Chapter 6. COMPUTATIONAL PROBLEMS BEHIND PUBLIC-KEY CRYPTOSYSTEMS, BIGINTEGERS IN JAVA

In this section we pay attention to computational problems that stay at the core of public key cryptosystems, RSA in particular. We exemplify computational problems with the help of the *BigInteger* class from Java. Rather than briefing through the capabilities of this class, we take a problem based approach in which we try to underline the math behind cryptosystems such as the RSA (pointing on issues that potentially cause insecurity). A shortcoming of this section is that we do not describe the particular algorithms behind these computations, however some of the algorithms are described during the lectures and here we try to fix the notions by playing with numbers.

6.1 THE JAVA BIGINTEGER CLASS

The Java *BigInteger* class allows working with arbitrary precision integers. There is virtually no limit on their size, except for the memory available. However, in public key cryptosystems we usually work with integers that are in the order of several thousands of bits, e.g., 1024-4096 in case of the RSA, so you should imagine this as the practical size that we target. To initialize a *BigInteger* is fairly simple, the constructor of the class can also take strings, for example,

BigInteger two = new BigInteger("2");

creates a *BigInteger* with value 2. You can initialize the integer with a value of your choice, e.g.,

BigInteger exponent = new BigInteger("65537");

Then operations are simply performed by calling the related methods. For example if you want to compute an exponentiation $2^{65537} \mathrm{mod} 3$ simply call:

BigInteger result = two.modPow(exponent, new BigInteger("3"));

In Table 1 we summarize the arithmetic operations and the equivalent Java BigInteger's methods.

Arithmetic Operation	**Java BigInteger Method**
additions and subtractions (+, -)	subtract(BigInteger val) add(BigInteger val)
multiplications and divisions (*, /),	multiply(BigInteger val) divide(BigInteger val) divideAndRemainder(BigInteger val) mod(BigInteger m) remainder(BigInteger val)
comparisons (<, >)	compareTo(BigInteger val) max(BigInteger val) min(BigInteger val)
exponentiation and modular exponentiation, a^x	modPow(BigInteger exponent, BigInteger m) pow(int exponent)
greatest common divisor (GCD) and multiplicative inverse, i.e., x^{-1}	gcd(BigInteger val) modInverse(BigInteger m)
primality testing	isProbablePrime(int certainty) probablePrime(int bitLength, Random rnd)

Table 1. A summary of arithmetic operations and the corresponding methods in Java

6.2 SOLVED EXERCISES

The private exponent reveals the factorization of the modulus. This is a commonly known property of the RSA. It is also the reason for which a modulus cannot be shared by two distinct entities even if they use distinct public exponents (since the private exponent of each of them can be used to factor the modulus and recover the private exponent of the other). This problem is also referred as the common modulus problem.

Let the following RSA key:

$$K_1 : \{n = 837210799, e = 7, d = 478341751\}$$

Show how the modulus can be factored given the private key and find the private exponent for the following key:

$$K_3 : \{n = 837210799, e = 17, d = ?\}.$$

Solution 1. *The mathematical relation between the private and public RSA exponents is the following:*

$$d \cdot e \equiv 1 \bmod \phi(n)$$

This implies that there exists a number k such that

$$d \cdot e = 1 + k \cdot \phi(n).$$

Since

$$\phi(n) = (p-1)(q-1) = p \cdot q - p - q + 1$$

It follows that

$$d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1)$$

Rearranging the terms we get

$$pq + 1 - \frac{d \cdot e - 1}{k} = p + q.$$

We know all values from the left side, except for k. *However, by closely examining the previous relation* $d \cdot e = 1 + k \cdot (p \cdot q - p - q + 1)$ *since on the right side* $p \cdot q$ *is much larger than* $-p - q + 1$ *we are not far by approximating k as:*

$$k \approx \left\lceil \frac{d \cdot e - 1}{p \cdot q} \right\rceil = \left\lceil \frac{d \cdot e - 1}{n} \right\rceil$$

In our case, starting from the already known key we get:

$$k \approx \left\lceil \frac{7 \cdot 478341751 - 1}{837210799} \right\rceil = 4$$

It follows that:

$$p + q = \frac{4 \cdot (837210799 + 1) + 1 - 7 * 478341751}{4} = 112736$$

This implies that p and q can be extracted as roots of the equation $x^2 - Sx + P = 0$ *where* $S = 112736$ *and* $P = 837210799$. *By elementary calculations, we get:*

$$\Delta = 112736^2 - 4 \cdot 837210799 = 9360562500$$

The roots follow as:

$$x_1 = \frac{112736 + \sqrt{9360562500}}{2} = 104743 \quad \textit{and}$$

$$x_2 = \frac{112736 - \sqrt{9360562500}}{2} = 7993$$

These are the factors of the modulus. Finding the second private exponent is now trivial as:

$$d = e^{-1} \bmod (p-1)(q-1) \Rightarrow d = 17^{-1} \bmod 837098064 = 246205313$$

Solution 2. *The private exponent always decrypts a message encrypted with the public one, since:*

$$\forall x \in Z_n, x = \left(x^e\right)^d \bmod n$$

Given the values from the first key we always have:

$$x = \left(x^7\right)^{478341751} \bmod 837210799$$

By multiplying with x^{-1} *and rearranging we get:*

$$x^{7\cdot 478341751-1} = 1 \bmod 837210799$$

Dividing by 2 we get:

$$\left(x^{\frac{7\cdot 478341751-1}{2}}\right)^2 = 1 \bmod 837210799$$

This means that the right quantity is a square root of 1. To eliminate the two trivial roots of 1, i.e., +1 and -1, we continuously divide the exponent until we get a non-trivial root. For example, let us fix $x = 10$ *and compute:*

$$x^{\frac{7\cdot 478341751-1}{2}} = 1, x^{\frac{7\cdot 478341751-1}{4}} = 1, x^{\frac{7\cdot 478341751-1}{8}} = 1, x^{\frac{7\cdot 478341751-1}{16}} = 562155682$$

It is easy to note that when dividing the exponent with 16 the result is no longer 1. For this final result we have:

$$cmmdc\left(562155682 - 1, n\right) = 7993 = p$$
$$cmmdc\left(562155682 + 1, n\right) = 104743 = q$$

In this way we have successfully extracted the factors of n*. The mathematical explanation is that we have:*

$$\left(x^{\frac{7\cdot 478341751-1}{16}}\right)^2 \equiv 1 \bmod n \Leftrightarrow \left(x^{\frac{7\cdot 478341751-1}{16}} - 1\right)\left(x^{\frac{7\cdot 478341751-1}{16}} + 1\right) \equiv 0 \bmod n$$

and since $x^{\frac{7\cdot 478341751-1}{16}} \neq \pm 1$ *it means that the two factors contain the prime numbers that divide* n.

Small encryption exponents. While small exponents are preferred for encryption because they result in faster operation, small exponents are known to cause insecurity. The .NET framework has the default exponent set to 65537, this should be secure, but it may be tempting to use even smaller exponents. Consider the following two 1536 and 2048 bit modules taken from the RSA challenge website http://www.rsasecurity.com/rsalabs/node.asp?id=2093

n_1=

1847699703211741474306835620200164403018549338663410171471785774910651696711
1612498593376843054357445856160615445717940522297177325246609606469460712496
2372044202226975675668737842756238950876467844093328515749657884341508847552
8298186726451339863364931908084671990431874381283363502795470282653297802934
9161558118810498449083195450098483937752272570525785919449938700736957556884
3693381277961308923039256969525326162082367649031603655137144791393234716956
6988069

n_2=

2519590847565789349402718324004839857142928212620403202777713783604366202070
7595556264018525880784406918290641249515082189298559149176184502808489120072
8449926873928072877767359714183472702618963750149718246911650776133798590957
0009733045974880842840179742910064245869181719511874612151517265463228221686
9987549182422433637259085141865462043576798423387184774447920739934236584823
8242811981638150106748104516603773060562016196762561338441436038339044149526
3443219011465754445417842402092461651572335077870774981712577246796292638635
6373289912154831438167899885040445364023527381951378636564391212010397122822
120720357

By this exercise we show that even if the factorization of these numbers is unknown (these challenge numbers were not yet factored, so it is impossible for us to know their factorization), one can still recover encrypted values in certain situations if the exponents are small. Consider that one fixes an encryption exponent $e = 2$ (this is in fact known as the Rabin cryptosystem and is a secure cryptosystem when correctly used, see the lecture material for more details) and that one encrypts the same message m once with each modulus, i.e., $c_1 = m^2 \mathrm{mod}\ n_1$, $c_2 = m^2 \mathrm{mod}\ n_2$. Given the result of the encryptions below, you are requested to find the encrypted message:

c_1=

1720824975522517857539467309146518060382842270514896093391979291030656292239
7291446654035136859446266905140522147597644944431643498057575862023479413245
6638260412096493538625812249998880361757163409597018001190001744747405240965
7500820140866171389821089899978493473235156488326073675749875367732149010528
9244104109064444335973488450882364503785143338799248614163518428477608940469
9678849571206887860878689927075639507531091535187214291140378602914898718344
7449947

c_2=
4561642280956381246774642331705575104523442518306294887033201504008906454855
8878555145972657908956759775539747979197737797768926554418702738975251318948
7102258520443358104409325508073221395545765319081041834133569912754811011387
3635190699932165850542152382657518899992710162713201334532551245793969597202
6692191157400036070478620074907493119547542465852819192370184492356694178657
6698578327560649299302223024036233077234207232288187628580786589383228234629
4300028016342171410187938861009812975635715641457865781951720724292241356964
6111551957961184286656146057704287329146644239215935313741848147782402529568
44983980

Solution. *The mistake comes from the fact that the small encryption exponent allows one to recover the message by squaring the output composed via the Chinese Remaindering Theorem (CRT). We show how this can be done in what follows. CRT implies that the following result holds:*

Iff $\begin{cases} m^2 \equiv c_1 \mathrm{mod}\, n_1 \\ m^2 \equiv c_2 \mathrm{mod}\, n_2 \end{cases}$ *then there exists a unique m modulo* $n_1 \cdot n_2$.

But message m *was encrypted with the first modululs, this means it cannot exceed 1536 de bits. Therefore the square of the message has at most 2x1536 = 3072 bits. CRT allows one to retrieve a solution modulo* $n_1 \cdot n_2$*, n.b., moreover, this solution is unique. Since the two modules have 1536 and 2058 bits respectively it means that this solution is unique for up to 1536+2048 = 3584 bits and thus message* m *can be fully recovered as square root of the value retrieved via CRT. We show how this can be done by using the CRT solution offered by Gauss. First, we compute the modular inverses:*

$n_1^{-1} \bmod n_2 =$
14310987589421656595052001273606996601993205437791295923061219355358
48932621722047996479172395205182484709925981345086823592361098676116
71392972306371407115917893215317798609151299752650828413641260143703
29155407443919355323334425193123995577457586594368899226059650898095
20834325441902847281013633185468633945939268807563205737631762188641
52671120930328170757381015429724281519672245536989347042821946707579
75832865425047290849934241209824297863258898100147349976522660848513
21478225880606620937765676470289712411515994875794907540854600946369
53345877613019417560448506378779860461784570861030428654699658479137
76536

$n_2^{-1} \bmod n_1 =$
79822742293307335384489483161431538390245026454391226276909057792674
23380579666935238128274202459805360169170453399966923117770181725592
75601482046960296591379092565607100459489204412824908723342592405951
83320111854009654964710771311177195733351955713468728607066480923161
79828221492242083990594584260123165469731743876993400374593233583932
30935565486090366472938842936241337967316093017879168325168806666801
8100504619091943607573735563055883749101636137747234 50

Then we use Gauss's solution for the CRT and compute:

$m^2 = \left(c_1 n_2 n_2^{-1} \bmod n_1 + c_2 n_1 n_1^{-1} \bmod n_2\right) \bmod n_1 n_2 =$
40248409279371781562594703314715910034847869225366381022540697914175
85602944732917136098048248669251363580202404678911735557994243268711
30957480186462490633501794023786817125556940132457090093447788623246
76312015640007845168955339378322270670911475586002444628901333977782
21666551093553199704408884732857724216266011039547799164354879332317
67021086547098554239862430087393177761620546493093153699808344190034
56501265688124968112372793434959057461901805742130368652426798835499
11146730234579613576919248080423603916547270288585014116973253480963
65792194781034259041465702725888150371192734835039659581519708126428
20437659327488904506012157371352696825838199381494305046066162771892

75083123873943549970558612016817243280483663927948950510612617417357297285884203981954351050666622817897351291284744036

Now we can extract the encrypted message as the square root of the previous message, i.e.,:

$m = \sqrt{m^2} =$
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
20062006200620062006200620062006200620062006200620062006200620062006
200620062006200620062006200620062006200620062006200620062006

Balanced vs. unbalanced RSA. The RSA version in which the two factors p and q have the same size is also referred as balanced RSA. An unbalanced version of the RSA was also proposed, it benefits from a large modulus (harder to factor, thus increased security) but still fast for decryption if this is performed via the smaller factor. Unbalanced RSA assumes the use of a small p (e.g., several hundred bits) and a larger q (e.g., several thousand bits). Only messages smaller than p are encrypted and then decryption is performed modulo p (this can be done only by the owner of the private key who is in possession of p). For correct encryption, a bound l on the size of the plaintext is made public (this does not make the scheme unsafe, it is simply the bitlength of p which does not make factorization trivial). We give a small numerical example:

Key generation:

$$p = 541, q = 104729, e = 7, l = 200$$
$$\Rightarrow n = 56658389, \phi(n) = (p-1)(q-1) = 56553120,$$
$$d = e^{-1} \bmod (p-1) = 463$$

Encryption:

$$m = 300 \Rightarrow c = 300^7 \bmod 56658389 = 18157376$$

Decryption:

$$m = 18157376^{463} \bmod 541 = 300$$

Show how a CCA2 (Chosen Ciphertext Attack) attack can be mounted such that the adversary can recover the private key. Use the previous numbers to illustrate the attack.

Solution. *The CCA2 attack assume that the adversary has unlimited access to the decryption machine, i.e., the machine accepts to decrypt messages at his choice. The adversary can cheat and encrypt a message that is larger than the bound* l*, e.g.,*

$$c = 1000^{7} \bmod 56658389 = 27641532$$

The decryption machine performs decryption according to the rules and answers with:

$$m = 27641532^{463} \bmod 541 = 459$$

Now the adversary can use this response to factor the modulus as:

$$\gcd(1000 - 459, n) = 541$$

Thus, the adversary can factor the modulus and completely break the cryptosystem. The mathematical fact behind this attack is trivial. Since $x \equiv \left(x^{e}\right)^{d} \bmod p$ *but* $x \neq \left(x^{e}\right)^{d}$ *mod p (note that ≫ p) it follows* $x - \left(x^{e}\right)^{d} = k \cdot p$ *and thus* $x - \left(x^{e}\right)^{d} \neq 0$ *which implies* $\gcd\left(x - (x^{e})^{d}, n\right) = p$ *and thus the modulus can be factored.*

6.3 FURTHER EXERCISES

1. Given the RSA encryption below with the corresponding modulus and exponent, find the encrypted message assuming that encryption was performed without padding.

n=
8716664131891073309298060436222387808362956786786341866937428783455365962391673917249574491595229207084297741464557132198229086365652604590297378403184129

```
e=3

c=
1375865583010982618632308529423371271821438577980922927124130396877
925863587827122886875024570556859122064458153631
```

2. Given the RSA key-pair below find the factorization of the modulus.

```
n=

5076313634899413540120536350051034312987619378778911504647420938544
7465177110314901155284204273194792744073890582538974985571109131603
02801741874277608327,

e=3

d=

3384209089932942360080357566700689541991746252519274336431613959029
8310118072592266557861250508877279212747197519861041620378008076415
22348207376583379547
```

3. The following fact is considered an interesting property of the RSA, although we do not know the sum of the two factors of the modulus, i.e., $p+q$, we can compute the value of $x^{p+q} \bmod n$. Figure out how this is possible and compute this value for the numbers below.

```
n=

1070064658568088584852050373529985247886583743870981513899285988324
9955498916287857233627498606657866763592788339595921943627412052904
161935201780928478603,

x=

7133764390453923899013669156866568319243891625806543425995239922166
6369992773872531940485057673409245980641693041362105818099065112161
68762318630818311867
```

4. Factor the following integer, knowing that it is the power of a prime number. What is the expected number of steps to factor an integer of this form?

```
n=
1412121655904559272391372547028455291589329729954595551258669512277
```

5. To speed-up verification time for multiple RSA signatures, rather than verifying each signature independently, one can check the following equality: $\left(\prod_{i=1}^{k} s_i\right)^e = \prod_{i=1}^{k} h(m_i)$ (this is called batch verification). This method is fast as it requires a single modular exponentiation, in contrast to k exponentiations (and indeed modular exponentiation is the most expensive computational step in verifying signatures). However, there is a problem with this method: show that given multiple signatures $\{s_1, s_2, \ldots, s_k\}$ corresponding to a set of messages $\{m_1, m, \ldots, m_k\}$ one can produce a fake set of signatures that passes the batch verification test but no signature will hold for any of the messages in particular.

6. Prove the equivalence between the following computational problems: RSA-Key, computing Euler-Phi and Integer Factorization.

Note. Since there is no method in the *Java.BigInteger* class for computing integer square roots, you may recycle the naive code below.

```
//recursively searches for the sqr root of a in interval [left, right]

private static BigInteger NaiveSquareRootSearch(BigInteger a, BigInteger left,
BigInteger right)

   {

      // fix root as the arithmetic mean of left and right

      BigInteger root = left.add(right).shiftRight(1);

      // if the root is not between [root, root+1],

     //is not an integer and root is our best integer approximation

if(!((root.pow(2).compareTo(a) == -
1)&&(root.add(BigInteger.ONE).pow(2).compareTo(a) == 1))){

         if (root.pow(2).compareTo(a) == -1) root = NaiveSquareRootSearch(a, root,
right);

         if (root.pow(2).compareTo(a) == 1) root = NaiveSquareRootSearch(a, left,
root);

      }

      return root;

   }


public static BigInteger SquareRoot(BigInteger a)

{

   return NaiveSquareRootSearch(a, BigInteger.ZERO, a);

}
```