Security Lab–Public Key Cryptography

VMware

- This lab does not require a VM.

Introduction

This lab will give you an introduction to the RSA cryptosystem, and will also demonstrate some attacks against that cryptosystem. Please download the file `RSALab.zip` from Moodle and unpack it. Inside you will find skeleton code for the exercises in this lab. The place where you should put your solutions to a given question is marked `// ------- > Your solution here < --------`. The project was created with Eclipse but should work in any IDE. But since the solution is missing, the files may not compile. In this case, you should put in some dummy code just to make it compile and run. You should replace the dummy code with your actual solutions as the lab progresses.

The RSA Cryptosystem

Historically, Diffie-Hellman key exchange (DHKE), invented in 1976 was the first "public key" algorithm known, but you couldn't encrypt with it, you could only exchange keys. The first genuine public-key cryptosystem was RSA, invented in 1977 and named after its inventors Ron *R*ivest, Adi *S*hamir and Len *A*dleman. Where DHKE uses the discrete logarithm as its basic problem, RSA uses the difficulty of factoring integers with large prime factors. For a long time, RSA was *the* public-key algorithm of choice. But RSA has been showing signs of old age: for 128-bit security, keys have had to get longer and longer as factoring has advanced. This makes RSA unsuited for embedded devices, which instead use elliptic curves because they are faster *and* have smaller keys. RSA is now very gradually being phased out and we expect that within ten to fifteen years, RSA will no longer be in use in mainstream computing.

Mathematical Preliminaries

Like Diffie-Hellman on the integers, RSA uses arithmetic modulo a large number. Here are some theorems you need to know in order to understand RSA. You can skip the proofs if you want to, but we *will* assume that you know the theorems and can apply them. The proofs may help you understand the theorems.

Theorem 1. Let m and n be two positive integers with $\gcd(m,n) = d$. Then there exist integers a and b such that $am + bn = d$.

Proof. Let us assume without loss of generality that $m < n$. Then we can *uniquely* write $n = q_1\, m + r_1$, where q_1 is the quotient and r_1 is the remainder with $0 \leq r_1 < m$. We also know that $\gcd(m,n) = \gcd(r_1,m)$. (We know this because we know that Euclid's algorithm is correct.) We now *uniquely* write

$m = q_2 r_1 + r_2$;

$r_1 = q_3 r_2 + r_3$;

…;

$r_{n-1} = q_{n+1} r_n + r_{n+1}$;

$r_n = q_{n+2} r_{n+1} + r_{n+2}$,

where $r_{n+2} = 0$ and $r_{n+1} = \gcd(m,n)$. We now write the penultimate equation as $r_{n+1} = r_{n-1} - q_{n+1} r_n$, which allows us to express $\gcd(m,n)$ in terms of r_n and r_{n-1}. We then write the third-from-last equation as $r_n = r_{n-2} - q_n r_{n-1}$, which, when we substitute this into the result from the previous step, allows us to express $\gcd(m,n)$ in terms of r_{n-1} and r_{n-2}. Continuing in this way, we eventually write $r_{n+1} = \gcd(m,n)$ as a combination of a and b, as required.

Corollary 1. Let m and n be two positive integers with $\gcd(m,n) = 1$. Then there exist integers a and b such that $am + bn = 1$.

Proof. This follows directly from Theorem 1 by setting $d = 1$.

Corollary 2. Let m and n be two positive integers with $\gcd(m,n) = 1$ and let $m < n$. Then there exists an integer a with $0 < a < n$ and $am \bmod n = 1$.

Proof. This follows directly from Corollary 1. Since $\gcd(m,n) = 1$, there exist integers a and b with $am + bn = 1$. Taking "mod n" on both sides of the equation gives $(am + bn) \bmod n = 1 \bmod n$. But $1 \bmod n$ is 1 and $(am + bn) \bmod n = am \bmod n + bn \bmod n = am \bmod n$, and the conclusion follows.

Lemma 1. Let p be a prime and let a and b be integers with $0 \le a, b < p$. Then $(a + b)^p \bmod p = a^p + b^p \bmod p$.

Proof. By the binomial theorem,

$$(a+b)^p = \sum_{k=0}^{p} \binom{p}{k} a^k b^{p-k} = a^p + b^p + \sum_{k=1}^{p-1} \binom{p}{k} a^k b^{p-k}.$$

The binomial coefficient in the last sum is equal to $p!/k!\,(p-k)!$, where $0 < k < p$. The binomial coefficient is an integer, which means that all factors in the denominator get cancelled by corresponding factors in the numerator. But the prime factorization of $p!$ contains a prime(!) factor of p, whereas the denominator can *not* contain a prime factor of p, since all factors in the denominator are less than p, and therefore no combination of factors can give p or a multiple of p. Therefore, $p!/k!\,(p-k)!$ must be a multiple of p for $0 < k < p$. Since multiples of p vanish when computing modulo p, what remains of $(a + b)^p$ is $a^p + b^p$.

Theorem 3 (Fermat's Little Theorem). Let p be a prime number and let a be an integer with $1 < a < p$. Then $a^p \bmod p = a$.

Proof. By induction on a. For $a = 0$, we have $0^p \bmod n = 0 \bmod n = 0 = a$. Now let $a \ge 0$. By induction, we can assume $b^p \bmod p = b$ for all b with $0 \le b \le a$. Then $(a + 1)^p \bmod p = a^p + 1^p \bmod p$ by Lemma 1, and this, by induction, is equal to $a + 1^p \bmod p$, which equals $a + 1 \bmod p$.

Definition 1 (Euler's Totient). Let n be a positive integer. The number of integers m with $0 < m < n$ and $\gcd(m,n) = 1$ is written $\varphi(n)$.

Lemma 2. Let p be a prime. Then $\varphi(p) = p - 1$.

Proof. Since p is prime, we have $\gcd(p,n) = 1$ for all integers n with $0 < n < p$. Since there are $p - 1$ such integers, the conclusion follows.

Lemma 3. Let p and q be primes. Then $\varphi(pq) = (p-1)(q-1)$.

Proof. Since p and q are primes, the only integers n between 0 and pq with $\gcd(pq,n) \neq 1$ are those that are multiples of either p or q. There are $pq - 1$ integers between 1 and $pq - 1$. Of these, $p - 1$ are multiples of q, and $q - 1$ are multiples of p. Therefore $\varphi(pq) = pq - 1 - (p-1) - (q-1) = pq - p - q + 1 = (p-1)(q-1)$.

Lemma 4. Let n be a positive integer and let a and b be integers with $0 < a, b < n$ and $\gcd(a,n) = 1$ and $\gcd(b,n) = 1$. Then $\gcd(ab,n) = 1$.

Proof. We make use of the fact that a, b, and n have *unique* prime factorisations. Then $\gcd(a,n) = 1$ means that no factor in the prime factorisation of a appears in the prime factorization of n, and the same is true of b and n. The prime factorization of ab has only prime factors that appear in the prime factorisations of a or b, and thus cannot contain a prime factor from n.

Theorem 4. (Euler's Theorem). Let n be a positive integer and let a be an integer with $0 < a < n$ and $\gcd(a,n) = 1$. Then $a^{\varphi(n)} \bmod n = 1$.

Proof. This obviously generalizes Fermat's Little Theorem. Let R be the set of all positive integers between 1 and n such that $\gcd(x, n) = 1$. There are obviously $\varphi(n)$ such numbers, by definition, so $|R| = \varphi(n)$ and we can write $R = \{ x_1, \ldots, x_{\varphi(n)} \}$. Now consider the set $aR = \{ ax \bmod n \mid x \in R \}$. Since $\gcd(a, n) = 1$, we have $aR = R$ because of Lemma 4. This means that

$$x_1 x_2 \cdots x_{\varphi(n)} \bmod n = (ax_1)(ax_2) \cdots (ax_{\varphi(n)}) \bmod n$$

$$= a^{\varphi(n)}(x_1 x_2 \cdots x_{\varphi(n)}) \bmod n.$$

Cancelling the factor of $x_1 x_2 \cdots x_{\varphi(n)}$ on both sides proves the theorem.

RSA Key Pair Creation

To generate an RSA key pair, consisting of a public key and a private key, we need two large random primes, p and q and form $n = pq$. They should be random and have about the same number of bits, but not exactly the same number of bits. There are many, many other restrictions that we cannot go into here. As always, do not implement your own RSA crypto, but use a library instead. Later in this lab, we will see what happens when (and how) RSA key generation can go wrong. Later we will also discuss the question how long n must be, in bits, to get 128-bit security.

The next step is to choose some positive integer e so that $\gcd(e, \varphi(n)) = 1$. Often, libraries choose $e = 3$ or $e = 65537$. Of course, in order for that to work, $\varphi(n)$ must not be divisible by 3 or 65537.

Finally, we use Euclid's extended algorithm to compute the unique integer d such that $ed \bmod \varphi(n) = 1$, which must exist according to Corollary 2.

The public key is then (e, n) and the private key is (d, n).

To summarise, here is "textbook RSA":

- Choose two large random primes p and q of approximately the same size. Compute $n = pq$.
- Choose e so that $\gcd(e, \varphi(n)) = 1$.
- Compute d so that $ed \bmod \varphi(n) = 1$, using Euclid's extended algorithm.
- The public key is (n, e), and the private key is (n, d).

We emphasise again that *if you implement RSA in this way, you are doing it wrong*. This formulation can *not* be used as a blueprint for a secure implementation of RSA.

RSA Encryption and Decryption

In RSA, messages to be encrypted are integers m with $0 \leq m < n$. If necessary, we will have to encode our message so that it fits this criterion. One simplistic way of encoding a message is to concatenate the bits that make up the UTF-8 encoding of that message and then to interpret the resulting bit string as an integer. Usually, the messages that RSA encrypts are not text messages anyway, but symmetric keys, and for these the problem of encoding does not arise.

Given a plaintext m, the encrypted ciphertext is $c = E(m) = m^e \bmod n$.

Given a ciphertext c, the decrypted plaintext is $m = D(c) = c^d \bmod n$.

In order for this to work, we would have to have $D(E(m)) = m$ for all messages m. And indeed:

Theorem 5 (RSA works). Let (n, e) be a RSA public key and (n, d) the corresponding private key. Let m be an integer with $0 \leq m < n$. Then $(m^e \bmod n)^d \bmod n = m$.

Proof. We know that $ed \bmod \varphi(n) = 1$. Therefore we can write ed as a multiple of $\varphi(n)$, plus 1, or $ed = k\varphi(n) + 1$ for some integer k. We now have:

$$(m^e \bmod n)^d \bmod n = (m^e)^d \bmod n$$

$= m^{ed} \bmod n$

$= m^{k\,\varphi(n)+1} \bmod n$

$= m\left(m^{\varphi(n)}\right)^{k} \bmod n$

$= m \cdot 1 \bmod n$

$= m.$

To summarise:

- Messages are integers m with $0 \leq m < n$.
- To encrypt m, compute $c = m^e \bmod n$.
- To decrypt c, compute $m = c^d \bmod n$.

Difficulty of Breaking RSA

All this would be of no use if it were easily possible to compute m from $m^e \; mod \; n$ without also knowing d. But it appears that this is precisely *not* possible. First, every attempt to break RSA could so far be shown to be equivalent to factor n. Factoring n gives us $\varphi(n)$, and e and $\varphi(n)$ give us d, which enables us to decrypt m.

And second, we believe that factoring n is hard. So far, the problem of factoring large integers with two or more large prime factors has resisted all attempts to find a polynomial-time solution. Does it have to remain this way? No. Is it possible that tomorrow someone invents an algorithm that breaks RSA without factoring n? Yes, absolutely. Is it possible that tomorrow someone invents an algorithm that factors n with large prime factors quickly? Yes, equally absolutely. Do we believe that this will happen? No. Why don't we believe it? Because many very, very clever people have tried and failed. Note that this is a Bayesian argument ("The sun has risen for the last 4 billion years, therefore it will very probably also rise tomorrow"), and not a logical one ("the sun *must* rise tomorrow because this follows logically from a, b, and c").

The best algorithm for factoring, known as the General Number Field Sieve (GNFS), uses approximately $W(b) = \exp\left(1.92\, b^{\frac{1}{3}}(ln\; b)^{\frac{2}{3}}\right)$ steps to factor a b-bit number. This is subexponential, but still slower than any polynomial; remember the graph shown in the Introduction to Cryptology lecture.

We express security in terms of the work factor, and we express that work factor in the form 2^w. If we want to know how large b must be for w-bit security, we must therefore find b so that $W(b) = 2^w$.

Exercise 1. Write a program that finds out the correct value of b for $w = 128, 256, 384,$ and 512. Remember that b is an integer, and therefore what we need is really the smallest b so that $W(b) > 2^w$. Your program need not be extremely efficient, but should do its job in less than a second. (The code skeleton for this exercise is in `PublicKeyLab.java` in the function `exercise1()`.)

The output of this program will show you that attaining 128-bit security can not be done with a 128-bit n, and not even with a 512-bit n. Show the values of b for the various values of n here:

```
128 bits work factor:   1766 bits RSA exponent
256 bits work factor:   9427 bits RSA exponent
384 bits work factor:  25816 bits RSA exponent
512 bits work factor:  53311 bits RSA exponent
```

RSA Example

In this section, you will develop a toy implementation of RSA. We emphasise again that the implementation that you will develop is not going to be good for production use, since there are many hidden subtleties, and secure implementations of RSA are tricky.

Exercise 2. Complete the class `RSA.java`. Replace every instance of `Your solution here` with your solution according to the specification given in the Javadoc.

The constructor that reads from the input stream should read n and e from the stream and then attempt also to read d. If reading d fails because the input stream is at end-of-file, there was only a public key in the input stream. In this case, only the encrypt method should be supported, but not the decrypt method: this should throw an `OperationNotSupportedError`. The encrypt and decrypt methods should throw a `BadMessageException` if the message, ciphertext or signature is not between 1 and $n - 1$. The save method should save n, e, and d to the output stream, in that order (and throw an `OperationNotSupportedError` if the private key doesn't exist). The `savePublic` method should save only n and e, in that order. Note that all input and output streams are *object streams*, suitable for object serialization and deserialization.

Exercise 3. Extend PublicKeyLab.java with a method that

- obtains a short string from the command line;
- encodes that string as a sequence of ASCII characters and then as a `BigInteger`;
- reads a public key from a file using the methods in the class RSA;
- encrypts the encoded message with the key, and
- writes the encrypted `BigInteger` to a file as an `ObjectStream`.

In the same program, also write functionality that

- reads an encrypted `BigInteger` from an `ObjectStream`;
- reads a private key from a file using the methods in the class RSA;
- decrypts the ciphertext with the private key;
- decodes the resulting `BigInteger` as a sequence of ASCII characters; and
- prints the decrypted message to `System.out`.

Exercise 4. Using the public key in the file `keypair.rsa`, encrypt a short message. Decrypt the ciphertext. Is it the message you expected?

RSA Signatures

RSA has a nice property, not shared by all public-key algorithms, which is that not only $D(E(m)) = m$, but also $E(D(m)) = m$ as well. This property is what enables RSA to be used for electronic signatures.

Recall that decrypting a ciphertext c with the private key (d, n) is done by computing $m = c^d \bmod n$. Since the private key is, well, private, only the legitimate owner of the private key can decrypt the message. Therein lies RSA's confidentiality guarantee.

But we can turn this around. Let's say that Alice wants to send a message m to Bob that doesn't need to be confidential, but which does need to be authentic. So there is no need to encrypt m; it can be transmitted in the clear. But how do we authenticate it? This can only be done by using something that only Alice can legitimately have. According to the previous paragraph, this could be her private key. If m is a number between 0 and n, Alice can "decrypt" m by computing $s = m^d \bmod n$. She can then send (m, s) as the authenticated message. Bob now receives the message, computes $m' = s^e \bmod n$

and compares m' with m. If $m' = m$, then s must have been computed by Alice. This is taken as evidence that Alice has sent m.

Exercise 5. Show the following. Let (e, n) be an RSA public key and let (d, n) be the corresponding private key. Let m be an integer with $0 \leq m < n$. Then $(m^d \bmod n)^e \bmod n = m$.

$(me \bmod n)d \bmod n = (me)d \bmod n$

We thus have a message transformation with the following properties:

- only Alice can transform the message, since only Alice knows her private key; and
- everyone can check the transformation, since everyone knows the public key.

We call a message transformation scheme that has these properties a *digital signature scheme* and we call s a *digital signature*. Not every public key cryptosystem can be used as a digital signature scheme, but RSA can.

The RSA key pair now becomes dual-use, since each key now has two uses:

- the public key is used to encrypt messages and to verify signatures; and
- the private key is used to decrypt messages and to create signatures.

Exercise 6. Let (e, n) be Alice's public key and (d, n) be her private key. Alice now composes and encodes the message "Send CHF 1'000 to Eve" into an integer m with $0 \leq m < n$. She then sends $(m, m^d \bmod n)$ to her bank. The bank verifies the signature on m, concludes that it is correct and indeeds sends CHF 1'000 to Eve. You now assume the role of Eve. Eve sits on the channel between Alice and her bank and has found out how she can listen to messages and send new messages, but she cannot change or remove messages. In this exercise, we assume that Eve cannot signatures for arbitrary messages that would verify as having come from Alice. (In other words, given m, Eve cannot herself compute $m^d \bmod n$.) How can Eve exploit her capabilities to her financial gain?

Eve can repeatedly send the same message m to Alice's bank and would receive the CHF 1000 each time.

Exercise 7. Suggest a remedy that solves the problem developed in Exercise 8.

Find a way that ensures that the message is actually from the correct person.

timestamp for verification

Exercise 8. Extend the RSA class by two methods:

```
public BigInteger sign(BigInteger message) throws BadMessageException;
public boolean verify(BigInteger message, BigInteger signature)
```

If the input stream in the constructor only contains a public key, only the encrypt and verify methods should be supported, but not the decrypt and sign methods: these should throw an `Operation-NotSupportedError`.

Exercise 9. Extend the methods starting `exercise9…` in `PublicKeyLab.java` so that they

- obtains a short string from the command line;
- encodes that string as a sequence of ASCII characters and then as a `BigInteger`;
- reads a private key from a file using the methods in the class RSA;
- signs the encoded message with the key, and
- writes the encoded string and the signature to a file as an `ObjectStream`.

In the same program, also write functionality that

- reads an encoded string and a signature from an `ObjectStream`;
- reads a public key from a file using the methods in the class RSA;
- verifies the signature with the public key;
- if signature verification was successful, prints the verified message to `System.out`

Exercise 10. Use the program with `keypair.rsa` to verify the signature on the message in the file `message-with-signature.bin`. Then copy that file and change a single bit in it. Does the signature still verify? No, the signature does not verify anymore.

Attacks on RSA

As we have claimed, RSA is *secure*, by which we mean that an attacker can not obtain the private key from the public key and arbitrarily many plaintext/ciphertext pairs, nor can he obtain any plaintext, given only the public key and arbitrarily many plaintext/ciphertext pairs. In this section, we will outline some of the problems and pitfalls that implementing RSA in the textbook way can bring.

Short Message Attack

We have already noted above that some implementations of RSA always use $e = 3$ or $e = 65537$. This is done for efficiency reasons, since x^3 or x^{65537} can be computed very efficiently. Assume now that your implementation has chosen $e = 3$. (For this to work, $\gcd(3, \varphi(n))$ must be 1.)

In this case, the ciphertext of a message m is simply $m^3 \ mod \ n$. How can this be a problem? Well, if m is so small that $m^3 < n$, then the ciphertext c is simply m^3 (without the "$\bmod n$"), because there is no overflow, and therefore no wrap-around. The attacker can simply extract the cube root of c and will get m back.

For example, if $p = 6221$ and $q = 7789$, then $n = pq = 48455369$. If $m = 323$, then $323^3 = 33698267$ and therefore $c = 323^3 \bmod 48455369$ is also 33698267. An attacker could recover m from c simply by taking the cube root. This is known as the *short message attack*, since it affects messages that are so short that there is no overflow.

Exercise 11. Given n, what is the largest m that is susceptible to a short message attack? First, calculate an answer for any n. Express your answer both *in terms of n*, and also *in terms of the number of bits needed to represent n*. Then look at some specific values for n: if the size of n in bits is 1024 (2048, 3072, 4096), what is the length, in bits, of the largest m that is susceptible to a short-message attack?

```
n: 1024 [bit] ---> max. m len: 341 [bit]
n: 2048 [bit] ---> max. m len: 682 [bit]
n: 3072 [bit] ---> max. m len: 1024 [bit]
n: 4096 [bit] ---> max. m len: 1365 [bit]
```

Exercise 12. Suggest one or more remedies that make short message attacks impossible. Note that "only send long enough plaintexts" is not a remedy, since the algorithm must work securely even when the plaintext to be sent is short. Also note that the length of the *plaintext* is not necessarily the same as the length of the *message* that is being sent.

maybe append or prepend a random number

Low Public Exponent Attack

In order to solve the short message attack problem, one could try to pad messages with random bits so that they are always as long as the modulus. This way, there are no short messages and therefore there is no short message attack.

Unfortunately this approach is also attackable if the message m is sent to at least e recipients. For example, if $e = 3$, then sending the same message to three or more recipients exposes m. How does this work? Let the public keys of the three recipients be (e_1, n_1), (e_2, n_2), and (e_3, n_3). We now send messages c_1, c_2, and c_3 according to the equations

$$\begin{aligned} c_1 &= m^3 \bmod N_1 \\ c_2 &= m^3 \bmod N_2 \\ c_3 &= m^3 \bmod N_3. \end{aligned} \tag{3}$$

There is a nifty theorem in number theory called the Chinese Remainder Theorem (CRT) which says that a system of equations of the form

$$\begin{aligned} y_1 &= x \bmod n_1 \\ y_2 &= x \bmod n_2 \\ y_3 &= x \bmod n_3 \\ &\dots \\ y_k &= x \bmod n_k \end{aligned} \tag{4}$$

has a *unique* solution modulo $n_1 n_2 \dots n_k$ if and only if $\gcd(n_i, n_k) = 1$ for all $i \neq j$. In other words, if $\gcd(n_i, n_k) = 1$ for all $i \neq j$, then there is a unique integer x between 0 and $n_1 n_2 \dots n_k$ that fulfils all the equations in Eq. (3). What this theorem means for Equation (3) is that if the gcd of N_i and N_j is 1 for all pairs of indices i and j with $i \neq j$ (which ought to be true, since otherwise the moduli are susceptible to a common factor attack, see below), then there is a unique number x with $0 \leq x < N_1 N_2 N_3$ so that x satisfies equation (3). Now, since $0 \leq m < N_1$, $0 \leq m < N_2$, and $0 \leq m < N_3$, we know that $x = m^3 < N_1 N_2 N_3$. Therefore, the plan of attack is clear:

1. Given the three ciphertexts of Eq. (3), invoke the CRT to compute $x = m^3$.
2. Compute m by computing the cube root of x.

From Discrete Mathematics, you should know that the solution x is given by

$$x = c_1 n_1 d_1 + c_2 n_2 d_2 + c_3 n_3 d_3, \text{ where } n_i = N_1 N_2 N_3 / N_i \text{ and } d_i n_i \bmod N_i = 1 \quad (i = 1,2,3) \tag{5}$$

Here, d_i is the modular inverse of n_i, which must exist since $\gcd(d_i, n_i) = 1$.

For $e = 3$, the above attack will work if the same message is sent to three different recipients. Of course the attack will also work for larger e, if the same message is sent to e or more different recipients. But the larger e becomes, the more improbable this gets. (Think about having $e = 65537$ different recipients to a message.) Choosing a larger value for e is therefore not a good, secure solution to this problem.

Exercise 13. Suggest a remedy for the above attack.

maybe add a different random number for every recipient

Common Factor Attack

This part of the exercise illustrates that looking only at the difficulty of factoring a single RSA key (cracking a single AES key, finding a single preimage for a hash, …) is often myopic. With well-chosen factors, factoring an RSA modulus into its constituent primes is computationally infeasible (we think). But now let us assume that we generate millions of RSA keys at the same time. This exercise explores how it can be that cracking many RSA keys may in some circumstances be easier than cracking a single RSA key.

Computers generate RSA keys by using the operating system's random number generation method. On Unix and Linux, this usually entails reading from the special file `/dev/urandom`. This special file is an interface to kernel-level code that takes existing randomness and constantly mixes in new sources of randomness into what's known as a *random pool*. When data is read from that pool, it is expanded into an essentially infinite nonrepeating and unpredictable stream of bytes.

When a Unix or Linux computer is booted for the very first time after installation, there is no data in the random pool, and at the time when that pool is initialised for the first time, the computer's state is quite predictable. That means that if you take many identically-installed machines and turn them on, the first random bytes that come out of `/dev/urandom` will often be the same. What that means, for example, is that the procedure for generating the RSA keys for secure login on these machines may, for a short while, use the same random bytes.

Exercise 14. Let us assume that we have N RSA moduli, $n_1 = p_1 q_1$, …, $n_N = p_N q_N$, and some of these were generated with identical bytes from `/dev/urandom` so that for some indices $i \neq j$, we have $p_i = p_j$. Let b be the number of bits in the largest modulus. Suggest an algorithm of complexity not worse than $O(N^2 b)$ that not only finds out all the pairs (i, j) so that $p_i = p_j$, but also then breaks the RSA moduli n_i and n_j, i.e., finds their prime factors. **Hint.** What is the gcd of two moduli that do not share a common prime factor, and what is the gcd of two moduli that do share a common prime factor? Also, you may assume that computing the gcd of two b-bit numbers has complexity $O(b)$.

```
for i in N:
 for j in N:
  a = gcd(n_i, n_j)
  if (a > 1):
    p_i, p_j = a
    q_i = n_i / p_i
    q_j = n_j / p_j
```

Exercise 15. Suggest a remedy for the problem given in Exercise 14. This solution must not cost so much so as to make its cost prohibitive. (For example, random number generators based on radioactive decay are not acceptable, since they would require expensive shielding.) This is a hard design question for which there is no single correct answer.

use another random generator from the system such as getrandom()

Points

In this lab you can get up to **4 Points**:

- **2 point** for solutions to the programming assignments.
- **1 point** for mostly correct answers to the theoretical assignments.
- **1 points** for suggesting a design that solves Exercise 15.