Implementation of SPA

Yoan Martin

School of Computer and Communication Sciences

Semester Project

December 2017

Responsible
Prof. Serge Vaudenay
EPFL / LASEC

Supervisor
Ms. Handan Kilinç
EPFL / LASEC

Chapter 1

Advanced cryptography

Blind signature The blind signature [1] is an extension of the digital signature. It permits to sign a message without revealing the message to the signer. The concept can be hard to understand because it is difficult to guess why a signer would sign a message without knowing it. Let's consider this simple example. Imagine a political party which wants to hold an election. The party wants to authenticate each vote but each elector does not want his vote to be known. The blind signature is a solution to this problem.
For this project, we use a blind signature based on RSA. The idea is to use a blind factor to blind the message. So, the signer is unable to know the message.
The original message is represented as m and the signature as s. First, we add the blind factor r to the message.

$$m' = r^e m \mod N$$

Then, the signer signs the message using its private key.

$$s' = (m')^d \mod N = r^{ed} m^d \mod N = r m^d \mod N$$

Finally, we remove the blind factor using the inverse.

$$s = r^{-1} s' \mod N = r^{-1} r m^d \mod N = m^d \mod N$$

Nevertheless, the RSA blind signature is not perfect because it is vulnerable to blinding attack. This comes from the fact that RSA uses the same key to sign or to decrypt a message. Let's consider an example where Alice is a server. She has a single private key to sign and to exchange messages. Imagine that Bob sends a message m to Alice using Alice's public key.

$$m' = m^d \mod N$$

Now, imagine that Oscar wants to read this message. If Oscar can intercept the message from Bob, he can send it to Alice by asking to sign

it. Since Alice is using her private key to sign a message, Oscar gets the original message of Bob.

$$m = m'^d \mod N = m^{ed} \mod N$$

In other words, Alice decrypt the message when she signs it. The solution to this problem is to use a different key pair for signing and exchanging messages. In this project, we do not have any message exchange. Therefore, the blinding attack does not concern us.

Oblivious transfer The oblivious transfer [2] is a protocol which permits to exchange data between a client and a server. Let's imagine that the server has n tuples of the form (w_i, c_i), w_i is an index and c_i the corresponding data. The particularity of this protocol is that the client can retrieve the data c_j from the index w_j without revealing the value w_j and without learning the value c_i with $i \neq j$
To achieve this, let's first consider a hash function H and a random generator G. Let's also assume that the server and the client have an RSA key pair. The server starts by creating n keys using the indexes w_i.

$$K_i = (H(w_i))^d \mod N$$

Then it uses these keys to encrypt the data in the following way :

$$E_i = G(w_i \| K_i \| i) \oplus (0^l \| c_i)$$

Then, the server sends every E_i to the client. Now, let's consider that the client wants to retrieve data for index w_j. Using blind signature, the client is able to reconstruct the key K_j corresponding to w_j. So, using a blind factor r, the client generates :

$$Y = r^e H(w_j) \mod N$$

Then, the client asks the server for a blind signature. This way, the server cannot learn the value w_j choosen. At the end, the client received the value :

$$K_j = H(w_j)^d \mod N$$

Finally, he can find the corresponding data using :

$$(a_i \| b_i) = E_i \oplus G(w_j \| K_j \| i)$$

If $a_i = 0^l$, b_i corresponds to the data wanted by the client. Since, the client has only one K_j, he can only decrypt the data c_j corresponding to w_j. So, he cannot learn any other value of c_i with $i \neq j$.

Chapter 2

Implementation

2.1 Cryptography library

Symmetric algorithms The first type of algorithms implemented is contained in the SymmetricEncryption.java class. It implements functions to encrypt, decrypt or sign a given message using a symmetric key. Since it already exists a java library [3] containing many algorithms, I start to investigate in it. Regarding encryption, my major concern for this project is to implement the protocol in the most secured way. Since AES256 is considered as one of the most secured symmetric algorithm, I naturally implemented it. Unfortunetly, security is not the only important feature in this project. Performance is also really important. In this case, AES256 is not efficient since it requires many calculations. Therefore, I considered another algorithm: one time padding encryption. It is very fast since it uses one xor operation. It is also secured enough since we need to encrypt one message and not to create a secured channel. The only important requirement is to generate a different key for each encryption. Furthermore, the implementation of two algorithms will permit to compare their performance. Regarding signature, the implemented algorithm is HMacSHA256. It generates MAC efficiently and in a very secured way.

Asymmetric algorithms The second type of algorithms implemented is contained in AsymmetricEncryption.java class. This class only implements RSA. The choice of this algorithm comes from the fact that blind signature can be based on it. So, the class contains functions to encrypt, decrypt, sign and verify the signature of a message. It also implements functions for blind signature. It means generating a random blind factor, blind and unblind a message. Like, SymmetricEncryption.java, there exists libraries [3] to implement theses functions. Nevertheless, I prefer to get rid of them to make my own implementation. It makes the code more readable and it permits to understand in details how blind signature is implemented.

Key generation Encryption can be relatively easy to implement, but key generation is more sophisticated. Therefore, I decided to generate keys using the existing java libraries [3] in MyKeyGenerator.java class. So, the generation is secured and efficient. Concretly it means that AES, RSA and HMacSHA256 keys are generated using theses libraries. One time padding key is generated using a secure random generator. My personal contribution is to offer different ways to generate a key. Symmetric algorithm key can be generated from nothing, from a given password or in a file and RSA key can be generated from nothing or in a file.

Hash Like SymmetricEncryption.java, the class Hash.java was implemented in the most secured way. So I investigate [4] and first implement hashing to store a password using PBKDF2WithHmacSHA1 algorithm. But, again, performance is also important. Furthermore, for this implementation, hashing is used to shorten data. So I implemented hashing using SHA256 which is faster. My first implementation is kept to test performance at the end of the project. These implementations are done using existing libraries. [3]

Oblivious transfer Since oblivious tranfer is used by a server and a client. I did not choose to create a static library. I created two objects OTSender and OTReceiver which can be instantiated by a server and a client to execute the oblivious transfer protocol. OTSender needs a Map representing the data and an RSAPrivateKey to be instantiated. It automatically generates the keys K_i. Then it provides two functions. One to encrypt the data using K_i by generating E_i and one to generate K_j from the index Y received from the client. OTReceiver needs an index w_j and an RSAPublicKey to be instantiated. Then it provides four additional functions. The first one blinds the index using a given blind factor r to generate Y. The second one unblind the value K_j received by the server. The third one is used to decrypt the data E_i received by server using K_j. And finally, the last one checks the data generated during the decryption to find and return the corresponding value.

2.2 Database

Database connection After implementing the necessary cryptographic components, the next step was to find an efficient database for the storage. This database must have many properties. It must be free, fast and it must offer a performance report. I chose to use the SQL database of the Microsoft Azure plateform [5]. It is free for academic purposes [6]. It is fast because their servers are available in Europe. Furthermore, their platform automatically provides many statistics about the database. The only disadvantage is the storage size since there is only 35 MB available. In the concern

of this project, it is enough, since any user store only a small amount of data.

The implementation has two parts: SQL and java. The SQL part means tables creation and testing queries. I created two tables. Storage_server_optimal for the server optimal protocol and Storage_storage_optimal for storage optimal and privacy optimal protocols. Storage_server_optimal has three columns (id, bsk and ctext), each one has type varbinary(256) to store 2048 bits. Storage_storage_optimal has two columns (id, ctext), each one is also of type varbinary(256). My SQL code can be found in the SQLCommands.txt file.
The java implementation is done using the JDBC library [7] created by Oracle and modified by Microsoft for their database. I provide database connection with an object written in the DatabaseConnector.java class. When the object is instantiated, the user provides which protocol he uses and then the corresponding connection is automatically created with the database. Then, I provide functions to execute SQL queries. It means inserting, searching and deleting a row. I want these functions to be the simplest for the user. So, there is only one sfor every protocols. For privacy optimal protocol, I also provide a function to fetch every elements of the table in a random order.

Chapter 3

References

[1] Kelum Senanayake. Blind Signature Scheme [Internet]. Available from: https://fr.slideshare.net/kelumkps/blind-signature-scheme

[2] Ogata W, Kurosawa K. Oblivious Keyword Search [Internet]. 2002 Report No.: 182. Available from: http://eprint.iacr.org/2002/182

[3] javax.crypto (Java Platform SE 7) [Internet]. Available from: https://docs.oracle.com/javase/7/docs/api/javax/crypto/package-summary.html

java.security (Java Platform SE 7) [Internet]. Available from: https://docs.oracle.com/javase/7/docs/api/java/security/package-summary.html

[4] Generate Secure Password Hash: MD5, SHA, PBKDF2, BCrypt Examples [Internet]. HowToDoInJava. 2013. Available from: https://howtodoinjava.com/security/how-to-generate-secure-password-hash-md5-sha-pbkdf2-bcrypt-examples/

[5] SQL Database – Cloud Database as a Service | Microsoft Azure [Internet]. Available from: https://azure.microsoft.com/en-us/services/sql-database/

[6] Azure in Education | Microsoft Azure [Internet]. Available from: https://azure.microsoft.com/en-us/community/education/

[7] mssql-jdbc: The Microsoft JDBC Driver for SQL Server is a Type 4 JDBC driver that provides database connectivity with SQL Server through the standard JDBC application program interfaces (APIs) [Internet]. Microsoft; 2017. Available from: https://github.com/Microsoft/mssql-jdbc