Haskell Cryptography

COMP40009 – Computing Practical 1

12th – 16th October 2020

Aims

- To provide experience with defining basic recursive functions over integers and tuples.
- To explore the basic algorithms underpinning public and (optional) symmetric key cryptography.

Submit by 15:00 on Friday, 16th October

Introduction

Cryptography is a domain of computer science which enables two parties to exchange secret data in such a way that no one else is able to observe the raw data being exchanged. An encryption scheme is generally composed of two functions:

- The *encryption* function, which is used by the sender to blind their message (also called *plain text*) with an *encryption key* to produce a *ciphertext*.
- The *decryption* function which, together with the *decryption key*, enables the receiver to recover the original plain text from the ciphertext.

Broadly, there exist two types of encryption schemes. In *symmetric key* encryption algorithms (or *private key* algorithms), the same key must be used during encryption and decryption. Thus, the parties have to agree beforehand on a shared private key which must be kept secret. In contrast, *asymmetric* encryption (or *public key* encryption) uses two different keys for encryption and decryption: the encryption key is public so that everyone can encrypt a message, but the decryption key is kept private by its owner, who is the only person who can decrypt the ciphertexts.

In this lab session, you will build Haskell implementations of a widely-used public key encryption algorithm and (optionally) a symmetric key algorithm.

A public key algorithm: RSA

RSA [2][1] is named after its inventors: Ron Rivest, Adi Shamir and Leonard Adleman. It is a widely used public-key algorithm based on the hardness of the prime factor decomposition problem[2].

[1] If you are logged in on a College machine then you can access the ACM Digital Library for free. There is also a publicly available pdf version of this document here:
http://people.csail.mit.edu/rivest/Rsapaper.pdf

[2] In the 1970s if someone had asked what prime numbers were used for, a legitimate answer would have been "nothing". Things have changed significantly since then!

The algorithm for generating the key pair comprises the following four steps:

1. Choose two distinct prime numbers p and q.
2. Compute the RSA modulus $N = p\,q$
3. Choose an integer $e > 1$ such that $gcd(e, (p-1)(q-1)) = 1$
4. Compute an integer d such that $e\,d = 1 \pmod{(p-1)(q-1)}$

where $gcd(a, b)$ denotes the greatest common divisor of a and b. Note that when we write $a = b \pmod N$ we mean that a and b are the same when *both* are expressed modulo N, i.e. that $a \bmod n = b \bmod n$, treating 'mod' as a Haskell-like operator.

We now define the *public key* as the tuple (e, N) used for encryption and the *private key* as (d, N) needed for decryption. The encryption and decryption functions are described as follows: for any plain text x and any ciphertext c:

$$\begin{array}{rcl} encryptRSA_{(e,N)}(x) & = & x^e \pmod N \\ decryptRSA_{(d,N)}(c) & = & c^d \pmod N \end{array}$$

The correctness of RSA ensures that for every message x, the following property holds:

$$decryptRSA_{(d,N)}(encryptRSA_{(e,N)}(x)) = x$$

Indeed[3], it can be shown that if $i = j \pmod{(p-1)(q-1)}$ then $x^i = x^j \pmod N$. In particular, if we choose e such that $e\,d = 1 \pmod{(p-1)(q-1)}$, then $(x^e)^d = x^1 = x \pmod N$.

The security of RSA resides in the hardness of prime factor decomposition. As a matter of fact, finding the private exponent d given e and N is no harder than factorisation. If we can factorise N into its prime factors p and q, we can easily compute step 4 of the key pair generation and recover d. Conversely[4], it can be shown that we can factorise N given the exponents e and d [1].

Bézout coefficients

In order to implement RSA you need to find the *multiplicative inverse* of an integer, say a. For conventional multiplication, the inverse of a is, of course, just $1/a$. Under modulo arithmetic the inverse of a is an integer a^{-1} that satisfies $a\,a^{-1} = 1 \pmod m$ for some given modulus m; or equivalently, $a\,a^{-1} = 1+mk$ for some integer k. The equivalent of a being non zero for conventional multiplication is that a should be coprime with m in modulo arithmetic.

One way to calculate inverses is to compute so-called *Bézout coefficients*: given two non negative integers a and b, two integers u and v are called Bézout coefficients of a and b iff they satisfy $au+bv = gcd(a, b)$. From this definition we can derive an elegant recursive algorithm for computing such coefficients as follows. First, divide a by b and let q and r be the quotient and the remainder respectively (you can use Haskell's `quotRem` function to give you both) such that:

$$a = b\,q + r \tag{1}$$

Now let us assume that we have recursively computed u' and v', some Bézout coefficients of b and r. Then, by definition:

$$b\,u' + r\,v' = gcd(b, r) \tag{2}$$

Before going further, we would like to argue that:

$$gcd(a, b) = gcd(b, r) \tag{3}$$

[3] Section 1 of the Technical Note provides a detailed proof of correctness.

[4] We also prove in Section 2 of the Technical Note that recovering d from e and N is equivalent to factorising N, which is intractable for large values of N. Note that this problem is different from decrypting a ciphertext c given e and N, which can be easier depending on c, e.g. if $c = 0$.

Indeed, let x be a non negative integer and let us assume that x divides $gcd(a, b)$ (conventionally written as "$x \mid gcd(a, b)$"). Then $x \mid a$ and $x \mid b$. But as $r = a - b\,q$, we also know that $x \mid r$ and thus $x \mid gcd(b, r)$ and thus $gcd(a, b) \mid gcd(b, r)$. Conversely, we can show that $gcd(b, r) \mid gcd(a, b)$ which proves the desired equality.

Putting Equations (1), (2) and (3) together, we can see that these imply $a\,v' + b\,(u' - q\,v') = gcd(a, b)$. In other words, $(v', (u' - qv'))$ are Bézout coefficients of a and b respectively. That's the basis of a recursive algorithm. Now, all you need is a base case. Well, if $b = 0$ then $gcd(a, b) = a$, so the coefficients in that case must be $(1, 0)$. Note that we can be sure that the recursion terminates when $b > 0$ because, in the recursive call, the second argument becomes a mod b which is strictly less than b and thus 'closer' to the base case.

Getting started

As per the previous exercise, you will use the `git` version control system to get the repository with the skeleton files for this exercise and its (incomplete) test suite. You can get your repository with the following (remember to replace the *`username`* with your own username).

```
git clone https://gitlab.doc.ic.ac.uk/lab2021_autumn/haskellcrypto_username.git
```

What to do: Part 1

You will now implement the required functions for the key generation and the encryption functions. Note that, in practice, RSA is typically used to encrypt 1024-bit messages, but here we will we work with 64-bit integers (type `Int`) to simplify auto-testing.

- Define a function `gcd :: Int -> Int -> Int` that returns the greatest common divisor of two given integers. To compute `gcd m n`, the rules are as follows:
  - If `n == 0` then the answer is `m`
  - Otherwise, the answer is `gcd n (m mod n)`

  Note that we could change the type to `gcd :: (Int, Int) -> Int` to make it consistent with the usual mathematical notation (e.g. $gcd(a, b)$), but you should write it in its so-called *curried* form, where there are two arguments rather than a single tuple.

- Using `gcd` define a function `smallestCoPrimeOf :: Int -> Int` which, given a non zero integer a, returns the smallest integer $b > 1$ that is coprime with a, i.e. for which $gcd(a, b) = 1$. To do this, check the candidate numbers 2, 3, 4, ... in order.

- Using a list comprehension and your `gcd` function from above, define a function `phi :: Int -> Int` that, given an integer m, computes the *Euler phi* or *Totient* function, $\phi(m)$, which is the number of integers in the range 1 to m inclusive that are *relatively prime* to m, i.e. for which $gcd(a, m) = 1$.

  Note: you won't be using `phi` directly to encode RSA, but it can be used to help set up the RSA parameters and verify some of the underlying theory. For example, you might like to test your function by verifying the (provable) properties that $\phi(p) = p - 1$ when p is prime and $\phi(n) = (p - 1)(q - 1)$ when $n = p\,q$ and both p and q are prime.

- Given integers a, k and m such that $0 \leq a < m$, define a function `modPow :: Int -> Int -> Int -> Int` that returns a^k mod m.

The trick here is to avoid overflow in the intermediate calculations, which you can do by inserting calls to `mod`. For example, adapting the 'logarithmic' powering scheme covered in the lectures, you can use the following properties:

- If $k = 2j$ is even: $a^k = (a^2 \bmod m)^j \bmod m$
- If $k = 2j + 1$ is odd: $a^k = (a\,((a^2 \bmod m)^j \bmod m)) \bmod m$

Note that the second case can make use of the first one. This exploits the fact that $(a\,b) \bmod c = (a \bmod \ \times b \bmod c) \bmod \ c$. It also means that a and m can be as large as $\sqrt{\texttt{maxBound :: Int}}$, without incurring arithmetic overflow, where `maxBound :: Int` is the largest `Int` that Haskell can represent. You can find out what this is by typing the above at the ghci prompt[5]:

```
*Main> maxBound :: Int
9223372036854775807
```

- Define a function `computeCoeffs :: Int -> Int -> (Int, Int)` that, given two non negative integers a and b, returns a pair of Bézout coefficients. The algorithm for computing the Bézout coefficients is given above.
- Using `computeCoeffs` and `gcd` define a function `inverse :: Int -> Int -> Int` which, given a and m, returns a^{-1}, the multiplicative inverse of a such that $a\,a^{-1} = 1 \pmod{m}$.

  Hint: An integer a has an inverse modulo m if and only if $gcd(a, m) = 1$. Moreover, if we have two Bézout coefficients u and v such that $au + mv = 1$, then $au = 1 - mv$, in which case $u \bmod m$ is the required inverse of a, modulo m.
- Define `genKeys :: Int -> Int -> ((Int, Int), (Int, Int))` which, given two distinct prime numbers, runs the RSA key generation algorithm and returns the key pair $((e, N), (d, N))$ as described above. Note that step 3 of RSA allows us to choose any $e > 1$ that is coprime with $(p - 1)(q - 1)$, but in this exercise we will choose the smallest such integer.
- Define a function `rsaEncrypt :: Int -> (Int, Int) -> Int` that takes a plain text x and a public key (e, N) and returns the ciphertext $x^e \bmod N$.
- Similarly, define a function `rsaDecrypt :: Int -> (Int, Int) -> Int` that takes a ciphertext c and a private key (d, N) and returns the plain text $c^d \bmod N$.

Symmetric encryption

This part of the exercise is optional, so you are only advised to do it if you have time; it will give you more practice at writing recursive functions and will give you some insight into symmetric key encryption.

Symmetric encryption schemes [3] require the sender and the receiver to agree on a shared key for encryption and decryption. The security of such protocols is highly dependant on the size of the key. Shannon even showed in the 1940s that the only way to build an unbreakable symmetric cipher would be to use a key at least as long as the message being sent, and to use that key only once[6].

[5] We will explain what is going on here later in the course but, essentially, many types can have a `maxBound` – this just asks for the one for `Int`.

[6] Such a cipher, called *one-time pad*, blinds every single bit or character of the plain text with a different part of the key. Every bit of the key is used only once, which makes any type of cryptanalysis impossible and thus no information about the plain text is leaked by the knowledge of the ciphertext.

In practice, the keys must have a fixed size and are used several times, while the messages we send can have an arbitrary length. There are two techniques to encrypt such messages: *Stream ciphers* encrypt the bytes of the plain text one by one, whereas *block ciphers* encrypt blocks of bits of the plain text en masse.

In the next part, you will implement an example of block cipher working on strings, where a block will be identified to a single character. You will then see how the security can be improved by linking the encrypted blocks together.

What to do: Part 2

- Define a function `toInt :: Char -> Int` which returns the position of a letter in the alphabet (where 'a' is at position 0).
- Define a function `toChar :: Int -> Char` which takes a position n and returns the n^{th} letter.
- Define a function `add :: Char -> Char -> Char`, that we will write $\oplus$, which "adds" two letters using modular arithmetic on their position in the alphabet. For example `'a'` $\oplus$ `'c' = 'c'` and `'y'` $\oplus$ `'e' = 'c'`.
- Define a function `subtract :: Char -> Char -> Char`, that we will write $\ominus$, which "subtracts" two letters based on their position. For example `'h'` $\ominus$ `'c' = 'f'` and `'b'` $\ominus$ `'e' = 'x'`.

The functions `add` and `subtract` define an encryption scheme on letters which takes a letter as a key[7]. The functionality of the associated encryption and decryption functions can be expressed thus:

$$\begin{aligned} e_k : x &\mapsto x \oplus k \\ d_k : c &\mapsto c \ominus k \end{aligned}$$

Considering a letter as a single block of a string, we will now see two *modes of operation* which enable us to compose this scheme to encrypt strings. In what follows, we will write a string s of length n as $s = s_1 s_2 \cdots x_n$ where s_i is the i^{th} letter of s. Those algorithms[8] return ciphertexts c of the same length as the messages m and are written $c_1 c_2 \cdots c_n$.

- Define a function `ecbEncrypt :: Char -> String -> String` which takes a key k and a message m and encrypts it with respect to the *Electronic CodeBook* mode (ECB), for all $i \in [\![1, n]\!]$: defined as
$$c_i = e_k(m_i)$$
- Define a function `ecbDecrypt :: Char -> String -> String` which takes a key k and a ciphertext c and reverts the `ecbEncrypt` function, for all $i \in [\![1, n]\!]$:, by computing
$$m_i = d_k(c_i)$$

[7] Furthermore, the key is of the same size as the message (which is also a letter), so this is a one-time pad, provided that each key is used only once. Indeed, let us imagine that we encrypt a message m with a key k to get the ciphertext $c = m \oplus k$. The knowledge of c leaks no information about m since the key k looks random, and thus m is equally likely to be any letter of the alphabet.

However, if we can intercept two messages, $c_1 = m_1 \oplus k$ and $c_2 = m_2 \oplus k$, encrypted with the same key k, we would be able to compute $c_2 \ominus c_1 = (m_2 \oplus k) \ominus (m_1 \oplus k) = m_2 \ominus m_1$, which would reveal some information about the plain texts.

Finally, we considered a simple shift cipher for the sake of simpicity. Other ciphers on single letters exist, such as affine ciphers or permutation ciphers, that you are free to implement as an extension.

[8] The correctness of those modes of operation is discussed in Section 3 of the Technical Note.

Remark: The main flaw of the Electronic CodeBook is that such an encryption is *deterministic*. Indeed, if the same block appears twice in the plain text, it will be encrypted into the same block in the ciphertext, which is a starting point for cryptanalysis. In order to reduce such threats, other modes of operations like the Cipher Block Chaining mode have been designed, where the encryption of a block m_i also depends on the last encrypted block c_{i-1}.

- Define a function `cbcEncrypt :: Char -> Char -> String -> String` which takes a key k, an initialisation vector iv[9] and a message x and encrypts it with respect to the *Cipher Block Chaining* mode (CBC), defined as:

$$\begin{cases} c_1 &= e_k(x_1 \oplus iv) \\ c_i &= e_k(x_i \oplus c_{i-1}) \quad \text{for} \quad 1 < i \leq l \end{cases}$$

- Define a function `cbcDecrypt :: Char -> Char -> String -> String` that takes a key k, an initialisation vector iv and a ciphertext c and decrypts the latter as:

$$\begin{cases} x_1 &= d_k(c_1) \ominus iv \\ x_i &= d_k(c_i) \ominus c_{i-1} \quad \text{for} \quad 1 < i \leq l \end{cases}$$

Note: your repository containing the skeleton for this lab can be found at `https://gitlab.doc.ic.ac.uk/lab2021_autumn/haskellcrypto_username`. As always, you should use LabTS to test and submit your code to CATe.

References

[1] Dan Boneh et al. Twenty years of attacks on the rsa cryptosystem. *Notices of the AMS*, 46(2):203–213, 1999.

[2] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[3] Nigel Paul Smart. *Cryptography: an introduction*, volume 5. McGraw-Hill New York, 2003.

Assessment

In general, the assessment for laboratory exercises uses the following scheme:

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.
```

[9] Both the key and the initialisation vector have the same size, and are thus a single letter in this example.

```
A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```