PathORAM - Design Document

Lior Ashkenazi | 315533059

May 5, 2022

1 Preface

This design document intends to introduce the PathORAM project - a project compromised of two compontents: the $client$ and the $server$, which communicates with each other by TCP/IP protocol to transfer files.

2 Client

The $client$ module compromised by 4 sub-modules:

- cloud
- crypto
- storage
- ui

As well more 3 helper files - config.py, data.py, log.py: config.py has some global constants which the $client$ module use, data.py has functions whose purpose is to handle files and log.py configures the logging of the client program, i.e., the informative messages the client will be shown during activation of the program.

The $client$ module process of sending a file is as follows:

$$\begin{tikzcd}
ui \arrow[d] \\
storage \arrow[d] \\
crypto \arrow[d] \\
storage \arrow[d] \\
{}
\end{tikzcd}$$

$$\begin{tikzcd}
cloud \arrow[d] \\
\boxed{server}
\end{tikzcd}$$

and the process of receiving a file is as follows:

$$\begin{tikzcd}
ui \arrow[d] \\
storage \arrow[d] \\
cloud \arrow[d] \\
\boxed{server} \arrow[d] \\
cloud \arrow[d] \\
storage \arrow[d] \\
crypto \arrow[d] \\
storage
\end{tikzcd}$$

2.1 cloud

- __init__.py
- cloud.py
- exceptions.py
- server_connection.py
- utils.py

Responsible for the communication with server, getting the data from other sub-modules and sending it to the server.

2.2 crypto

- __init__.py
- aes_crypto.py
- exceptions.py
- key_map.py
- utils.py

Apply all the cryptography of the program. For encryption (and decryption), the sub-modules uses:

- AES (256-bit key length)
- CBC-mode with IV(128-bit)

And lastly padding with PKCS7.
Moreover, the sub-module uses HMAC (256-bit key length) with SHA256 for authentication. Furthermore, the sub-module uses a key-map for storing for each password (currently, the program can only have one password!) a map of the pertained keys (specifically - Salt, AES and MAC) for future encryptions and decryptions.

2.3 storage

- __init__.py
- controller.py
- data_file_map.py
- exceptions.py
- file_processor.py
- oram.py
- stash.py
- tree_map.py
- utils.py

The core of the *client* module and perhaps of the entire program. Composed by controller.py which oversees the entire module operation, stash.py which resembles stash and oram.py which is a PathORAM tree. Furthermore, the module has maps for the PathORAM tree, for tagging each data block to a leaf in the tree, and for data blocks - note that we tag each data block with a data ID, the sub-module doesn't work with files per say, it works with data files, which are chunks of the files ther user uploads. For createing such chunks the file_processor is used for. Also, for combining chunks and then for dismantling them, the data_file_map is necessary - for padding and upadding if necessary (almost always it **is** necessary) and for saving information about expected file size.

2.4 ui

- __init__.py
- handler.py
- utils.py

Connected to the main.py (will be described later), is the bridge to the user. Handles every call of the user for a chosen action.

2.5 miscellaneous

- config.py
- data.py
- log.py

Described above.

3 Server

- server.py
- utils.py

A module which represents a simple server which uses TCP/IP protocol. Recieves and sends files and stores them in a data folder in $blockX.oram$ format. Unaware of the data in those files.

4 Benchmarking

We first show the latency of the program - we uploaded a single txt file with size of 4 bytes where the $BLOCK\_SIZE$ configed to be 4096 bytes (4 KB), although results were very similar for block sizes ranging from 1 KB to 32 KB. Note that the data of the given file enters one data block, including padding and encyrption. These were the results:

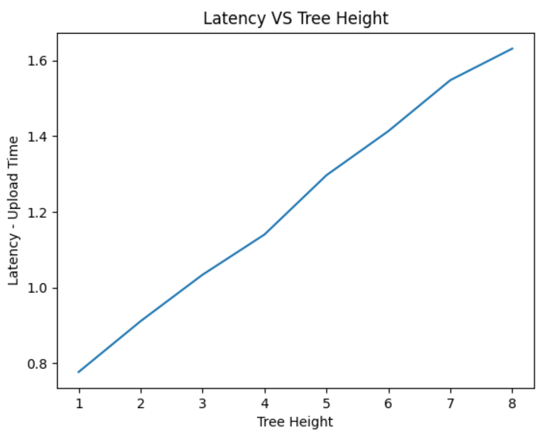

Figure 1: Latency

Where the x axis is the tree height so the number of nodes, n is 2^x,and where the y axis is the latency which is the upload time in seconds that took to the txt file to be uploaded to the server. Note that the graph is logarithmic in n which does make sense because every linear increase in the tree height results in linear increase in the length of path, when the tree is always full and balanced binary tree.
Therefore, the throughput, which measured by files that can be uploaded in a unit of 2 seconds, is the following graph:

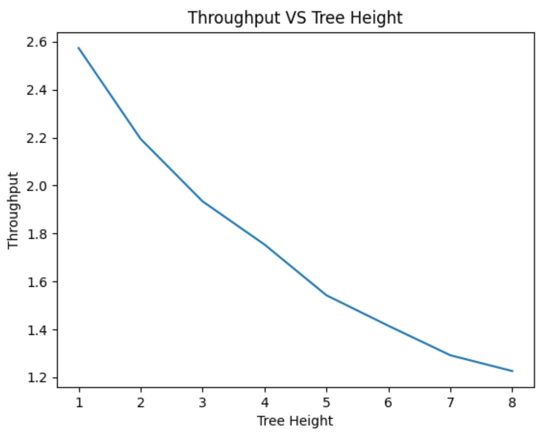

Figure 2: Throughput

Again, the graph is logarithmic in n. However, this time the graph is a decreasing graph which make sense - it takes less time to upload to the server any given data block (in our case, a file with padding and encryption) when for each file the program has to read less nodes from the server before uploading the given file, which comprises most of the latency time. Thus, the less nodes we have in a tree - the more files we can upload unto a time-unit.

Moreover, multi-threading was considered for this project. However, for multi-threading under file transportation, one has to make sure that both the client and the server support that kind of file transportation. Furthermore, in between the file transporation, there is a process called $write\_stash$ which doesn't include live transportation and therefore multi-threading was

considered to that process; however, parallelism is not well recommended for that process a not all processes are excatly the same - one has to result potentially in finding a desired file in the stash and to get it for further operation - thus the idea of implementing multi-threading in that process was unfortunately omitted.
As for multi-thread the file processing, before uploading a file to the server or after downloading a file from the server, in each occasion, the client has to operate synchronically because of the importance of packing and unpacking the data in orderly way (of course, with encryption and decryption following that) - the order of data is a must in order to restore the files correctly and not corrupt them during those processes, thus implementing parallelism by multi-threading is not recommended in that case also. Important to note however is that indeed one can implement multi-threading precariously in the the file transportation protocol by implementing skillfully a server and a client who support that kind of transportation; indeed one can even implement numerous servers for that purpose or even numerous clients.

5 Instructions

Before activating the program, please make sure you stand by the requirements of the program and have installd the following modules in your virtual environment:

- tqdm
- six
- cryptography

For using the program, firstly one has to open a terminal in the PathORAM folder and activate the server in the following way:

```
python3 server/server.py
```

If the server is online, you should see the line:

```
Server Waiting...
```

Now, for activating the client, one has to open a **new** terminal, **while the server is online**, and enter the following line:

```
python3 main.py
```

Thereafter, a new user has to enter a password (in the next activations, the client has a saved key_map file so the password is saved and has to be used in the next time). After entering a password, intructions for using the program are shown.

IMPORTANT NOTE For resetting the program, one has to open a terminal in the PathORAM folder and enter the following command, **only after terminating successfully both the client** (by entering the command: quit) **and the server** (by closing the terminal):

```
python3 reset.py
```

This will result deleting all the map files and **all the stored data**. Note that after activation of the reset program, one has to enter a new password in order to use the program. **DO NOT RUN THIS LINE WHILE THE SERVER IS STILL RUNNING. PLEASE ONLY DO SO AFTER CLOSING THE SERVER, SIMPLY BY CLOSING THE TERMINAL**. Not doing so will result in bugs.
Good Luck!