CS 5010: Programming Design Paradigms

Fall 2017

HW 5 – Data Collections, RSA Cryptographic Algorithm, Digital Signatures, and Online Transctions

Assigned: Monday, October 23, 2017, Due: Monday, October 30, 2017 by 6pm

College of Computer and Information Science
Northeastern University – Seattle

Assignment Goals

The purpose of this assignment is to give you additional practice and understanding of:

- Various data collections, and differences between them, in particular:
  - Trees (binary search and AVL trees)
  - Dictionaries
  - Hash tables and maps
- Efficiencies of operations (e.g., traversal) on different data collections

As a bonus, you will get a quick (and simplified) introduction to public key cryptography, and to cryptographic digital signatures.

Background

Now days, many of us are familiar, and (for the most part) comfortable performing various online transactions in our everyday lives. For example, we buy stuff online, take care of our finances online, participate in petitions, even submit our tax returns online. A big security challenge in many such online applications is: (a) confirming that you are who you claim to be (with an authority to perform certain actions), and (b) making sure that the message, containing the description of an action that you want to perform, reaches the side tasked to perform that action exactly as you sent it.

To see why and how are those two requirements important, let's consider an online banking example. If your bank did not have a way to confirm your identity, then anyone could claim that they are you, and would be able to log into your bank account. That would be pretty bad. It would be even worse if some malicious entity (attacker) had a way to alter the messages that you are sending to your bank. For example, imagine that you just sent an online request to your bank to send a check to your landlord for the November's rent, and an attacker intercepted your message, and altered it so that the bank receives a request for two checkes - one to the landlord, and the other requesting that $10 000 are send to the attacker. Pretty bad...

To prevent many such uncomfortable things from happening, most online services rely on **digital signatures**. Digital signatures are a **public key-based** cryptographic mechanism that allows a bank, an online retailer, or the IRS to do two things simultaneously: verify your identity, and verify your message.

Public key cryptography (in a nutshell): To show how is such a simultaneous verification possible, let's review the concept of public key cryptography. There exist many well-established public key cryptographic algorithms, but all of them have one thing in common: they all consist of two keys: **a public key**, PK and a private (secret) key, SK.

Digital signatures (in a nutshell): Let's assume you want to digitally sign your message to the bank, m. To do so, you would take your message, and you would then perform some cryptographic operation f on that message using your **private key**, SK. You would then append your digital signature to your original message, $(m, f(m, SK))$, and send that pair to your bank.

To verify your identity, and the content of your message, the bank would then take your public key, PK, and apply some related cryptographic function, $g(m, f(m, SK), PK)$, on the received pair. If the verification returns `true`, the bank would know that it is indeed you sending the message (because, presumably, you are the only one who knows your private key), and that the message hasn't been altered *en route*.

Your Task

In this assignment, your task is to simulate the described identity and message verification process in some typical, but imaginary, financial institution, **Secure Bank, N.A.**
You will create a program, `SecureBankVerificationSimulator`, that takes four input arguments: **the number of unique bank clients, the number of unique verifications, a fraction of invalid messages** and **an output file**. Additional information about these arguments is provided in the next section.

The simulator then randomly creates the requested number of unique clients, and the requested number of unique pairs `(message, digital signature)`, taking into account parameter **fraction of incorrect message parameter**, as explained below. For each of the unique `(message, digital signature)` pair, it then simulates the verification process, and writes the simulation results into the provided output file.

Implementation Details

One of the goals of this assignment is to expose you to different data collections, including binary search trees, maps and hashes, and to allow you to explore the differences between them, in particular the difference in efficiency and memory requirements that some typical operations on those data collections may have. Different approaches might be possible, but in this assignment you want to find approaches that have a "reasonable" overall worst-case efficiency in terms of time complexity. For the purpose of this assignment, **"reasonable"** means that if there exist a data collection and/or an algorithm to perform an operation in constant (O(1) time), or in logarithmic time (O(log(n))), you should choose those approaches, and not an approach with a quadratic (O(n^2)) (or worse) complexity. If needed, you can also take memory requirements as a secondary criterion.

Processing Input Arguments: Your program, `SecureBankVerificationSimulator`, is expected to always take four input arguments:

1. **Number of unique bank clients -** an integer, bounded from above by 50 000 (inclusive). It represents the number of unique clients that the banks has, and that can use the bank's online services.
2. **Number of unique verifications -** an integer, bounded by 10 000 (inclusive). It represents the number of distinct digital signature verifications that the bank needs to perform.
3. **Fraction of invalid messages -** a real number, in the range [0, 1], representing the fraction of `(message, digital signature)` pairs either incorrectly generated on the client's side, or incorrectly received by the bank. In the given range, bound zero means that all of the `(message, digital signature)` pairs are correct, and bound 1 that all of the pairs are incorrect.
4. **Output file -** a string, representing the name of the output file.

The order of arguments should be as defined above, and if one of the arguments is missing, the program should terminate with an appropriate message.

Generating and Storing the Requested Number of Unique Bank Clients: An important observation about this assignment is that, while we are not building a networked system, **we are trying to simulate a system that consists of two logical components, a <u>bank</u>, and multiple <u>bank clients</u>**. These two components have different information available, as specified below.

Additionally, we are assuming that the bank, and all of the individual clients are using the **RSA-based digital signature** to secure the clients' online interaction with the bank. The RSA digital signature algorithm consists of three distinct phases, **key generation, digital signature generation**, and **signature verification**, and their details are described below.

With that in mind, for the purpose of your simulation, you will need to randomly generate the requested number of unique bank clients. Every bank client has:

- **A unique ID number -** this number should be a randomly generated integer, but it should be unique for every client (i.e., it should never be the case that two clients have the same ID). This number is known to both the client and the bank.

- **A (private key, public key) RSA pair** - this pair of integers should be generated using the RSA key generation algorithm, described below. Both keys are known to the client, but **only the public key is known to the bank**.
- **A deposit limit** - this number should be a randomly generated integer from the range [0, 2000], and it does not have to be unique for every user. The information about the deposit limit is only known to the bank.
- **A withdrawal limit** - this number should also be a randomly generated integer, but from the range [0, 3000]. It again does not have to be unique for every user, and it is only known to the bank.

Generating (message, digital signature) Pairs: To generate the requested number of `(message, digital signature)` pairs, your simulator should randomly choose the requested number of clients, and for every client, it should generate:

1. **A message**, where a message is a randomly generated integer from the set [0, 30000], and it does not have to be unique for every transaction. For simplicity, the last digit of the generated number represents the transactions that a client is requesting - numbers 0-4 mean that a client is requesting a deposit, whereas numbers digits 5-9 mean that a client is requesting a withdrawal. That last digit should be included into a computation of a digital signature.
2. **An digital signature**, where the digital signature is generated in two possible ways, depending on whether a message is valid or not. You should make a random determination of whether a message is valid or not, taking into account the input parameter **fraction of invalid messages**.
   - If a message is deemed invalid, then a digital signature should be just some randomly generated integer.
   - If a message is deemed valid, you should generate the corresponding digital signature using the RSA signature generation algorithm, described below.

Processing (message, digital signature) Pairs: Another important part of your program is the simulation of the steps that a typical bank would take to verify a received `(message, digital signature)` pair. In this implementation, we assume that the bank knows the IDs, the public keys, and the withdrawal and deposit limits of all the clients, but it does **not know the clients' private keys**.

So, every time the bank receives the `(message, digital signature)` pair from some client A who has a unique ID, ID_A, it has to find the client A's corresponding public key, PK_A. It then uses that public key to verify the received `(message, digital signature)` pair, using the RSA signature verification algorithm, described below.

If the signature verification process succeeds (i.e., it returns `true` as a result), the bank proceeds to validate a client's request. Based on the last digit of the received message, it determines whether a client wants to make a deposit or a withdrawal, and then based on the limits set for that clients, verifies if the received request can be fulfilled or not. Please note that the last digit of the message **should only be used for action determination, and not for amount determination.** For example, a message 15005 means that a client wants to withdraw \$1500 from their account, and message 23004 means that a client wants to deposit \$2300 to their account (for example, using quick pay or a check scan).

Note 1: Since the bank is expected to process a large number of `(message, digital signature)` pairs daily, it needs to be able to efficiently find the corresponding public key of the current client. **As a part of this assignment, you want to take seriously this requirement for efficient access, and encode the knowledge accessible to the bank in an appropriate data collection that will achieve that efficiency.**

RSA Key Generation, Digital Signature Generation, and Signature Verification: The RSA digital signature consists of three components, namely **Key Generation**, **Signature Generation**, and **Signature Verification**, described as follows:

RSA Key Generation

Some client, Alice, generates keys for the RSA digital signature scheme via the following procedure:

1. Alice generates two distinct large primes p and q, and then computes $\phi(n) = (p-1)(q-1)$, and $n = p \cdot q$.
2. Alice generates a random integer a that satisfy $\gcd(a, n) = 1$ and $\gcd(a, \phi(n)) = 1$, where gcd denotes a greatest common divisor.
3. Alice computes b such that $ab \equiv 1 \bmod \phi(n)$.
4. Alice's public key PK_A is given by (b, n). Her private key is $SK_A = (a, n)$, she keeps it as a secret.

Note 2: The public key and private key for the RSA digital signature are generated in the same way as the public and private keys for RSA encryption.

RSA Signature Generation

Alice generates a signature on a message m by computing:

$$sig_{SK_A}(m) = m^a \bmod n.$$

Note 3: the RSA signature generation is similar to encryption in the RSA cryptosystem. The only difference is that Alice signs the message using **the private key SK_A instead of the public key PK_A**.
RSA signature generation is illustrated in Figure 1.

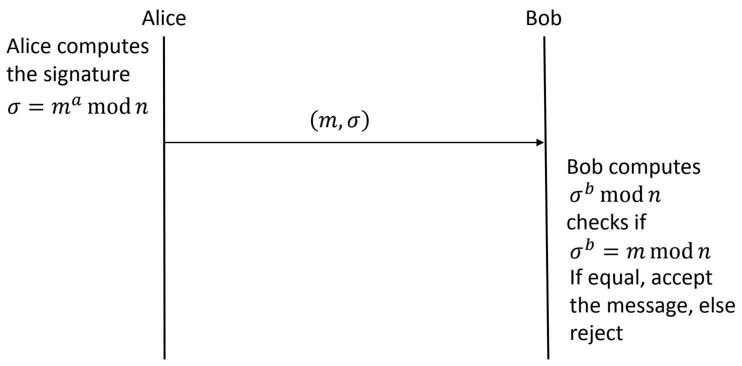
Alice
Bob
Alice computes the signature $\sigma = m^a \bmod n$
(m, σ)
Bob computes $\sigma^b \bmod n$ checks if $\sigma^b = m \bmod n$ If equal, accept the message, else reject

Fig. 1. RSA signature generation and verification.

RSA Signature Verification

Some bank, referred to as Bob, who receives a (message, signature) pair (m, σ) from Alice, verifies the message through the following procedure:

1. Compute $m' = \sigma^b \bmod n$.
2. If $m' = m$, **accept** the message (return `true`). Otherwise, **reject** (discard) the message (return `false`).

Note 4: RSA signature verification is analogous to decryption in the RSA cryptosystem, with the main difference that **Bob verifies the message using <u>Alice's</u> public key PK_A**.
RSA verification is illustrated in Figure 2.
In this assignment, you will need to implement all three steps of the RSA digital signature algorithm, key generation, signature generation, signature verification. In doing so, you may want to use classes `java.math.BigInteger` and `java.security.SecureRandom`.

Note 5: Please note that there exist many implementations of RSA algorithm online. You are allowed to consult online resources, but please cite any resources that you may use. More importantly, please be mindful because many of the implementations available online either are not correct, or do not conform with our established software development practice.

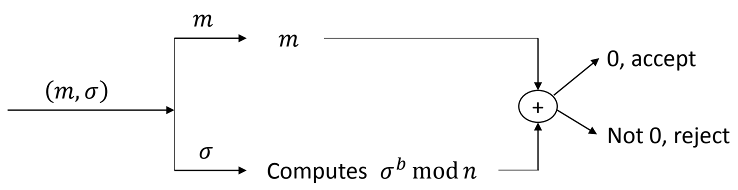

Fig. 2. RSA signature verification.

Generating the Output File: The results of your simulation should be stored in the provided CSV file (if the file does not exists, it should be created), and printed out on the console. The raw data, randomly generated during the simulation, should be stored into the CSV files, that will include the following header:

- Transaction number
- Date,
- Time,
- Client ID,
- Message,
- Digital signature,
- Verified,
- Transactions status.

For every processed `(message, digital signature)` pair, the information indicated in the header should be provided as a row in the file, where column `verified` can have only two values - **yes**, if the message was verified, and **no**, if it verification failed. Similarly, column `transaction status` can have only four values - **deposit accepted, deposit rejected, withdrawal accepted,** and **withdrawal rejected**.

The data printed on the console should include "metadata" generated from all of the processed transactions. Such a "metadata" represents some rudimentary information that a bank may use for security and fraud prevention purposes. Information provided on the console should include at least (your are encouraged to include more useful information):

1. The number of distinct transactions with different IDs, but the same public key (*this indicates how secure is the RSA key generation process*).
2. Top ten unique users with the largest number of transactions during the simulation.
3. List of all IDs with rejected deposit transactions.
4. List of all IDs with rejected withdrawal transactions.

Bonus Part - 2 points

In lecture 8, we will talk about **AVL trees**, binary search trees, with the property that the heights of two children subtrees of any node differ by at most one (**the AVL property**). Implement your own version of the AVL tree, and use it to generate, and store the requested number of unique bank clients, and to process the requested number of (message, digital signature) pairs. For your bonus approach, you are allowed to reuse all of the objects and interfaces that you wrote for the original approach.
(*Note: Once again, the possible bonus points do not reflect the time and effort that may be required to implement the bonus problem. Please focus on the the required part of the assignment first, and attempt the bonus part only after your have fulfilled all of the components of the required part.)*

What To Submit?

When submitting your assignment, you should continue using the same Maven archetype that we used in Assignments 2, 3, and 4. You will want to submit the following:

1. Class `SecureBankVerificationSimulator`. That class will be assumed to be the starting point of your program, and will be called from the command line with input arguments.

2. All classes that you have developed for this assignment.
3. All abstract classes and interfaces that you extended and implemented in this assignment.
4. All classes that you have developed to test your code.
5. A UML diagram, corresponding to the design of your program.
6. A brief write-up, which summarizes the main relations between your classes, and how does your program handle errors and/or exceptions. Additionally, in this assignment, you will want to explain which data collection have you decided to use for your unique clients, and which for (message, digital signature pairs). You will want to comment what are the advantages of your chosen data collections, as well possible drawbacks.
7. If you attempted the bonus question, please include class `SecureBankVerificationSimulatorAVL`, all of the classes it uses, and the new UML that corresponds to this program. Please also update your write-up, to comment on the pros/cons of this bonus approach, compared to your original approach.