CS/COE 1501 Assignment 5

Released: Friday, April 5

Due: Tuesday, April 23, 11:59 PM

Goal

To get hands on experience with algorithms to perform mathematical operations on large integers, using RSA as an example.

Important note: The result of this project should NEVER be used for any security applications. It is purely academic. Always use trusted and tested crypto libraries!

High-level description

You will be writing two programs. The first will generate a 512-bit RSA keypair and store the public and private keys in files named `pubkey.rsa` and `privkey.rsa`, respectively. The second will generate and verify digital signatures using a SHA-256 hash. You will use Java's MessageDigest class to complete this project. In order for either of these programs to work, however, you will need to complete an implementation of a class to process large integers.

Specifications

1. You are provided with the start of a class to process large integers called `LargeInteger`. `LargeInteger` objects are represented internally as two's-complement *raw integers* using byte arrays (i.e., instances of `byte[]`).
    1. Currently, `LargeInteger` has the following operations implemented:
        - A constructor to generate an n-bit random, positive, probably prime integer using a specified source of randomness. This constructor uses a probabilistic primality test to ensure that it is probably prime (with 2^-100 chance of being composite).
        - A constructor that creates a new `LargeInteger` object based on a provided `byte[]`.
        - A method to compute the sum of two `LargeInteger` objects.

- A method to determine the negation of a `LargeInteger` object.
- A method to compute the difference of two `LargeInteger` objects.
- Several other helper methods.

2. Due to the use of a two's complement representation of the integers, `LargeInteger` objects should always have at least one leading 0 bit (indicating that the integer is positive) in their `byte[]` representation. This property may cause the array to be bigger than expected (e.g., a 1024-bit generated prime will be represented using a length 129 byte array).
3. LargeIntegers are represented using a *big-endian* byte-order, so the most significant byte is at index 0 of the `byte[]`.
4. In order to generate RSA keys and perform RSA encryptions and decryptions, you will further need to implement the following functions:
   - `LargeInteger multiply(LargeInteger other)`
   - `LargeInteger[] XGCD(LargeInteger other)`
   - `LargeInteger modularExp(LargeInteger y, LargeInteger n)`
   - Any additional helper functions that you deem necessary.
5. You may *not* use any calls the Java API class `java.math.BigInteger` or any other JCL class when writing `LargeInteger`. The probably-prime `LargeInteger` constructor calls `BigInteger`'s `probablePrime` method; this is the only call allowed to `BigInteger` in your `LargeInteger` class.

2. Once `LargeInteger` is complete, write a program named `RsaKeyGen` to generate a new RSA keypair.

   1. To generate a keypair, follow the following steps, as described in lecture.
      1. Pick p and q to be random primes of the appropriate size to generate a 512-bit key
      2. Calculate n as p*q
      3. Calculate ϕ(n) as (p-1)*(q-1)
      4. Choose an e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$ (e must not share a factor with ϕ(n))
      5. Determine d such that $d = e^{-1} \bmod \phi(n)$
   2. After generating e, d, and n, save e and n to `pubkey.rsa`, and d and n to `privkey.rsa`.

3. Once you have your RSA keys generated, write a second program named `RsaSign` to sign files and verify signatures. This program should accept two command-line arguments: a flag to specify whether to sign or verify (`s` or `v`), and the name of the file to sign/verify.

   1. If called to sign (e.g., `java RsaSign s myfile.txt`) your program should:

      1. Generate a SHA-256 hash of the contents of the specified file (e.g., `myfile.txt`).

2. "Decrypt" this hash value using the private key stored in `privkey.rsa` (i.e., raise the hash value to the d power mod n).
    - Note: Your program should exit and display an error if `privkey.rsa` is not found in the current directory.
3. Write out the signature to a file named as the original, with an extra `.sig` extension (e.g., `myfile.txt.sig`).

2. If called to verify (e.g., `java RsaSign v myfile.txt`) your program should:
    1. Read the contents of the original file (e.g., `myfile.txt`).
    2. Generate a SHA-256 hash of the contents of the original file.
    3. Read the signed hash of the original file from the corresponding `.sig` file (e.g., `myfile.txt.sig`).
        - Note: Your program should exit and display an error if the `.sig` file is not found in the current directory.
    4. "encrypt" this value with the key from `pubkey.rsa` (i.e., raise it to the e power mod n).
        - Your program should exit and display an error if `pubkey.rsa` is not found in the current directory.
    5. Compare the hash value that was generated from `myfile.txt` to the one that was recovered from the signature. Print a message to the console indicating whether the signature is valid (i.e., whether the values are the same).

Submission Guidelines

- **DO NOT** upload any IDE package files.
- You must name your key generation program `RsaKeyGen.java`, and your signing/verification program `RsaSign.java`.
- You must be able to compile your program by running `javac RsaKeyGen.java` and `javac RsaSign.java`.
- You must be able to run your key generation program by running `java RsaKeyGen`, and your signing/verification program with `java RsaSign s <filename>` and `java RsaSign v <filename>`.
- You must fill out `info_sheet.txt`.
- The project is due at 11:59 PM on Tuesday, April 23. Upload your progress to Box frequently, even far in advance of this deadline. **No late assignments will be accepted.** At the deadline, your Box folder will

automatically be changed to read-only, and no more changes will be accepted. Whatever is present in your Box folder at that time will be considered your submission for this assignment—no other submissions will be considered.

Additional Notes/Hints

- An example of using `java.security.MessageDigest` to generate the SHA-256 hash of a file is provided in `HashEx.java`
- You may find the creation of `pubkey.rsa`, `privkey.rsa`, and signature files to be most easily accomplished through the use of `java.io.ObjectOutputStream`. The format of your key and signature files is up to you.
- **NEVER USE CODE FROM THIS PROJECT IN PRODUCTION CODE.** This is purely instructive. Always use trusted and tested crypto libraries.

Grading Rubric

`LargeInteger`

Feature	Points
`multiply`	20
`XGCD`	25
`modularExp`	10

Key generation

Feature	Points
p and q are generated appropriately	3
n and ϕ(n) computed appropriately	3
e is selected appropriately	4
d is selected appropriately	5
Key files are generated appropriately	5

Signing

Feature	Points
Hash is generated correctly	2
Hash is "decrypted" (signed) correctly	5
Signature file is generated appropriately	3

Verification

Feature	Points
Hash is re-generated correctly	2
Signature is "encrypted" (verified) correctly	5
Signed files are appropriated verified	3

Other

Feature	Points
Assignment info sheet/submission	5