UNIVERSITY OF TWENTE.

SECURE DATA MANAGEMENT

Name	University	E-mail address	TUD #	UT #
Ali Al-Kaswan	TUD	a.al-kaswan@student.tudelft.nl	4679768	s2659158
Yana Angelova	TUD	y.y.angelova@student.tudelft.nl	4649370	s1910205
Dex Bleeker	UT	dex@bleeker.nl	5432014	s1460366
Célio Porsius Martins	TUD	c3lio.martins@gmail.com	4711955	s2487586

November 15th 2021

Abstract

Searchable encryption keeps proving to be a challenging field. This paper presents a proof of concept solution to some system in which users can encrypt data, share it with their consultant and both user and consultant are able to search for keywords through the encrypted data. It is important to note that the consultant can decrypt the data of all users, while users can only decrypt the data they have encrypted. The same goes for searching through the data. Additionally the system allows for the consultant to encrypt and save data for their users. In essence the implemented system is a many-to-many searchable encryption scheme.

1 Introduction

The need for better security of individual's data is a key priority in the recent years. To aid in this matter, more and more searchable encryption is used in practice. In this report the development of such a scheme will be discussed. The domain of the project is searchable encryption for financial consultancy firm. In this set up the main users are on one side the clients of the firm, who can upload and query their own data, the consultants, who can query and upload data to their own client's accounts and on the other side the server used to store the data. The main security assumption is that the server is acting in a semi-honest way (honest but curious). This report will first lay out the presumed data model in section 2 and discuss the main design choices and architecture in section 3. The main limitation of the system are mentioned in section 4. Finally the discussion of the project will be done in section 5

2 System overview

This section will first outline a use case scenario for the system. Afterwards the system's architecture and data model will be presented.

2.1 Use case

The use case of the system includes 3 distinct parties, namely, the server, the consultant as well as a group of clients. The server is responsible for the storage of all the files, the indexes, the cryptographic parameters as well as the public variables used by the other parties to create the keys. The server is also responsible for the evaluation of the trapdoor functions. The consultant can upload data associated with a certain client and read all data uploaded to the server. Any client can also upload data, which can only be read by him (as well as the consultant). Furthermore the client and consultant can upload a searchable index associated with every file they upload, this index will contain keyword associated with that file. The index should only be searchable by the parties who have access to the file.

To use this application, the server must first be started. The server will initialise a prime and a generator. The consultant must then connect to the server, get the prime and generator, generate his own public and private key and send back the public key to the server. The server will then add his public key to the Users map with ID 0. The clients can then also enrol into the server similarly, but they would get different IDs.

When a client then wants to upload a document to the server, they will simply encrypt the document with their own public key, as well as the public key of the consultant. Furthermore, they will generate an

mPECK of the keywords, encrypted with their own public key as well as the public key of the consultant.

When a consultant wants to upload a document to the server, they will first encrypt the document with their own public key and the public key of the client associated with that document and then create an mPECK of the keywords, encrypted with their own and the client's public key.

To search the files either the consultant or a client creates a trapdoor function, this trapdoor function contains the keywords that they want to search and is encrypted using their private key. This trapdoor will then be evaluated by the server, which will then return all the files that contain those keywords that are accessible by the maker of the trapdoor. Note that the trapdoor does not reveal the keywords to the server. In the case that the user has no access rights to the documents they are trying to retrieve, the server will return an empty list.

2.2 Data model

The data model is shown in Figure 1. The User is the supertype of Consultant and Client, the User stores an ID for identification as well as a private and public key. The Consultant has a fixed ID of 0, and no user is allowed to have their id as 0 because of that. Both the Client and Consultant can create a File, every file has one Client and one Consultant which can decrypt the file, every file has its own ID as well as the encrypted data bytes. There is no limitation to the types of data that can be encrypted, the file is split up into bytes and every byte is encrypted one-by-one. Every File has its own mPECK object, this mPECK object is also associated with a single Consultant and a single Client. Every mPECK is also associated with one ore more keywords from the keyword set. Each User can also generates a trapdoor function, with which they query the server for files containing specific keywords. Finally, there is a single Server which stores the prime and the generator which are used by the Users to generate the keys, as well as a list of all users and their public keys. The server stores all the files as well as the mPECK's associated with them, and evaluates the trapdoor functions sent to it by the users.

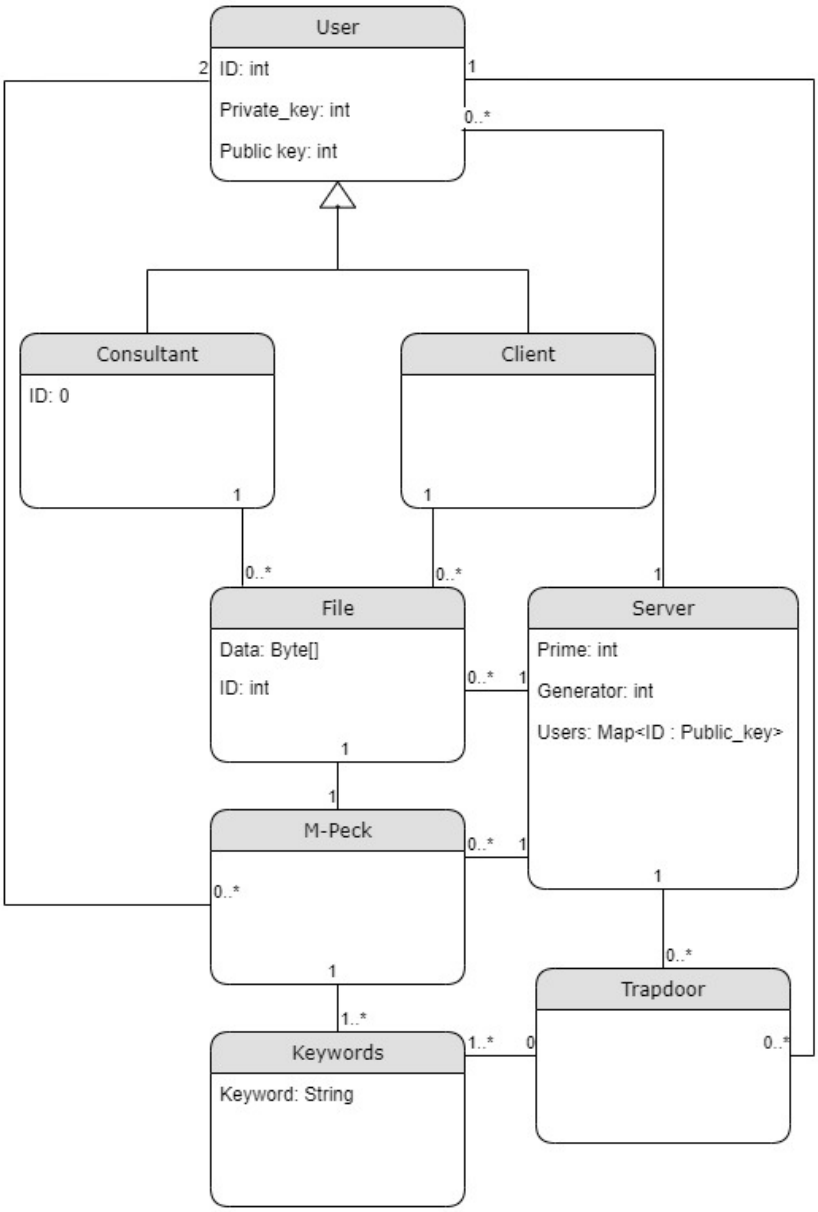

Figure 1: Data model definition diagram

3 Design choices

This section presents the main design choices of the system, with the argumentation behind them.

3.1 Implementation details

Language Our programming language was decided quickly, as all team members have previous affinity and/or experience with the Python programming language. Furthermore, Pyhton has support for multiple cryptographic libraries, which will come in handy for this system's implementation.

Tests The tests were written with the help of the excellent (and aptly named) unit testing framework - unittest[1]. They provide a way to validate the expected behaviour of the system.

Encryption The ElGamal Multi-Receiver Encryption has been used. The encryption scheme has been implemented as outlined by [2]. This scheme was chosen because of the simplicity of the ElGamal encryption and it's security under the Diffie-Hellman assumption.

Pairing functionality The pairing function for the trapdoor generation and evaluation has been implemented using the PBC (Pairing-Based Cryptography) library [2]. To be able to use it in Python we have made use of the following PYPBC extension [3]. This library was used in order to make the pairing step easier, as it does not require additional implementation of a pairing function.

Hashing algorithm We have been using SHA3 with block size 256 and 512 for use in the trapdoor functions. Since this functionality requires two hash functions we have decided to use the same function with different key lengths. SHA3 is proven to be a secure and fast hash function for both block sizes, rendering it a good fit for this use case [3].

3.2 Multiple key sets

Unfortunately, we were unable to use the same keys (from the PBC library) in both the encryption/decryption of the data and the trapdoor generation/evaluation. To get a working proof of concept in the end, we decided to 'just' use two key sets for each user. One pair of keys is used for the encryption/decryption of the data. These keys are generated using the ElGamal key generation function using the prime provided by the server. The second pair of keys is used for the generation of the mPECK and trapdoor. These keys are derived from the PBC library again using the required parameters provided by the server.

4 Limitations

In this section the main limitation of the system will be discussed as well as reasons will be provided as to why they are still present.

4.1 No data protection

The server assumes the provided user_id is the actual ID of the user that is requesting (or storing) some data. It is currently not enforced, so this is a limitation of the system. Ideally, the user should not have to provide its user_id and the server should be able to deduct it from whoever is calling the function. We did not find a way to do this (reliably) and have therefore opted for this circumvention. This imposes the assumption that the users are acting in a honest way.

[1] https://docs.python.org/3/library/unittest.html
[2] https://crypto.stanford.edu/pbc/download.html
[3] https://github.com/debatem1/pypbc

4.2 File encryption/decryption is slow

With the current implementation (splitting the file into bytes and encryption/decryption the bytes individually), encryption and decryption of files is slow. The test that executes this code is skipped in normal tests runs for this reason. This could be circumvented by the use of a different encryption scheme, which in it's turn will introduce different limitations to the system. This is why we have opted for the current set up.

4.3 The use of two key sets

From an architectural perspective, the use of two key sets introduces more complexity than needed. The system should be able to function (both encryption/decryption as well as trapdoor/search) with a single key pair (for each user). Since our team was not able the find a way to make this work, due to the limitations of the PBC library, which is not intended for encryption/decryption us, the current set up relies on both key pairs.

4.4 Inefficient data storage

The data that is submitted by the users and subsequently stored on the server, is stored twice. This is an implementation detail to let the consultant (easily) search for and decrypt all stored data, but this should not be needed. Ideally, the data storage should be refactored to store the data only once and the consultant should still be able to search trough the entire storage. This refactoring is however, outside of the scope of this project as the main focus is the searchable encryption functionality.

4.5 Corresponding user for returned data

The consultant can successfully search for and decrypt the users data, but it currently has no way of knowing what user the data belongs to. This is a huge drawback but not hard to fix. The data storage could be expanded to have the user ID included, so that the consultant can use it to derive which user the data is from. However, the same point as raised in subsection 4.1 applies to this limitation as well.

4.6 Access pattern is leaked

A common limitation in searchable encryption schemes is the fact that the access patter is leaked to the server. This system is also vulnerable to it as the server can link a user's query to the documents they are trying to access. To mitigate this limitation we would have to trade off the efficiency of the system.

4.7 Replaying of the trapdoor

A malicious user could intercept a trapdoor from the consultant and reuse the trapdoor. This would return a certain amount of files, then the malicious user could insert new documents with different key words. Using the trapdoor the malicious user can now check if the consultant was looking for those keywords by using the same trapdoor again. If more files were returned than the first time, the malicious user knows that at least one of the keywords of his new document were being searched by the consultant. If the same amount of files get returned the user can try again with new key words as there is no limit to the amount of files that a user can upload.

5 Discussion

In the entire system we are currently assuming that the server is honest but curious, but if the server is actually malicious he could give everyone his own public key instead of the consultants and he would then be able to read everything stored on the server. In practice this could easily happen and it would be hard to know for anyone on the outside as the server could still send proper seeming files back to the user and the consultant. This vulnerability can be mitigated by introducing a trusted third party which will be responsible for the key generation procedure, such that the server has no power over it.

Another point of discussion is that currently all encryption is done using the El Gamal encryption scheme, which is very slow compared to symmetric key encryption. This would seriously impact the scalability of the system. If the system were to contain many large files and many users, the computation time required to encrypt and decrypt would make searching very slow. A solution would be to use symmetric key cryptography, which is significantly faster than public key schemes. However, that would require a more robust key distribution algorithm. This is a trade-off between search and encryption speed, as well as the ease of key distribution.

References

[1] Yong Ho Hwang and Pil Joong Lee. "Public key encryption with conjunctive keyword search and its extension to a multi-user system". In: *International conference on pairing-based cryptography*. Springer. 2007, pp. 2–22.

[2] Mervat Mikhail, Yasmine Abouelseoud, and Galal Elkobrosy. "Extension and application of El-Gamal encryption scheme". In: *2014 World Congress on Computer Applications and Information Systems (WCCAIS)*. IEEE. 2014, pp. 1–6.

[3] Morris J Dworkin. "SHA-3 standard: Permutation-based hash and extendable-output functions". In: (2015).