SCHOOL OF ELECTRICAL ENGINEERING, UNIVERSITY OF THE WITSWATERSRAND

ELEN3015 - Data and Information Management

Lecturer: Ling Cheng

Lab 3 RSA

20 March 2018

Tyson Cross 1239448

INTRODUCTION

RSA is an important public key cryptographic public-key system names after its authors, Rivest, Shamir and Adleman[1] in 1978. The system encrypts a message with a public key (a large prime number, whose value is published and freely availible), which is only able to be decrypted with a separate private key (also prime, but kept secret from non-trusted parties.) The basic mechanism of RSA is the use of large prime numbers in a system of modular mathematics. The security of RSA is premised on current techniques involving the composites of large prime numbers: it is widely believed to be computationally infeasible to factor big integers into large primes within a reasonable period of time[2]. Given the public key, it should not be possible (or extremely difficult) to find the private key.

1 THE RSA ALGORITHM

RSA is still widely used in a variety of contexts and applications, despite how slow it is compared to other newer encryption methods. Examples of the system's modern usage include SSL certification, SSH, HTTPS tunnelling, PGP and RSA SecureID[3]. The use of a private, confidential key to decrypt a message is a convenient mechanism enabling private, secret and non-reputability of online and electronic communication.

1.1 Cryptographic communication

Alice publishes her public key openly. She encrypts a message using her public key e, and a related number n. This second component of the key pair is the product of two large prime numbers (these two primes, p and q are also kept secret). She secretly sends her private key e, and again the same component, n, to Bob, so he can decrypt the message. Eve has access the to public key e and the value of n, but for RSA to work, it must be very difficult for Eve to find the value of d, p or q.

1.2 Steps in the RSA algorithm

The RSA algorithm consists of two main stages:

- Key generation
- The RSA function f

This first step involves the generation of large secret prime numbers. This second step uses modulo exponentiation to encrypt or decrypt the message. Before discussing in details these two steps, an overview of the techniques and concepts involved in RSA cryptographic system is warranted. These two stages correlate with the two laboratory exercises. The fundamental task of RSA's first steps is to generate two useful prime numbers.

2 PRIME NUMBER GENERATION

Prime numbers are an infinite set of integers with the property that they have no factors but themselves and one. A non-prime number is called a composite. Integers that are relatively prime (also known as co-prime) have a greatest common divisor of one. There several ways to identify prime numbers, with the predominate techniques being deterministic (usually using a "sieving" technique of eliminating composites from a range of values) and probabilistic methods, that evaluate the primality of a number based on a series of tests that reduce the error of false identification.One limitation of RSA is that it requires time and computing resources to find sufficiently large primes, which could limit its use on certain devices with limited processing power, storage or difficulty representing large integers. A strength is the speed and simplicity of encryption and decryption, which involves raising a number (the message) to a power (the key) and then taking the modulus of a known value (the product of the two secret primes). Finding these primes is the first stage.

2.1 Eratosthenes' sieve

Deterministic methods include the ancient technique of the sieve of Eratosthenes, which is suitable for small primes but takes polynomial time to compute, and so is non-optimal for large primes, such as the large magnitude primes used in RSA. A simple implementation used during development, but not included in the final laboratory solutions, is shown in Code Listing 10 in Appendix I.

2.2 Rabin-Miller test

The method implemented in this project was the Rabin-Miller test, which uses a series of random or predetermined *bases* as "witnesses" to the number being tested as either being composite, or probably-prime. The code for the implementation is shown in in Code Listing 5 in Appendix I. The witnesses chosen in the implementation are random integers smaller than the number being tested. According to [4] it is sufficient to use the primes 2, 7 and 61 as witness to confirm primeness of any tested 32-bit number. Similar techniques exist using the first few primes for larger bitsize numbers. To meet the lab project specifications, the primes used for the implementation presented only use a random selection method. To avoid the pitfalls of psuedo-random generation function `randi()`, the engine is seeded by the current time on every call for a new random number inside the main function test loop. MATLAB's native `isprime()` is used as an `assert` condition, but not as a direct test. In general, Rabin-Miller is considered to be fast and accurate, with a confidence of 4^k with k bases5.

2.3 Implementation of generatePrime()

The implemented method to generate primes, `generatePrime( bitsize, <coprime>)` is shown in Code Listing 4 in Appendix I). The function proceeds as follows: a random number is chosen, of a specified input argument bitsize. This number is then tested 7 times (7 random bases as witnesses) with the `RabinMiller()` function, to confirm its composite nature (in which case it is rejected, and new random integer in the range is tested) or to confirm its probable primality. Optionally, a co-prime number (such as the totient n, the Euler Totient of two large primes) can be provided as a condition for relative primeness. The probably prime number is returned as output.

3 Greatest Common Divisor

The gcd between two integers, is the largest non-zero factor that they share (that divides both numbers without remainders.) This function was implemented in MATLAB as `GCD(a,b)` shown in Code Listing 6 in Appendix I.

4 Modular Mathematics

Modulo is an integer division method developed by Gauss[5] in 1801, but with similar properties to Sun Zi's Chinese Remainder Theorem from 3 CE, and the incommensurability proofs attributed[6] to the eponymous Theaetetus in Plato's dialogue from 4 BCE. A positive integer is divided by another positive integer, returning the remainder as the result. For example, in the equation

$$x \ mod(p) = c$$

the result c is the remainder (or residue) left after the Euclidean division x/p.

4.1 Modular exponentiation

Modular exponentiation is the operation of raising an integer to the power of an exponent, with modulus n, i.e

$$a^b mod(n)$$

Reference [7] describes a memory-efficient algorithm with reduced steps for modular exponentiation, utilising the property that

$$a^b = \begin{cases} (a^2)^{b/2} & \text{if } b \text{ even, } b > 0 \\ (a) * (a^2)^{(b-1)/2} & \text{if } b \text{ is odd} \\ 1 & \text{if } b = 0 \end{cases}$$

This method allows repeated iteration down through the exponent, without having to calculate or store the actual value of a^b, which in the case of both a and b being very large numbers, can easily result in overflows in the naive integer representation of the value. The function `moduloExponentiation(a,b)` is implemented in MATLAB as shown in Code Listing 3 in Appendix I. The function returns the result of the modulo as output.

4.2 Integer representation in MATLAB

MATLAB's default class to represent numbers is the double-precision floating point `double`. Although there is an extended, variable precision tool-set, `vpa`, it has the trade-off of decreased speed of code execution. Typecasting or casting values into unsigned integers of a specific bit-size can offer precise control of the maximum integer value available, but native support is only offered up to unsigned 64-bit values with `uint64()` but generating or factoring primes larger than this are not performant in MATLAB. The software application's limited ability to represent and perform operations on large integers (which are typically between 256-bit to 4096-bit in practical RSA applications) substantially limits MATLAB's suitability for RSA implementations to the domain of educational experimentation, or purely conceptual understanding of the system. For an industrial or professional implementation of RSA, a more suitable framework such as SAGE, or programming in a language such as Python or C++, or a library that allows the use and manipulation of very large integers would be a better choice. For the purposes of this lab, given the limitations of MATLAB's ability to represent large integers, a trade-off of speed and complexity was made: prime generation was limited to below 24-bit numbers, and the practical exercise for the decryption/encryption process only used 8-bit keys, with 12- and 14-bit primes used for the generation of the totient function. For example, even with modular exponentiation optimisation, the value of each $base * base$ in the modulo operation, which occurs inside the exponent iteration loop could still overflow, resulting in an inaccurate result. In Matlab `double` saturates rather than overflows, but the inaccurately encrypted text is unrecoverable in both situations.

5 Euler's Totient

Euler Totient function is a method to find the multiplicative inverse in a set of modulo integers. The totient of the product n of two primes p, q is trivial to calculate:

$$\varphi(n) = (p-1)(q-1), \text{ where } n = pq$$

In the key generation process, the totient of the product of the two chosen large primes should be co-prime with the private key. It is important to keep the composite product n secret, as revealing the information of its value will allow easy breaking of the cipher, by finding the factors which are the original primes. Because the calculation of n is simple when p and q are prime, no separate function was written for this process. An example in the implementation presented is shown in line 26 in Code Listing 1 in Appendix I.

6 Extended Euclidean Algorithm

This algorithm computes the GCD of two numbers, and extends the classical Euclidean algorithm by back substitution of factored terms to express the GCD result in terms of multiples of the two input values. This additional back substitution can return the multiples as two coefficients which solve the equation

$$as + bt = gcd(a, b)$$

This allows the calculation of the multiplicative inverse of modulo operations, (often called the inverse modulo for

short) using the condition of relative primeness between a and b that states

$$as + bt = 1 \implies qt = 1 mod(b)$$

Thus if you calculate the coefficients s and t, the inverse modulo of $amodulusb$ is s, where $a * smodulusb = 1$. The calculation of the extended euclidean algorithm is implemented as `inverseMod(a,b)` in Code Listing 7 in Appendix I.

7 Encryption and Decryption

In RSA, although the keys are not symmetric, alternating use of the two keys result in successive encryption/decryption. The difference is that one key e is public, and the other, d, is kept private. Both encryption and decryption use the modular exponentiation function, as follows:

$$\text{Encryption} \rightarrow \text{m}^e mod(n)$$

$$\text{Decryption} \rightarrow c^d mod(n)$$

where m is the plaintext message, and c is the ciphertext, and n has the meaning from above (the product of the two secret prime p and q).

8 Exercise 1

The code for Exercise 1 is shown in Code Listing 1 in Appendix I. Two 16-bit primes, p and q were generated using the function `generatePrime()`. The bitsize was limited to 16-bits for speed of generation and for simplicity of numerical representation in MATLAB. In actual practice the size of the prime should be between 256 to 4096 bits, for greater security. The product n and the totient of n were calculated. The size of the prime should be of similar magnitude, to make factoring them out of the known value n very difficult. Public key s was generated using the condition of relative primeness to the totient. The private key h, was found as the multiplicative inverse of the public key e with modulus n. The function prints the found values in decimal and hex, along with their binary lengths to the command window.

9 Exercise 2

The code for Exercise 2 is shown in Code Listing 1 in Appendix I. Exercise 2 used values for primes, keys, n and $\varphi(n)$ generated using Exercise 1, with smaller bit lengths to avoid overflow and saturation, while retaining simple coding methods.

1000 random integers were generated, within the range $[16384, 32767]$. This is the decimal range of 15-bit integers. These numbers were encrypted using the `moduloExponentiation` function, which raises the current random integer m to the power of the public key s, modulus n.

$$\text{Encryption} \rightarrow \text{m}^s mod(n)$$

This process was timed, using the MATLAB functions `tic` and `tic`. This process took an mean time of $\approx$ 23 milliseconds on an i7 MacBook Pro laptop with 16GB of RAM.

The array of encrypted numbers were then decrypted using the private key. Using the `moduloExponentiation` function, raising each separate encrypted number c to the power of the private key h, modulus n. The total time to decrypt the numbers was timed on the same hardware and timing tools as above. The decryption took an mean time of $\approx$ 13 milliseconds. This confirms a core strength of RSA, the speed and simplicity of encryption and decryption. By contrast, generating the large primes to create the keys is significantly longer task. The security of RSA lies in the computational difficulty and time to attempt to factor out these primes from the known n.

10 Additional code

Minor functional utilities were also written to perform simple tasks and to conform inputs to the main functions. These include `makeInt`, shown in Code Listing 8 in Appendix I, which checks and formats an input number into an integer. A small function, `bitlength`, shown in Code Listing 9 in Appendix I, returns the length of the binary representation of an input number.

Conclusion

The basic RSA stages were implemented using MATLAB, with constrained size of prime number generation due to the limited ability of MATLAB to represent the product of extremely large integers, within the context of an educational exercise. The Miller-Rabin test was chosen to test a random integer for primality. Two primes p and q were generated, and their product n was calculated. A public key e was generated, with the condition that e should be co-prime to the totient ($\varphi(n)$. The private key d, is found as the multiplicative inverse of the public key e with modulus n. 1000 random numbers were encrypted using the public key, and then decrypted successfully using the private key. The details and methods of the implementation were discussed.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. M. Adleman, "Cryptographic communications system and method," September 1983, US Patent 4,405,829.

[2] R. Crandall and C. B. Pomerance, *Prime numbers: a computational perspective*. Springer Science & Business Media, 2006, vol. 182.

[3] S. Nisha and M. Farik, "Rsa public key cryptography algorithm–a," *International Journal Of Scientific and Technology Research*, vol. 6, no. 7, pp. 187–190, July 2017.

[4] M. O. Rabin, "Probabilistic Algorithm for Testing Primes," *Journal of Number Theory*, no. 12, pp. 128–138, 1980.

[5] B. Vallée, "Gauss' algorithm revisited," *Journal of Algorithms*, vol. 12, no. 4, pp. 556–572, 1991.

[6] T. Gowers, J. Barrow-Green, and I. Leader, *The Princeton companion to mathematics*. Princeton University Press, 2010.

[7] B. Schneier, *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, 2007.

APPENDIX I

EXERCISE 1

```
% ELEN3015 Lab3 Exercise 1
% Tyson Cross 1239448

clc
clear all

% keylengths
keylength = 24;                     % should be larger than (p-1), and (q-1)
plength = 16;                       % p and q should be of similar length
qlength = 16;

% primes
tic;
stop = false;
disp('Hunting for primes...')
while ¬stop
    p = generatePrime(plength);
    q = generatePrime(qlength);
    if gcd(p,q)==1
        stop = true;
    end
end

% Euler totient
n = q*p;
totient = (p-1)*(q-1);

% keys
stop = false;
while ¬stop
    s = generatePrime(keylength, totient);
        assert(gcd(s,totient)==1, 'Public key s and the totient are not coprime');

    h = inverseMod(s,totient);


    if (gcd(h,totient)==1) && (1==mod((s*h),totient))
        stop = true;
    end
end
primetime = toc;

%% output
clc
disp(['p = ', num2str(p), ';          % hex = ', num2str(dec2hex(p)),...
    '  [', num2str(length(dec2bin(p))), '-bit]'])

disp(['q = ', num2str(q), ';          % hex = ',  num2str(dec2hex(q)),...
    ' [', num2str(length(dec2bin(q))), '-bit]'])

disp(['n = ', num2str(n), ';          % hex = ', num2str(dec2hex(n))])

disp(['totient = ', num2str(totient), ';          % hex = ',...
    num2str(dec2hex((totient)))])

disp(['s = ', num2str(s), ';          % hex = ', num2str(dec2hex(s)),...
    ' [', num2str(length(dec2bin(s))), '-bit]'])

disp(['h = ', num2str(h), ';          % hex = ', num2str(dec2hex(h)),...
    ' [', num2str(length(dec2bin(h))), '-bit]'])
disp(' ')
disp(['(Calculations completed in ',...
    num2str(primetime), ' second(s))'])
```

Code 1: exercise1.m

```m
% ELEN3015 Lab3 Exercise 2
% Tyson Cross 1239448

clc
clear all

% values generated in previous exercise
% p = 157, hex = 9D  [8-bit]
% q = 211, hex = D3 [8-bit]
% n = 33127, hex = 8167)
% The totient Phi(n) = 32760 (7FF8)
% public key s = 2179, hex = 883 [12-bit]
% private key h = 8299, hex = 206B [14-bit]

p = 157;
q = 211;
n = p*q;
totient = 32760;
s = 2179;
h = 8299;

% generate 1000 random numbers to encrypt
bitsize = 15;
range = [2^(bitsize-1),2^bitsize-1];
numbers = [];
for i=1:1000
    numbers(i) = randi(range);
end

% encrypt numbers
tic;
c = [];
for i=1:length(numbers)
    c(i) = moduloExponentiation(numbers(i),s,n);
end
time_encrypt = toc;

% decrypt numbers
tic;
m = [];
for i=1:length(c)
    m(i) = moduloExponentiation(c(i),h,n);
end
time_decrypt = toc;

if ¬isequal(numbers,m)
        disp('Decryption failed!')
else
    str = {'Original';'Cipher';'Decrypt'};
    fprintf('% -10s -> % -10s -> % -10s\n', str{:});
    fprintf('-----------------------------------\n')
    for i=1:(min(length(c),length(m)))
    fprintf('% -10d -> % -10d -> % -10d\n', numbers(i), c(i), m(i))
    end
    disp(' ')
    disp(['The ', num2str(length(c)) , ' messages were encrypted in ', num2str(time_encrypt*1000), ' ...
        milliseconds'])
    disp(['The ', num2str(length(m)) , ' messages were successfully decrypted in ', ...
        num2str(time_decrypt*1000), ' milliseconds'])
end
```

Code 2: exercise2.m

```
function [ output ] = moduloExponentiation( base, exponent, modulus )
%  moduloExponentiation calculates (a^b)%c
%  This algorithm uses the following logic to reduce computational time to O(log2(b))
%
%            {   (a^2)^(b/2)          if b is even and b > 0
%      a^b = {   a*(a^2)^((b-1)/2)    if b is odd
%            {   1                    if b = 0

% ELEN3015 Lab3
% Tyson Cross 1239448

    if exponent == modulus
        output = 0;
        return
    end

    % Algorithm only useful with integer modulo values
    base        = makeInt(base);
    exponent    = makeInt(exponent);
    modulus     = makeInt(modulus);

    largest_int = realmax;

    output      = 1;
    base        = mod(base,modulus);                    % ≤ overflow

    while (exponent > 0)
        if mod(exponent,2) == 1                         % b is odd
            output = mod(output*base,modulus);
            assert((base < largest_int) && (output < largest_int) && ¬isinf(output), 'Overflow');
        end
        base = mod(base*base,modulus);
        assert(base < largest_int && ¬isinf(base), 'Overflow');
%         exponent = bitshift(exponent,-1);
        exponent = floor(exponent/2);                   % floor() slightly faster than bitshift()
    end

    output = double(mod(output,modulus));
end
```

Code 3: moduloExponentiation.m

```
function [ prime ] = generatePrime( bitsize , coprime )
% generatePrime generates an odd number likely to be prime
% and then tests this number's primality using the Rabin-Miller test

% ELEN3015 Lab3
% Tyson Cross 1239448

    if nargin < 2
        coprime = 1;
    end

    if bitsize <3
        error('Bitsize must be > 3')
    end

    start = bitsize-1;
    range = [(2^start)+1,(2^bitsize)-3];
    r = randi(range)+2;

    found_prime=false;
    while ¬found_prime
            if RabinMiller(r) && (GCD(r,coprime)==1) && bitlength(r)==bitsize
                prime = r;
                found_prime = true;
                stop = true;
            end
        if ¬found_prime
            r = randi(range)+2;
        end
    end
    assert(isprime(prime)==true,'RabinMiller false positive')
    assert(length(dec2bin(prime))≤bitsize,'Incorrect bitsize')
end
```

Code 4: generatePrime.m

```
function [ primality, numBinaryDivisions, oddPartNum ] = RabinMiller( n , k )
% RabinMiller tests an input number for primality using the Rabin-Miller test
% returns false if the test returns composite, and true if likely prime

% ELEN3015 Lab3
% Tyson Cross 1239448

    if nargin < 2
        k = 7;      % confidence of testing (number of witnesses/bases)
    end

    n = abs(makeInt(n));

    % shortcuts for n < 6
    switch n
        case 0
            primality = false; return
        case 1
            primality = false; return
        case 2
            primality = true;  return
        case 3
            primality = true;  return
        case 4
            primality = false; return
        case 5
            primality = true;  return
    end

    % check if n is even
    if bitand(n,1)==0
        primality = false; return
    end

    % check if n is multiple of 3 or 5
    if mod(n,3)==0
        primality = false; return
    elseif mod(n,5)==0
        primality = false; return
    end

    % split into the form (2^r)*d , with d odd
    numBinaryDivisions = 1;
    oddPartNum = (n-1)/2;
    while bitand(oddPartNum,1)==0
        oddPartNum = oddPartNum/2;
        numBinaryDivisions = numBinaryDivisions + 1;
    end

    % "Witness" loop
    primality = true;
    rng('shuffle');
    for i=1:k
        randomNum = randi([2, n-1]);            % alternatively, we could use the first k primes
        randomNumPower = moduloExponentiation(randomNum,oddPartNum,n);
        if (numBinaryDivisions==1) && (randomNumPower≠1) && (randomNumPower≠n-1)
            primality = false; return
        elseif (randomNumPower==1) || (randomNumPower==n-1)
            % Stop the loop, leave result as true (might be a strong liar)
            % Don't start j loop, continue with other witnesses (in outer for loop)
        else
            % Cannot make prediction for this witness yet. Start squaring randomNumPower mod n
            for j=1:numBinaryDivisions-1
               randomNumPower = moduloExponentiation(randomNumPower,2,n);
               if randomNumPower==1
                   % Definitely not a prime
                   primality =  false; return
               elseif randomNumPower==(n-1)
                   % Probably prime, but might be strong liar: need to check rest of witnesses
                   break
               end

               if j == (numBinaryDivisions-1)
                   % if we left the j loop without breaking
                   % then current randomNum is a witness
                   % to n as a composite (non-prime)
                   primality=false; return
               end
            end
        end
    end
end
```

Code 5: RabinMiller.m

```
function [ gcd, sa, ta] = GCD( a, b )
% GCD() returns the greatest common divisor between two numbers
% It implements Euclid's extended algorithm

% ELEN3015 Lab3
% Tyson Cross 1239448

    q = 1;
    r = 1;
    sa = 1;
    sb = 0;
    ta = 0;
    tb = 1;

    while b≠0;
        q=floor(a/b);
        r=a-q*b;
        a=b;
        b=r;
        sc=sa-q*sb;
        sa=sb;
        sb=sc;
        tc=ta-q*tb;
        ta=tb;
        tb=tc;
    end

    gcd = a;
end
```

Code 6: GCD.m

```
function [ inverse ] = inverseMod( a, b )
% inverseMod() returns the inverse modulo, using extended euclidian algorithm

% ELEN3015 Lab3
% Tyson Cross 1239448

    [gcd, x, y] = GCD( a, b);

    if gcd ≠1
        error('Modular inverse does not exist')
    else
        inverse = mod(x,b);
    end
end
```

Code 7: inverseMod.m

```
function [ output ] = makeInt( input )
% makeInt checks the input to see if it is an integer.
% if not, the value is floored to the nearest int and returned

% ELEN3015 Lab3
% Tyson Cross 1239448

    if isnan(input)
        error('Value cannot be NaN')
    end

    if isinf(input)
        error('Value cannot be Inf')
    end

    if mod(input,1)==0
        output = input;
    else
        output = floor(input);
        warning(['Using integer value of ''', num2str(output),...
            ''' in place of ''', num2str(input), ''' '])
    end

end
```

Code 8: makeInt.m

```
function [ x ] = bitlength( value )
% bitlength() returns the length in bits to represent an input value

% ELEN3015 Lab3
% Tyson Cross 1239448

x = length(dec2bin(value));
end
```

Code 9: bitlength.m

```
function [ primes ] = sievePrimes( limit )
% sievePrimes() is an implementation of the Sieve of Eratosthenes
% Iteratively marks composites as 0, then copies all remaining (unmarked) primes to a new array

% ELEN3015 Lab3
% Tyson Cross 1239448

    number_range = 2:limit;
    assert(length(number_range)>2, 'Range length must be greater than 2')

    n = 1;

    while n.^2 < limit
        number_range(n^2:n:limit) = 0;
        n = find(number_range>n, 1);
    end

    primes = number_range(number_range>1);
end
```

Code 10: sievePrimes.m