e071f7f0e0fec849887b0c1b328
c2b10f8d12feae38278d89a2738d11a0d22dc0a7f44d2beea3bdeb79829e4e261d80a200db5e164a42e963264d157639a65e11d496b717e04f55a37f00e197f
493971ffd60d81af9aebb895ac17a0acaf476c4d4e03ebb012bb2509efad619ba82d85a5804edf116a00d8daa69926b804c7b69ece57e4a9ca99a294d0d321f
5fa212ad9322b5cb7f67f3d1cf6757b7d238f5a6bb786343562e2810529705d4c7d2a4c41d121d52cc5941db485c37e7714f6766aa7022fe470b97e66a32e2e
1d5f4a8122837f849f9dd53a166bb74c736acc6b77d7f46631dd4ba60420d036d6394d25c6e24ccb26634e2e494be7c46c7ba1bf4cbb757d4404bf69dda992c
b4594b05
```

图 2.4　数字签名运行结果

对消息“Cryptography and Network Security; 2020214245; 肖文韬 (Wentao Xiao)”的 RSA 数字签名，运行结果如图 2.4所示。