Applying Homomorphic Encryption in the Cloud

Jess Antonio Soto Velzquez

August 31, 2015

Summary

This work proposes an implementation of a client-server architecture based software that employs *HElib*, an homomorphic encryption library, to perform computations on encrypted data.

Cloud computing is a cost-efficient alternative to host data remotely and subsequently perform computations on it. Confidentiality becomes an issue when data is delegated to the cloud, as often the information hosted is considered to be sensitive. To overcome this, a cryptographic algorithm may be used to encrypt the data with a secret key, allowing access only to the trusted parties. The downside is that whenever some modification or calculation needs to be made on the encrypted data, it must be downloaded and decrypted before working with it. Consequently, the data would be re-encrypted and reuploaded after it has been modified. An alternative to this approach is to make use of *homomorphic encryption*, an advanced technique in cryptography that enables computations on encrypted data without needing the secret key. HElib is a C++ library that implements the Brakerski-Gentry-Vaikuntanathan (BGV) homomorphic encryption scheme, and it is considered to be as a set of building blocks for any application that uses homomorphic encryption.

To tackle a real-life application of homomorphic encryption in the cloud, consider a scenario where a household has an expected pattern of activity, i.e. empty during the day and non-empty at night, so that the resident seeks to ascertain the number of people inside at any point in time during the day. Assuming the resident has put in place certain sensors around the building that detect who comes in and out, he would like to remotely learn the value of a counter that keeps track of the sensor data. As the resident chooses to store the value in the cloud, he quickly realizes he does not want others to learn of this value, not even the cloud service itself, as to prevent potential burglars to break in when the household is empty. Additionally, he wishes that the counter gets updated at certain intervals, e.g. every 20 minutes.

In contrast with other works, the proposed solution tackles the problem by employing homomorphic encryption. It is a simple client-server architecture which could be expanded to adapt with other scenarios and requirements. The proof of concept provides the means to operate on the encrypted data, i.e.

performing additions on the counter value, without compromising confidentiality and wasting resources caused by the overhead costs of re-encrypting data at every change.

The proof of concept was implemented in C++, relying on HElib and a library that to establish communication between the server and client. Following the client-server architecture, certain functionalities are delegated to the client, while others are performed by the server. In general, the tasks that either the server or client performs are briefly described:

1. Parameters to use with the HElib are arbitrarily chosen by the client, while other required values are computed.
2. Public and private keys are generated by the client.
3. Public key is serialized into a file and sent to the server via TCP sockets.
4. The initial counter value is set into a plaintext data structure.
5. The initial counter value is encrypted using the public key and stored into a ciphertext data structure.
6. The encrypted counter value is sent to the server to be stored, until a change in the value is requested.
7. Values are arbitrarily added or substracted from the ciphertext depending on the activity of the household.
8. Ciphertext is decrypted using the private key, and its value is stored in another plaintext structure.
9. Newly decrypted plaintext is printed to verify correct result from operations.

To show the feasibility of using HElib to do homomorphic encryption in a client-server environment, experiments were designed by varying the value of a security parameter k. This value has a direct impact on the execution time of key generation, encryption, and decryption, as well as the size of the resulting public key. The results are shown in the following table:

Table 1: Experimental results

k	Key Gen.	Key Size	Encryption	Decryption	Addition	Ctext Size
40	13.130 s	165.922 MB	0.355 s	0.048 s	0.074 s	30.977 kB
60	10.710 s	91.635 MB	0.414 s	0.056 s	0.077 s	36.053 kB
80	8.380 s	57.080 MB	0.642 s	0.057 s	0.082 s	37.745 kB
100	9.529 s	35.453 MB	0.579 s	0.064 s	0.082 s	43.676 kB

The findings show that even though key generation time is long, taking approximately between 9 and 13 seconds, and key size is large, being between

35 megabytes and 165 megabytes, it is not considered to be an issue, as this step is done once. On the other hand, encryption, decryption, and addition times are amazingly fast, none of them exceeding a second. However, the ciphertext raises some doubts over the magnitude of its size: during development, it was found that on average, ciphertext size was 74 megabytes; meanwhile, during experimentation, it was found that ciphertext size rose no higher than 45kB.

At a first glance, the results obtained from the experimentation were satisfactory; however, some doubts were raised upon noticing that there was a great variation between the development and experimentation phases of the ciphertext size. More experimentation is recommended to fully ascertain what other factors have an impact in the size of the ciphertext.