Advanced Topics in Online Privacy and Cybersecurity - 67515

Path ORAM implementation – Inbal Mishal

This implementation is based on the path-ORAM article

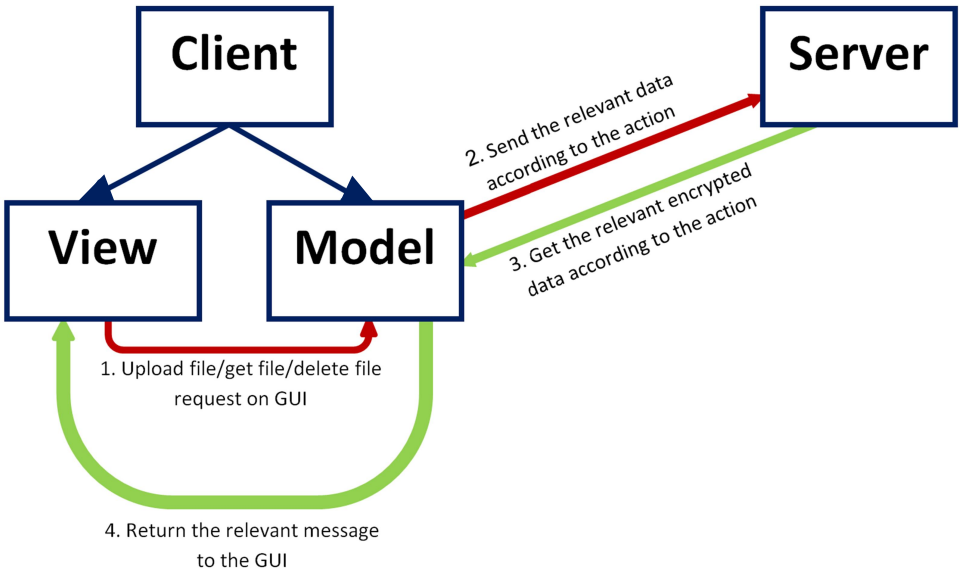
Description of the solution:

1. **API:**
   In order to run the project, you use two files – server.py and client_ui.py. Each one of them represents other side of the system. After you run the server side you can run the client side and start to use the application with the GUI.

2. **Libraries:**
   - **socket** – used to create the connection between the clients and the server.
   - **select** – allows the server get messages from multiple clients.
   - **json** – used to represent lists and dictionaries with strings.
   - **cryptography.fernet** – used to encrypt and decrypt the data.
   - **os** – Used to save the key on the client computer.
   - **string** – used to generate dummy files and padding.
   - **random** – used to generate dummy files.
   - **numpy.random** – used to generate the padding bytes.
   - **math** – used to calculate log and power.
   - **hashlib** – used to get authentication by saving and checking the hashed data.
   - **tkinter** – used to create the GUI.

3. **Classes & methods:** In the code.

4. **Files and Architecture:**



- **view\client_ui.py –** The file that starts the program running and responsible to the user interface. It starts with zero_page function that initializes the number of the files and the size of the tree in the server size. Then, according to the action the user wants to do he gets a different page – upload_file page, get_file page and delete_file page. In addition, it is responsible to show exceptions to the user.

- **model\bucket.py –** Class that represents the bucket that saved in the server side. When we upload the data to the server, we encrypt object of type Bucket that include a list of ServerFile objects, and the number of real files (not dummy). Each encrypted bucket represents a node in the server tree.

- **model\clientfile.py –** Class that represents the data that saved in the client side for each file. The data is the filepath, the leaf number, the real size of the file before the padding, the hashed_file and the padded file name.
  When we upload file to the server, we save on the client side this object in a dictionary according to the file name.

- **model\serverfile.py –** Class that represents the data that saved in the server side for each file. When we upload the data to the server, we encrypt object of type Bucket that include a list of ServerFile objects. Each one of them includes the file name and the data.

- **model\client.py –** This file includes the code of the client side. The Client class includes the upload, get and delete function. This code connects to the server using python sockets.

- **model\server.py –** This file includes the code of the server side. The Server class includes the upload, get and delete function. This code connects to the clients and serves them according to their requests with python sockets.

- **model\constants.py –** Includes the server ip, the server port, the file size, the message size (how many bytes can be transferred in one message), the filename size, the dummy size and the constant that determines the bucket size according to log(num_of_files).

5. **Security considerations:**
   - We use end to end encryption using Fernet package (cryptography.fernet).
   - We use Path-ORAM protocol to hide sensitive information about the unencrypted data that can be inferred using statistics.
   - Authentication – we use hashing to validate our data using hashlib package and sha1 hash.

6. Performance benchmarks:

Throughput (number of requests/sec)	Latency (time to complete a request)	DB size	number of requests
2.06	0.48	3	30
1.83	0.56	3	60
1.65	0.46	3	150
2.29	0.43	7	15
2.18	0.49	7	60
2.11	0.47	7	210
2.25	0.45	15	15
2.21	0.43	15	60
2.17	0.49	15	150
2.15	0.52	31	30
2.14	0.46	31	60
2.13	0.47	31	150
1.75	0.63	63	15
2.02	0.45	63	30
1.02	0.98	63	210
2.31	0.45	127	15
1.75	0.64	127	60
1.12	0.63	127	210
2.01	0.48	255	15
0.76	0.65	255	60
1.64	0.64	255	150

- **Benefit from multicore**

  In my implementation, I don't use multicore but the fact is that multicore may help the program run more efficiently.
  Any process that can be parallel (calculations, decryption, etc.) can use multicore for acceleration and optimization.

  We will not be able to use multicore in operations that require access to the database because we can't allow reading and writing to the database at the same time. This is impossible because then we have a critical section problem - writing to a file can impair the read operation.

  Instead, we can optimize each action individually by reading and sending each bucket separately with a separate thread instead of reading the entire path at once.

  In addition, when working with several clients, a separate thread can be used for each client, since each client has his own database.