Digital Signature using RSA

Overview

The aim of this assignment was to implement a digital signature using RSA. Before the digital signature could be implemented, an appropriate public/private RSA key pair was set up in the following way:

1. Two large, distinct probable primes "*p*" and "*q*" were generated.
2. *N = p*q* was calculated.
3. The Euler totient function - *phi(N)* - was computed.
4. The public encryption exponent *e* is a constant. *phi(N)* is tested to make sure that it is relatively prime to *e*. If it is not, steps 1-4 are repeated with different values for *p* and *q* until *phi(N)* is relatively prime to *e*.
5. The private decryption exponent *d* is calculated using the Extended Euclidean GCD algorithm – *d* is the multiplicative inverse of *e(mod phi(N))*.
6. A decryption method which computes *c^d (mod N)* is provided. This is a case of modular exponentiation and a portion of the code from Assignment1 was reused here.
7. The main class (a text file) of the assignment is read in and a 256-bit digest is produced for it.
8. The digest is then encrypted using the private key: *m^d (mod N)*.
9. The ciphertext can be decrypted using the public key e: *c^e (mod N)* to recover the message digest.

Algorithms

The algorithms described below are linked to the steps in the Overview section above.

phi – used in step 3.

This function computes the Euler's totient function of two different primes. Since we're dealing with two different primes, we know they will be coprime to each other, with the Greatest Common Divisor value 1. Because of this we can use a simplified, faster version of the Euler's totient function that works on relatively prime numbers, where *phi(N) = (p – 1)(q - 1)*.

Are two numbers Relatively Prime? – used in step 4.

This function checks whether two BigIntegers are relatively prime. It utilises the Greatest Common Divisor (GCD) method and compares the result to the value 1. If the GCD of the two numbers is 1, they are coprime. The GCD method is discussed in its own subsection - "GCD (Greatest Common Divisor)".

GCD (Greatest Common Divisor) – used in step 4.

This is an application of the Euclidean algorithm, but instead of substitution, like in the original algorithm, it uses Euclidean division which is more efficient.

The function returns the Greatest Common Divisor of two BigIntegers.

Multiplicative Inverse – used in step 5.

The multiplicative inverse function is implemented using the Extended Euclidean algorithm. It is possible to apply the Extended Euclidean algorithm here because the two values that are used for the calculation (*e* and *phi(N)*) are known to be coprime – that test has already been carried out before (in Step 4.).

For coprime BigIntegers, *e* and *phi(N)*, the function returns the multiplicative inverse of *e* modulo *phiN* – exactly what is required for RSA.

Modular Power – used in steps 6. 8. and 9.

This function takes in a base, an exponent and a modulus and computes (base^exponent) % m. It is a slightly modified version of the modularPower algorithm used in Assignment1 – it can now handle negative exponents. This function is used to encrypt and decrypt messages, where the exponents are equal to *e* and *d*. While *e* is always positive (since it was provided for this assignment), *d* can be negative for certain values of *phi(N)* – the function was modified to handle negative *d* (*d* is the private key).

If the exponent is negative, the base becomes the multiplicative inverse of base modulo m and the exponent is taken as an absolute value. The rest of the modular power function proceeds as before and utilises what is known as "exponentiation by squaring":

Given a prime p and $a \in \mathbb{Z}_p^*$ we want to calculate $a^x \pmod{p}$.
Denote x in binary representation as $x = x_{n-1}x_{n-2}\ldots x_1 x_0$ where:

$$x = \sum_{i=0}^{n-1} x_i 2^i$$

Therefore, $a^x \pmod{p}$ can be written as:

$$a^x = a^{2^{(n-1)}x_{n-1}} a^{2^{(n-2)}x_{n-2}} \ldots a^{2x_1} a^{x_0}$$

Algorithm:

```
r:=1
for i := n - 1 downto 0 do
    r := r^2 a^x_i mod p
```

Hash String – used in step 7.

This function takes in a message and uses SHA-256 to compute and return the digest of that message in the form of an array of bytes. Standard Java MessageDigest library is used to implement this.

Read Text File – used in step 7.

The program has to create a digest of its own code. To facilitate this, an exact copy of the program is kept in a separate text file and that copy is used to create the digest. This function takes in the location of that copy on the disk as an argument and reads in the contents of this file.

It is important to note that all whitespace is ignored and the code is returned as a single String. It would be equally valid not to ignore whitespace, but in that case, there would be a possibility of

doubling up or including extra spaces during submission or testing. It is easier to spot unwanted content if the message is a single, one-lined string.

Digest & Encryption

The message digest is acquired using SHA-256, with no padding. The message that is used is a string with no whitespace that contains all code of the application on a single line.

BigIntegers *d* and *e* are used to encrypt and decrypt messages. *d* is the private key in this arrangement while *e* is the public key and *e* = 65537.

message^key (mod *N*) converts the message to ciphertext and vice versa.

As discussed before, when encrypting, the message is a digest of the code with no whitespace, the key being our private *d*. This gives us the ciphertext *c*:

messagedigest ^ *d* (mod *N*) = *c*

Public key *e* is used to decrypt the ciphertext:

c ^ *e* (mod *N*) = messageDigest

Issues & Solutions

Two issues were encountered.

The first issue involves the Modular Power function with negative exponents. The original function from Assignment 1 did not have to handle negative powers and had to be adjusted to do so. This has been resolved fully, as outlined in the Algorithms' "Modular Power" section.

The second issue involves the SHA-256 hash function scheme that is used to generate the digest. This scheme produces negative digests for certain inputs. Unfortunately, RSA was designed to work only on positive representations of messages. Luckily, the code that is used produces a positive digest and the issue does not manifest itself. A more "general" solution would be to use the absolute value of the digest whenever a negative value is encountered.

References

- Geoff Hamilton, Assignment2 specification – http://www.computing.dcu.ie/~hamilton/teaching/CA547/Assignments/assignment2.html
- Geoff Hamilton, Number Theory – http://www.computing.dcu.ie/~hamilton/teaching/CA547/notes/Number1.pdf
- Geoff Hamilton, Public Key Cryptography – http://www.computing.dcu.ie/~hamilton/teaching/CA547/notes/PublicKey.pdf
- Wikipedia, RSA – http://en.wikipedia.org/wiki/RSA_(algorithm)
- Wikipedia, Euclidean algorithm – http://en.wikipedia.org/wiki/Euclidean_algorithm
- Wikipedia, Extended Euclidean algorithm – http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm
- Wikipedia, Euler's totient function – http://en.wikipedia.org/wiki/Euler's_totient_function

- Wikipedia, Modular Exponentiation – http://en.wikipedia.org/wiki/Modular_exponentiation