



# UNIVERSITY OF PISA

Foundations of Cybersecurity

*Year 2021/2022*

Secure Cloud Storage

*MIRCO RAMO*

*FRANCESCO DEL TURCO*

<b>1.0 Introduction</b>	<b>3</b>
<b>2.0 Components</b>	<b>4</b>
2.1 Actors	4
2.2 AES-128	5
2.3 HMAC	5
2.4 SimpleAuthority	6
<b>3.0 Message Structure</b>	<b>7</b>
<b>4.0 Protocols</b>	<b>9</b>
4.1 Key Exchange Protocol	9
4.2 Upload Protocol	12
4.3 Rename Protocol	14
4.4 Delete Protocol	15
4.5 Download Protocol	16
4.6 List Protocol	17
4.7 Logout Protocol	17
<b>5.0 Implementation</b>	<b>19</b>
5.1 Message Exchange	19
5.2 Client-side implementation	20
5.3 Server-side implementation	22
5.4 Protocol implementation	23
<b>6.0 User Guide</b>	<b>25</b>

Github repository: <https://github.com/Mirco-Ramo/Secure-Cloud-Storage>

## **1.0 Introduction**

The purpose of this project is to create a cloud storage able to provide confidentiality and integrity to its users, therefore providing the essential operations of any cloud storage but with a further guarantee on the privacy of sensible data and on the integrity of such data. Such achievements are obtained by exploiting modern cryptography algorithms and secure coding approaches, in order to have a good security level both in the application protocol and at implementation level. For this project, we have a set of pre-requisites to keep in consideration, in particular we have to assume each user to be already registered to the cloud storage, so a long-term RSA key-pair has already been negotiated; moreover, we consider the user to already be in possess of the certificate of the CA, that in turn released the certificate of the Cloud Storage Server, which will be used to check the authenticity of the cloud storage's own certificate; finally, we assume that the Cloud Storage Server already knows the username of each registered client along with their public RSA key, together with their allocated dedicated storage. The aim of the project is therefore to design a secure session keys exchange protocol and some operations to be performed with such keys, providing authentication and encryption during the session along with protection against replay attacks. In Chapter 2, we will show the components of our project; in Chapter 3, we will show the structure of the messages we will send at application level; in Chapter 4, we will go through the protocols, showing the sequence diagrams and explaining our main design choices; in Chapter 5, we will highlight some implementation details which show how we handled some possible vulnerabilities at code level; in Chapter 6, we will present a simple user guide to run the project.

## 2.0 Components

In this Chapter, we will give some insights on the main components of our project, highlighting the actors involved and exposing an overview of the operations required to be performed for the project. Moreover, we will highlight the main cryptographic algorithms and techniques we will exploit in our project.

### 2.1 Actors

The first actor we consider is the user: following the project's guidelines, the user is pre-registered to the cloud storage, meaning that the username and the public key are already stored in the latter. The user must authenticate when starting the application, so they have to send a value signed with their private key, as we will see inside the key exchange protocol: for this Perfect Forward Secrecy is required. After login, the user can perform an upload of a file, require the cloud storage the list of available files, download, rename or delete a file contained inside the storage or perform the safe logout operation. All these operations must be encrypted (either totally or partially) and authenticated to provide integrity and confidentiality; moreover, we need to be careful at application level to avoid the possibility for an attacker to perform replay attacks, for example forcing the download of a file from the cloud storage in order to occupy bandwidth.

The second actor is instead the Cloud Storage Server, which can be referred to as "server" for simplicity: the server is supposed to be running when a client application starts, so that it can receive the authentication request from the user; the server will negotiate the symmetric session keys with the user, which will then be used to encrypt and authenticate the whole session.

To communicate with each other, the user and the server will exploit a TCP socket, which ensures that the messages sent by one of the parties will not only arrive without modifications due to noise to the other party, but they will arrive in the order that they've been sent: with this assumption, we will be sure that if the message arrives modified to the receiver, it must be due to the presence of an adversary. In such case, it is a good engineeristic technique not only to discard the message, but to completely restart the session and actually also the connection: we consider that only a session can be established on a TCP socket, meaning that after performing the logout operation (or finding out that a message has been manipulated) it is not possible to open a session over the same socket, but we need to create a new socket to start a new session; this is considered a good engineeristic practice.

## 2.2 AES-128

AES is one of the most used encryption algorithms as of today: this algorithm became a standard in 2000 after a request for proposal published by NIST and it is still considered secure. In particular, AES is a block cipher which can exploit keys of different sizes for encryption: for our project, we decided to use AES-128, which uses a 128-bit key and a block size of 128, which is the size typically used for public applications. Differently from its predecessor DES, AES is able to encrypt an entire block at each round, reason why it is actually better than DES since it requires a lower number of rounds to encrypt the same number of blocks. The best known attacks for AES are still unfeasible as of today since they require either too many plaintext-ciphertext pairs or too much computational power, so as of today AES-128 is considered to be very secure, up to 128-bit security level. Moreover, AES is conceived for an efficient software implementation, since it exploits a particular mathematical setting which is the one of Galois Fields. It is important to highlight that we decided to exploit in particular AES-128 in CBC mode, which allows us to randomize the encryption operation exploiting an Initialization Vector which is XOR-ed to the plaintext before encryption in order to obtain different ciphertexts if we send multiple times the same message; this mode is also CPA-secure. According to the project's functional requirements, AES-128 in CBC mode was enough to accomplish the security aims, so we decided not to choose bigger keys, that would have slowed down the performance and the throughput.

## 2.3 HMAC

A Message Authentication Code can be considered as a particular hash function which exploits a key, which is still able to provide message authentication and integrity: the idea of MACs is to generate a digest, called in this case tag, by exploiting not only the input but also the secret key which is shared between the two parties involved in the communication. The general idea with the MAC approach is to compute the MAC on the sender, attach such MAC to the message and then compute the MAC also on the receiver, checking if the two values corresponds: if they don't, the message has been modified by a man-in-the-middle, otherwise we can consider the message to be intact. There are actually multiple ways of computing the MAC, which may exploit block ciphers or hash functions: for our project, we decided to use HMAC, which is a method which exploits a hash function, in our case SHA-256 (and SHA-512 only to hash DH keys), to build the MAC for a message, accepting the fact that it is quite complex since it uses multiple stages and computations to obtain the final MAC. Given how we compute the MAC, we need to consider the authenticated encryption scheme to use, which is the scheme that allows us to have both integrity and secrecy for our

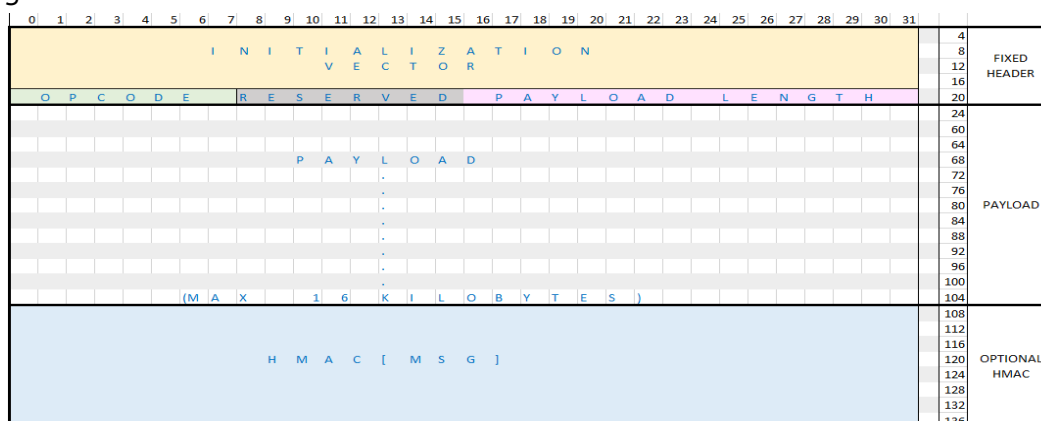
messages: in our project, we exploited the **Encrypt then MAC** scheme, in which we first encrypt the original message and then we compute the HMAC over the encrypted message, using **two different unrelated keys**; with this approach, if I transmit twice the same message I will obtain two different MACs since they're computed on the ciphertext, which is randomized; moreover, even if the MAC leaks some information, they will be information about the ciphertext, which won't be of any use of the attacker. For these reasons, EtM is considered to be a very good approach in any protocol. Since we exploit an encryption algorithm which is CPA-secure and an HMAC algorithm which is secure, it is possible to prove that the combination of the two is CPA-secure, which is the reason why we decided to exploit these algorithms.

## 2.4 SimpleAuthority

As an assumption for the project, we considered that the user had the certificate of the Certification Authority which released the certificate of the server, which must be used on user's side to authenticate the server itself: however, we had first of all to create such certificates, and to do so we exploited a tool called SimpleAuthority, which allows to create a fictitious Certification Authority for which we generated a self-signed certificate and a Certificate Revocation List. In the meantime, we also created a couple of public-private keys for the server exploiting the openssl commands: given such a public key, we used SimpleAuthority to generate a certificate for the server over that public key. In this way, the server was able to maintain on its side its own certificate along with its private key, while we assumed the user to have the certificate for the CA: during the authentication phase, the server is therefore supposed to send its certificate, which is verified by the user using the CA's certificate; if the verification succeeds, the user extracts the server's public key from the server's certificate and uses it to verify a signature over a message sent by the server itself, thus completing the authentication of the server.

## 3.0 Message Structure

In this Chapter, we will show the structure of the messages that are exchanged between the client and the server: this structure is standardized for all the messages, and it is the following:



In the picture, each column represents one bit, while on the right there's the count of the bytes for each row.

First of all, we can divide the message in a header and a payload. The header has fixed size equal to 20 bytes, containing three fields common to almost every message, which are the Initialization Vector, the opcode field and the field containing the length of the payload, which has variable size no greater than 16 kilobytes. The payload can then be followed by a HMAC, which is actually optional since it is not used for obvious reasons during the key exchange protocol: the size of the HMAC digest is equal to 256 bits, since it will be generated through SHA-256.

The IV field is composed of 16 bytes which contain the initialization vector to be used for encryption, in particular it will be given as input to AES-128 used in CBC mode. This value must be different for each encryption to obtain different ciphertexts mapping the same plaintext, while if the message does not contain any encrypted field we set the relative bits to 0.

The OPCODE field is an 8-bit string which contains an operative information which is sent in-the-clear from one party to the other: such opcode either specifies a command coming from the user to the server or a response to a command from the server to the client; however, we need to highlight that the command from the user can be sent in-the-clear even if it can be used for traffic analysis, since in any case an attacker could still perform traffic analysis by computing the bandwidth of the request and associating it to each different operation (for example, the bandwidth of the list operation is smaller than the one of a download, so even if we encrypt the opcode it will be clear to the attacker that we're performing a list and not a download operation); on the other side, the response of the server to a request from the user may be more meaningful, therefore we need to give an opcode which specifies that the

message contains a response to the request sent by the user, while the actual response will be contained in encrypted format inside the payload. In particular, let's consider that we have a download request, so the user sends a message containing the encrypted filename for the file they want to retrieve: if the file is not present in the cloud storage, the server will send a "missing file" response. If such response is in the clear, it might be intercepted by the attacker which can then understand that the user doesn't have the file in the cloud storage: this information may not be useful since the attacker does not know anything about such file, but it gives some hints, like the fact that it is not loaded in the cloud storage. On the contrary, by answering with a generic "download response" opcode, we can encrypt the actual response inside the payload, so that the user will be the only one knowing which the real response is. The values that the opcode can take are shown in the sequence diagrams in Chapter 4, above the relative message; possible encrypted responses for each message are reported on the sender-side of the schemes.

The third field contained inside the header is a field specifying the length of the payload, which can actually be variable in size and can span to up to 16KB: this was due to the implementation of the standard TCP receive operation, which by default creates a buffer of 64KB (the default parameter to create the buffer actually is equal to 32KB, but the buffer is doubled in size to accommodate additional overhead data) which is easily filled with 32KB messages, therefore we use 16KB messages to then use the default settings for TCP without worrying of filling it. In this case, the payload length field needs to be specified on 14 bits, but we leave 2 more bits on the payload length which will always be set to 0 in order to ease the construction of the message.

It is important to highlight that we also have 8 reserved bits before the payload length: these bits can be used in case we exploit a transport protocol which supports bigger messages, it is already possible in our implementation to exploit them, at this time they're just all set equal to 0 for all messages.

The HMAC field will contain the MAC of the message concatenated to the counter of the sender, which introduces freshness to the message and is generally considered a secure yet easy implementation. In fact, the usage of counters substitutes the nonces, which are theoretically very secure but complex to implement due to the need of generating random bits for each message and keeping a table containing all the nonces which have already been used in the past.

The content of the payload field will be variable in size and will depend on the type of request it is executed at that moment: in some cases, it will actually be empty, like in the case of the request of the execution of the list operation from the user, which does not require any encryption or additional parameter to be sent in the message besides the opcode to recognize the request.

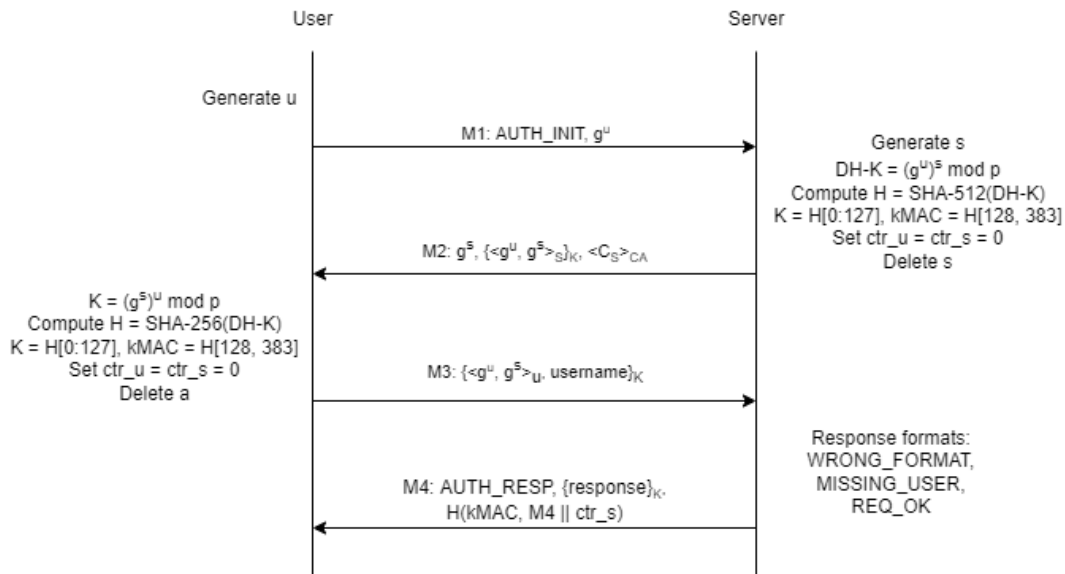


## 4.0 Protocols

In this Chapter, we will give an in-depth look at the protocols we've designed for our project, specifying their strength and the reasons that brought us to have such designs.

### 4.1 Key Exchange Protocol

The first protocol we will study is the key exchange protocol, for which we have the following sequence diagram:



The key exchange protocol allows to exchange a symmetric key which is computed on both sides of the protocol according to the Diffie-Hellman Station-to-Station approach, which is a protocol which allows the exchange of a symmetric key ensuring also Perfect Forward Secrecy. As in the case of standard DH, we have here that the protocol is based on the exploitation of the Discrete Logarithm Problem, which ensures that given the public parameter generated by the user and the one generated by the server ( $g^u$  and  $g^s$ ) it is computationally complex to compute “ $u$ ” and “ $s$ ”, which are though needed for the generation of the symmetric key. The terms “ $g$ ” and “ $p$ ” are considered to be publicly known and are encoded on 2,048 bits, which provide a security level between  $2^{80}$ , which is the one obtained with the 1,024 bits encoding, and  $2^{128}$ , which is provided by the 3,072 bits encoding: such security level is enough to consider the DLP difficult to break with modern technologies.

The first message in the protocol is intended to initialize the authentication from the user's side, which generates a private key “ $u$ ” and computes  $g^u$ , which is a value which can be sent in-the-clear since we consider difficult to find “ $u$ ” starting from this quantity due to the DLP;

the value  $g^u$  is sent in a message with opcode set to "AUTH\_INIT", to make sure that the server understands that a user is trying to establish a new connection. When the server receives M1, it generates its own private value for the DH-STS called "s", which is then used to compute the key DH-K with the Diffie-Hellman approach, where "p" is again to be considered publicly known. DH-K is the key we can generate on both sides thanks to the Diffie-Hellman approach, and in this case it stays on 2,048 bits since it is computed with modulo p. Since though we need to generate two secure keys, a typical approach we can take is to use this key as input to a hash function and take a portion of the output of the hash as the first key and another portion of the output of the hash to generate the second key: in our case, since we use AES-128 with CBC as encryption algorithm and HMAC-SHA-256 as MAC, we need two keys of size 128 and 256, therefore we need to compute the hash of the key using SHA-512 (SHA-256 would not give us enough bits in the output), then we take the first 128 bits as key for AES and the 256 bits from index 128 to 383 as key for the HMAC. It is important to highlight that this approach works thanks to the deterministic nature of hash functions, which allow us to compute the hash function of the DH-K on both sides and get the same output from which to obtain the key. Once the two keys are computed, we set two counters called "ctr\_s" and "ctr\_u", both equal to 0: these counters are actually used to prove freshness for the message we send from both sides, and are an alternative to nonces which are complex to implement. Using a counter and concatenating it to the message before computing its MAC, we can give freshness to the MAC itself, which indirectly gives freshness to the whole message; moreover, this approach allows not to send the nonce in-the-clear, which is therefore not known to the attacker; finally, even if we set the counter to 0 at the beginning of each connection, it won't be a problem since the result given by the HMAC will depend not only on the message and the counter but also on the key, so if the attacker is sure that they have the same message and the same counter contained in two spoofed messages, they will still see two different values for the MAC since the MAC key for the session will be different. The counter in this way is used to avoid replay attacks, so the receiver must concatenate the arriving message to the corresponding value of the counter of the sender and compute the MAC to check if it is equal to the MAC contained in the message which just arrived. After computing the two keys, the server deletes "s" since it is no longer needed and therefore must be deleted to avoid the risk of disclosing it by mistake, then the server sends the message M2.

In message M2, we have the same scheme of the classic DH-STS approach, where the server sends the value of  $g^s$  in-the-clear to be used by the user to compute the value of the Diffie-Hellman shared key. Along with  $g^s$  though, the server adds two other fields in M2, which are the certificate of the server signed by the certification authority and a field encrypted with

the key obtained from the output of SHA-512 (i.e., the session key) and signed with the private key of the server. As an assumption for the project, it is said that the user has the certificate of the certification authority which certified the server, so after receiving  $M_2$  the user will retrieve the public key of the certification authority to verify the digital signature over the certificate of B: if such signature matches, we have the insurance that the certificate we received was the certificate of the server, but we still don't have any insurance that the message  $M_2$  has been sent by the server, since an attacker could have just taken the value from a different message and used it in a manufactured message of their own. To make sure that the message is sent by the server, the user must decrypt the other field of  $M_2$ , so they must compute again the value of DH-K exploiting the DH paradigm, then use DH-K as input to SHA-512 to compute the key for AES-128 and the key for the HMAC-SHA-256; at this point, they can exploit the key for AES-128, called  $K$ , to decrypt the encrypted part of  $M_2$ , then use the public key provided in the certificate of the server to verify the digital signature of the content of the encrypted message; if the content that we verify matches  $g^u$ ,  $g^s$ , then the server is correctly authenticated, since we have the field signed with its private key. Notice that it won't be a problem to send the second field of  $M_2$  without encryption, since it contains a digital signature over elements which are publicly known, and the user would notice if an attacker replaces the digital signature with their own because the provided certificate will show that the owner of the signature is not who the user expects; however, encrypting this value allows Direct Authentication for the server to the user, meaning that the user knows that the server has the key, and also allows to make sure that the user computes correctly the AES key. After having computed the keys and authenticated the server, the user sets the counters to 0 as well and sends  $M_3$ .

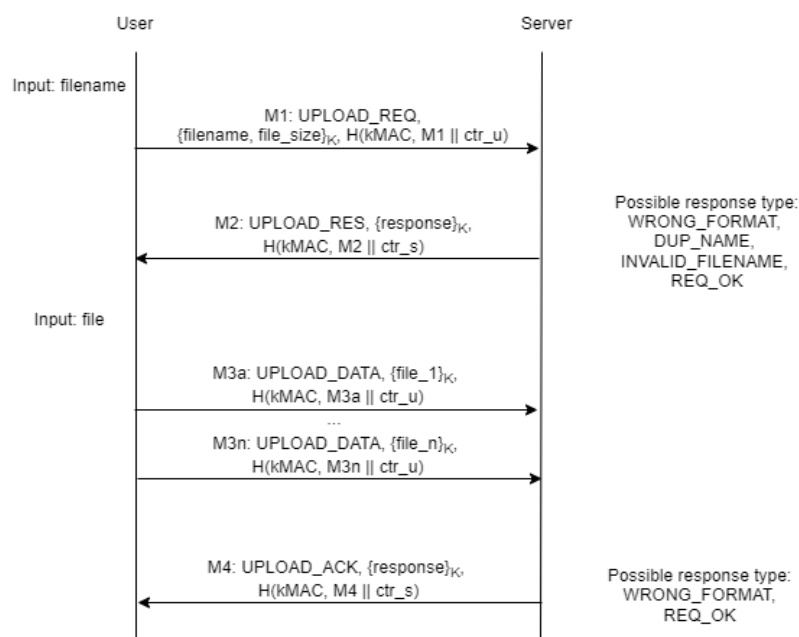
Message  $M_3$  is used for the authentication of the user towards the server: at this point in fact, the keys have already been exchanged, so we need to complete the authentication of the user by sending their username to the server. The username cannot be sent in-the-clear, since for privacy reasons we want to keep it secret, therefore it is included in the encrypted part of  $M_3$ , which also contains a portion signed by the user containing again  $g^u$  and  $g^s$ , similarly to the approach taken into DH-STs. The server decrypts  $M_3$  once received and takes the username sent by the user: since by assumption the users have already registered themselves, the server will have the username and the associated public key, so taken the username from  $M_3$  the server can retrieve the corresponding public key to verify the signed part of the message. Of course, this requires the username to stay outside of the signed portion, otherwise it would be impossible for the server to retrieve it.

At this point, once  $M_3$  has been controlled, there are three possible responses that the server can send to the user: if the message has a wrong format or the verification of the

signature was not successful, the server responds with a WRONG\_FORMAT encrypted response; if the user gave as input a wrong username, the server will reply with a MISSING\_USER encrypted response; finally, if everything went well, the server will send an affirmative response with REQ\_OK as encrypted response. There are two important things to highlight about M4: the first one is that the response is actually sent to the user encrypted and not as an in-the-clear opcode as in M1, since in this case the response may give us information we do not want to disclose, for example we don't want an attacker to see the difference between a MISSING\_USER response and a WRONG\_FORMAT one easily; the second thing to highlight is that the message integrity is guaranteed by exploiting the HMAC, computing it over the entire message (therefore both header and payload, to be sure also the header has not been modified) concatenated with the counter of the server. In this case, we need to compute the HMAC since it ensures the impossibility for the attacker to replay an old response, but it also proves the user that the server has the key for computing the MAC. After the arrival of M4, the user will know that the authentication process ended, with the outcome which depends on the response: if the outcome is positive, both user and server have authenticated themselves and have the session keys, so the user can start sending commands to the server; if something went wrong in any phase of the protocol, we need to restart everything, making sure that sensible data like "u" and "s" are deleted.

## 4.2 Upload Protocol

The first protocol we propose about the operations that the user can perform on the platform is the upload protocol, which can be represented by the following sequence diagram:



The protocol exploits counters as we explained previously in Chapter 3 and is designed such that the user will be able to send a file of up to 4GB of size. The user starts the protocol by sending message M1 to the server, which will contain the size of the file which needs to be uploaded and the name of the file with which the server needs to save such file on the user's dedicated storage, both encrypted: encryption is mandatory in this case since we don't want an adversary to know the size of the file we're moving or the name of the file, which can give away information about the content of the file. In this case, the opcode contains the type of the request, which is sent in-the-clear since an attacker can easily infer the type of the operation by measuring the traffic exchanged between the user and the server and the direction of the messages, so it is not important to encrypt this information. The message contains also the HMAC of, again, the entire message concatenated with the counter of the user: this allows to avoid replay attacks, since if the attacker replays a message it will get an error on the counter for the user, and will also ensure the integrity of the message, so the server will be sure that the message is received as it was sent by the sender. Notice that the file size is important for the server since it allows it to understand how many messages to expect in the future.

After receiving M1, the server checks if the filename is valid or not: the check on the name of the file is performed on the user side as well, but we need to consider on server side that everything the user sends us may be tainted, therefore we perform canonicalization of the input on server side as well. If there's already a file with the same name in the storage, an error message is sent to the user specifying that there's already a file with such name. Once the server has performed these two checks, it will send a response message where the response is encrypted and the opcode only contains a placeholder which still allows the user to understand that the received message is the response to the upload request: in particular, we encrypt the response once again since the different responses could give information to the attacker which we do not want to give away. The presence of the counter in this case allows to avoid replay attacks which could deny the service for the user: for example, the attacker could replay a negative response over and over, without the user noticing it, denying them the service.

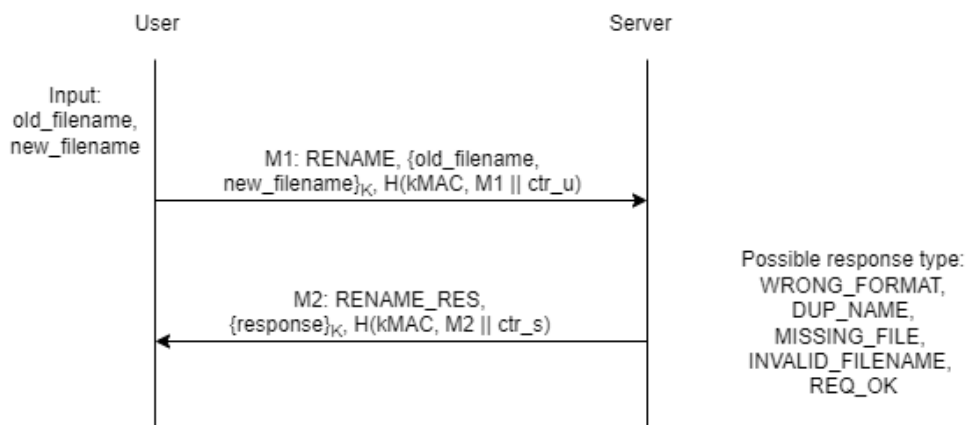
When the user receives M2, they check if the response is positive, in which case they send the file in multiple messages where the payload can have maximum size of 16KB: this is done in order to avoid loading in memory the whole file at once, which may saturate the memory if the size of the file is too big. However to trade-off memory consumption and computational time, we fetch and encrypt up to 16MBs of file at each iteration, sending each of those chunks in messages of maximum 16Kb each. For each message M3i we send it specifying the UPLOAD\_DATA opcode to show the server that the message contains the

data for the file: this opcode can be sent in the clear for the same reason the opcode in M1 is sent in the clear, it is actually a value that an attacker may not retrieve if encrypted, but whose meaning is still easily understandable by traffic analysis, therefore it is pointless to encrypt it. Each message is equipped with its MAC that includes the counter to guarantee integrity and freshness. Each time a message M3i arrives to the server, its integrity is checked along with the fact that the value of the counter is correct; if everything is fine, the receiver saves it in memory until it receives the whole encrypted chunk: in this case it decrypts it, writes the chunk in the destination file and starts again waiting for additional chunks, if any.

Once all the chunks have arrived at the server, a message is sent towards the user with the UPLOAD\_ACK opcode, which does not mean that the transfer has been successful but only that the transfer is concluded. In fact, we don't want the attacker to know if the transfer was successful, therefore the file is in the cloud storage, or not, which means that the file is not in the storage: to do this, we encrypt once again the actual response of the server, which again is followed by the MAC for the same reasons seen for message M2.

### 4.3 Rename Protocol

The rename protocol is a simpler version of the upload protocol and is represented by the following sequence diagram:

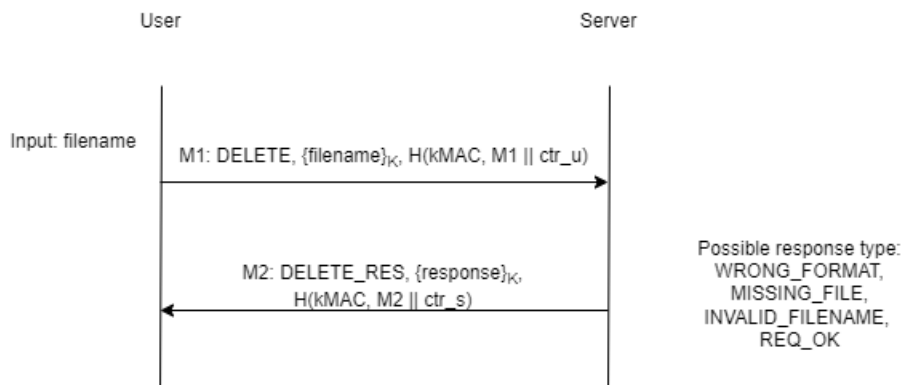


As we can see from the picture, the first message has a similar format to the one of the upload operation, but contains both the old filename of the file whose name we want to change and the new filename to use, both encrypted to maintain the privacy of such information. The message contains also the MAC of the message with its counter to ensure integrity and avoid old rename messages to be replayed by the attacker, which may change the name of the files.

M2 contains instead the encrypted response to the rename operation requested by the user, using a generic `RENAME_RES` opcode which does not give any information about the outcome of the operation to a possible attacker. In this case, the attacker won't even be able to know if the operation was successful or not: in the upload protocol, the attacker would know that the response of the server in M2 was positive or negative by checking if the protocol was going on or not; in this case, since there are no other messages after M2, an attacker cannot know if the operation succeeded or not. In case of error, the server can send multiple error responses, with `WRONG_FORMAT` being the response for a wrong format of the message (notice that it is not about the integrity of the MAC, in which case the connection is closed by the server, but it's just about how the message is constructed), `DUP_NAME` indicating that the new filename that the user wants to use is already used for another file, `MISSING_FILE` to highlight that the file with name `old_filename` for which we're requesting the renaming operation is not present in the storage and `INVALID_FILENAME` indicating an error in the naming of either of the filenames.

#### 4.4 Delete Protocol

The delete operation is a further simplification of the upload and rename protocols, with the sequence diagram represented in the following picture:

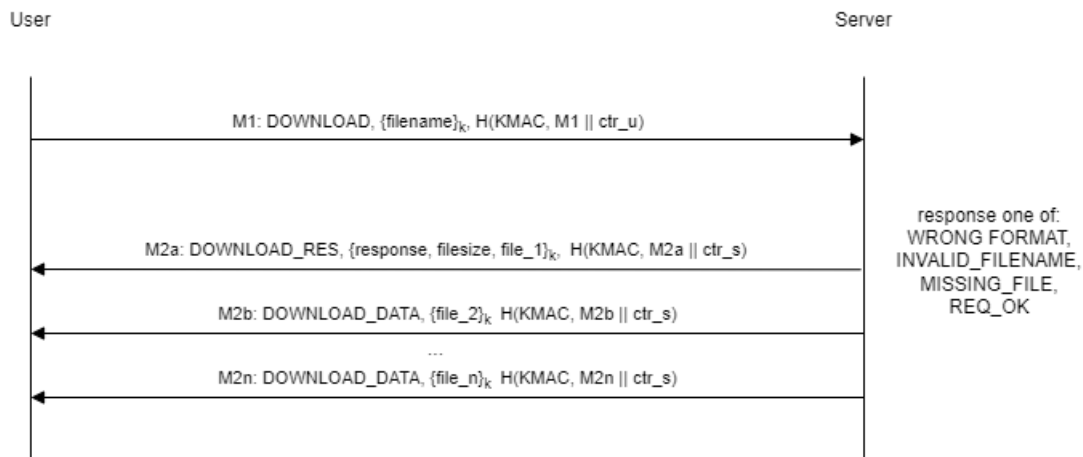


As we can see, the structure of the protocol is exactly the same as the one for the rename protocol: the only difference stands in the fact that the message M1 contains only the name of the file we want to delete, since we're not trying to change anything in this case, but everything else is the same, in particular we use again the MAC in both messages to ensure integrity of request and response and to make sure that an old request cannot be replayed, since it could remove a file from the cloud storage which could've been re-uploaded by the user, as well as an old response, which could allow an attacker to make a user think that a file was not removed when instead it was or vice versa. We can also notice that the same

responses used for the rename protocol are present for this protocol as well, except for the DUP\_NAME response for obvious reasons.

## 4.5 Download Protocol

The download protocol is once again very similar to the upload protocol, but it's actually simpler:



If the user wants to download a file from the cloud storage, they can initiate a download by sending a request to the server with the DOWNLOAD opcode, which can be sent in-the-clear for the same reasons explained in the previous paragraphs. The message, which is message M1 from the sequence diagram, contains the encrypted filename, which must not be sent in-the-clear for privacy and confidentiality reasons; it also contains the MAC of the message concatenated to the counter, which ensures integrity and in particular also ensures that the message cannot be replayed, which would be problematic since an attacker could require the server to download a file multiple times, wasting its bandwidth.

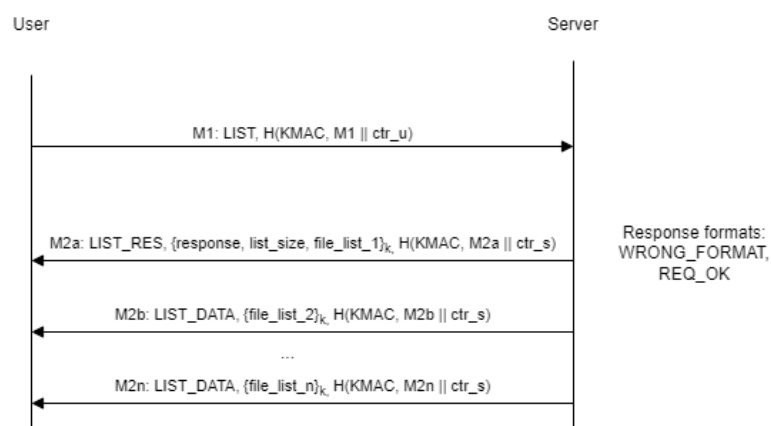
After receiving the user's request for downloading a file, the server checks the MAC and then the filename: if the filename is not present or is not acceptable, the server send respectively a response message containing the MISSING\_FILE or INVALID\_FILENAME responses, while if it is present it sets the response for the message M2a to RES\_OK, it encrypts the response along with the size of the file to be downloaded (which is needed by the user to know how many messages to expect) and then it is sent along with the MAC; this message also contains a portion of the file (or the entire file, if it fits one message), which can be sent already at this point to optimize the operation. After that message, the server starts sending the chunks of the file, in a similar way to how it sends the chunks for the upload operation. In this case, after all the chunks have arrived at the client side, there's no need for the client to send a response to the server to say everything went well or that some chunk is missing: in fact, if something



went wrong, since the system is build over TCP which ensures that all the messages from the server will arrive in order to the client, the only source of failure may be due to a manipulation of one or more messages performed by an attacker, which are noticed when checking the MAC at user side, in which case the connection is closed; if everything went well, there's no point for the server to know it in any case.

## 4.6 List Protocol

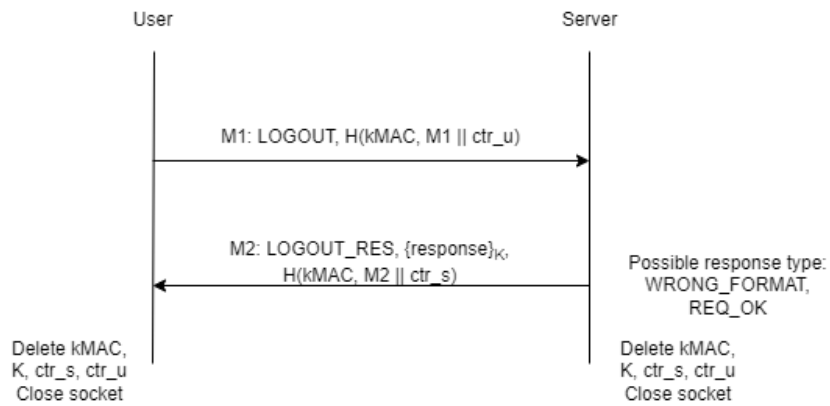
The list protocol works exactly as the download protocol, as we can see from the following sequence diagram, but it's slightly simpler:



As shown in the picture, the structure for the list protocol is the same as the one for the download operation. In M1, we don't have in this case a filename to pass, therefore the payload results to be actually empty for the M1 message: in the server, we check if the payload is truly empty and we return an error in case it is not, but since a normal user should not be able to fill the payload we close the connection since we assume this has been done by a malicious user. For what concerns M2, we have that M2a contains again in the encrypted payload the response, the size of the list to be downloaded and the first portion of the list, with following chunks sent in following messages if the list does not fit a single message.

## 4.7 Logout Protocol

The logout protocol is an important protocol since it is the one in charge to deallocate all the sensitive information that have been established for the session, like the session keys:



The protocol has a very simple format, with a request sent by the user in M1 with opcode LOGOUT and a response from the server, which is encrypted for confidentiality and contains the approval or denial by the server. In case the logout operation is accepted by the server and message M2 is sent, the server can deallocate the key generated for the session, as well as the two counters, and then close the socket; the user, upon receiving the message M2, will see that the logout operation has been accepted and therefore will perform the same deallocation operation performed by the server. It is important to notice that M2 could be stolen or blocked by an attacker: in this case, the user will never receive it, but if this happens the user still deallocates everything because it understands that there's an adversary in the network, since it is impossible for a message to be lost over TCP, therefore after a certain amount of time it performs the deallocation operations anyway. The timeout is actually present every time one of the two sides is expecting a message from the other side, so that if it is triggered everything is deallocated because we must assume either a failure of the other party or the presence of an adversary, which both lead to the closure of the connection.

## 5.0 Implementation

In this chapter we will present some implementation details which are important both to understand how we actually implemented the protocols shown in Chapter 4 and to highlight how some possible threats at code level are handled.

### 5.1 Message exchange

As already said previously, for this project we decided to use TCP as transport layer protocol since it allows to make sure that the messages sent from one side are received on the other side in the same order they're sent; moreover, it guarantees that the messages are delivered, therefore it allows us to assume that if a message is lost then it must've been intercepted by an adversary, therefore the connection must be considered as compromised and closed immediately. However, we need to consider that in this case we are sending messages with a specific format, composed by a header, a payload and a MAC field, therefore we need in our case to define three special functions: the function *build\_message* allows to create a message in the format shown in Chapter 3 by passing the content of the message itself as the different values we need to include in it, specifying in particular the presence or absence of the MAC field and the counter which must be used to compute the HMAC; the function *send\_message* is used to easily ask for a TCP send operation over a message, which is implemented as a structure in our case with all the different fields shown in Chapter 3, so we need to use this redefined send to perform serialization and then actually perform the send operation; the third function is the *recv\_msg*, which is the dual of the *send\_message* function and allows to receive a message by reading field by field the content of the receive buffer for TCP, recreating then the message on the receiver's side. An example of the usage of the first two functions is the one used on client side for the M1 message during the UPLOAD protocol, which is reported in the picture below:

```
m1 = build_message( iv: IV_buffer, opcode: UPLOAD_REQ, payload_length: encrypted_payload_len, payload: encrypted_payload, hmac: true, hmac_key, counter: client_counter);
if(send_msg(socket_id, msg: m1, hmac: true, identity) < FIXED_HEADER_LENGTH + (int)encrypted_payload_len + DIGEST_LEN){
    cerr<<"Cannot send UPLOAD_REQ request to server"<<endl;
    return false;
}
```

As we can see, we call the *build\_message* by passing the IV which has been used for the encryption of the payload of that message (when we don't have anything to encrypt, this field can be set to NULL), the opcode for that message, the length of the payload (which at this point has already been encrypted), the encrypted payload itself, a Boolean which is set to true if the MAC field is present, the key used to compute the HMAC in case it is present (it can be left to NULL if it is not present) and finally the counter which must be concatenated to the message to compute the HMAC. Once the message has been built, we can call the *send\_msg* function to actually send the message M1 over the socket which has been

established for the communication between the user and the cloud storage, passing true if the HMAC is present and an “identity” field which is a string useful to understand what is happening during the execution of different functions: the *send\_msg* returns the number of bytes which have been transmitted to the receiver, which therefore must be equal to the length of the header (which is fixed and equal to 20 bytes), the length of the payload which is actually variable and the length of the MAC digest. The *recv\_msg* function instead simply takes the socket on which to receive a message, a pointer to an empty message structure, a Boolean to say if we should expect a MAC field or not and again the identity: the message is then handled exploiting the structure itself. All these functions are implemented both by the user and the server, so they’re declared in a shared *communication\_utilities.cpp* file, along with other functions for the de-serialization of a message, the retrieval of the size of a file and other I/O operations on files.

## 5.2 Client-side implementation

The client-side application is a very simple command-line application handled by a single thread which allows to take some commands in input and perform the corresponding operations on the portion of the cloud storage dedicated to the corresponding user. In particular, when launching the application the user will be asked for a username, whose validity will be checked against a whitelist in order to make sure the user cannot input anything dangerous as username. In particular, the user will be able to only input alphanumeric characters along with the “\_” characters, as it is shown by the following snippet of code:

```
bool check_username(const string& username){
    char ok_chars [] = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890_-";

    if(username.find_first_not_of( s: ok_chars)!=string::npos){
        return false;
    }
    return true;
}
```

The exact same approach is also used when checking a filename passed from the user, with the difference that in this case the whitelist also accepts the “.” which is needed for the extension. At this point, after inserting the password for opening the file with the private key for the user, the client initiate the key exchange protocol with the server, sending its public DH key to the server in a message with no MAC and opcode equal to AUTH\_INIT as seen in Chapter 4, and the whole protocol is then carried out. The protocol has been already exposed in Chapter 4.1, therefore we won’t again enter in the details of the process: however, it is important to highlight that during the protocol both on server and user side we will

allocate some buffers which will be used to contain sensitive information like the keys and the counters: each time a buffer for such information, as well as for other information like chunk of files to be sent over the network, is created, we collect the pointer to such buffer in a vector, which is used at logout phase or in case an error is encountered to deallocate all the possible sensitive information contained in memory, both on client and server side.

Once the session has been established, the client application enters in a loop waiting for a command to be passed from the user: in particular, the user can only insert the name of the command in caps, for example “DOWNLOAD” or “LIST”, still not specifying at this point the name of the files; the input of the user is checked against a whitelist, which this time allows only for capital letters. Depending on the command, the client applications then performs different checks: for example, for the LIST command no checks are performed and we simply call the corresponding handler function; for the download operation, the user is asked to input the name of the file to download, which again must be considered tainted, so in this case we check if the name of the file may be able to induce traversing in the dedicated storage, in which case we ask the user to input a different name. In case the name is acceptable, we also perform a check in the default download folder for the client application to see if a file with the same name is already present, if it is it will be overwritten. The same checks are performed for the rename and delete operations, while for the UPLOAD operation the check is slightly different: in fact, the first thing that is done is to perform the canonicalization of the name of the file, exploiting the *realpath* function as follows:

```
char* canonicalize(const string& file_name){
    //canonicalization//
    //e.g. ../file.txt => /home/user/myfiles/file.txt
    char* canon_file_name = realpath( name: ("."+file_name).c_str(), resolved: NULL);
    if(!canon_file_name)
        return NULL;
    return canon_file_name;
}
```

After canonicalization, we take only the last part of the obtained filename, in particular the part which follows the last “/” character, which is the name of the file along with its extension, which is checked against a whitelist to make sure the name of the file does not contain unacceptable characters. If the input of the user is not malicious, we pass to check the permissions of the user over the file, in order to make sure the user is not handling a file which they can’t actually handle like the */etc/passwd* file. At this point, if the client actually has the permissions and the name of the file is correct, the upload operation is started.

As a final note, it is important to highlight that the client application allows for a user to log to their dedicated storage from multiple devices at the same time; however, if no requests are sent by a device for a certain amount of time, the server closes the corresponding socket and a new authentication operation must be performed on that device.

### 5.3 Server-side implementation

The server-side application is more complex than the client one, since it has to be able to handle multiple clients at the same time and even the same client from multiple devices, ensuring in any case the consistency of the dedicated storage for each user. In order to do this, the server is created as a process which listens for TCP connections from users and allocates a thread for each one of those connections, which each thread called Worker and whose pointer is kept by the main listener thread to be deallocated at server shutdown (either intentional shutdown or followed by some error). Each worker is implemented as a class which has a set of private members used for communication (like the identifier of the socket which is used to communicate with the corresponding client) and the others used for encryption, decryption and hashing. Along with other utility members; inside the Worker class, we also declare all the functions which must be used by a Worker to handle the requests coming from the user, along with some other utility functions. For what concerns the implementation of the functionalities of the Worker, upon creation it immediately tries to establish the session with the user, performing the key exchange protocol already seen in Chapter 4.1: during such protocol, it authenticates the client exploiting the username, which is used to recover the public key of the client which is used to verify the signature on M3. Once the session has been established, the Worker enters in a loop, from which it exits only in case of error or in case of reception of a logout request: in such loop, the Worker waits for a message, and upon receiving such message it performs a different set of checks depending on the type of message itself, but in all cases it checks the validity of the HMAC field (it is always present in all the messages from the user) and that we don't have an overflow for the client counter, then it proceeds to handle the protocol. It is important to highlight that, for the LIST and LOGOUT commands, the Worker also checks if the payload field is empty before handling the operations for the corresponding protocol, since the message is supposed to have an empty payload: if the payload is not empty and the MAC is still correct, we must assume that the client is trying to perform something which may be dangerous for the system, since it should not be able to put anything inside the payload, so in this case the connection is immediately closed.

At logout (or after an error), each Worker is deallocated and their stack is deleted, in order to delete all the session information. The server is conceived to be resistant to unexpected terminations: every possible software interruption signal is caught by the server main process, that interrupts the operations each thread was performing and smoothly provides to the deletion of sensible data of any worker which was still active.

## 5.4 Protocol implementation

For what concerns the implementation of the protocols, there are a few things which must be studied more in depth, starting from the implementation of the LIST operation. This operation must be able to return the full list of filenames which are stored in the dedicated storage for a certain user, so a naïve implementation for a functionality like this one is to use a vector which stores all the filenames and is converted to a string to be sent to the user (or directly store all the filenames in a string, with each name separated by a special delimiter). This is a very simple solution, but it has two problems: the first one is that the user may have a high number of files stored in the cloud storage, therefore the list may be very long, which results in a high occupation of the memory, which can be an even bigger problem if we consider that we have multiple workers storing different lists at the same time (it is unlikely to have such a very big list since the maximum size for the name of the file is limited to 30 characters, but it's still possible if we have many users with many files to occupy a good amount of memory); the second problem, which is actually more important, is that if we load the list in memory for a Worker but we allow the same user to login to their storage from different terminals at the same time, the list loaded in Worker 1 may be outdated if we perform an upload/delete operation from the other Worker 2, leading to consistency issues which we would like to avoid. To solve both problems, the list is dynamically retrieved at each LIST request by exploiting the "ls" Linux command, which can be executed as follows:

```
string Worker::GetStdoutFromCommand(string cmd) {  
  
    string data;  
    FILE * stream;  
    const unsigned int max_buffer = 356;  
    char buffer[max_buffer];  
    cmd.append( s: " 2>&1");  
  
    stream = popen( command: cmd.c_str(), modes: "r");  
    if (stream) {  
        while (!feof(stream))  
            if (fgets( s: buffer, n: max_buffer, stream) != NULL)  
                data.append( s: buffer);  
        pclose(stream);  
    }  
    return data;  
}  
  
string Worker::get_file_list_as_string(){  
    return GetStdoutFromCommand( cmd: string("ls ../UserData/" + this->username));  
}
```

The Worker will call the *get\_file\_list\_as\_string()* function, which will build the ls command for the folder which indicates the dedicated storage for the user. In order to get the output of such a command, an *fgets()* function call is exploited, but it's actually in its secure version,

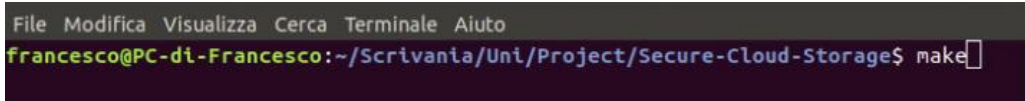
with the *max\_buffer* parameter which ensures not to have a buffer overflow due to the *fgets* function itself. In this way, each time we perform a LIST request, the server will check the current content of the dedicated folder for the user.

Another important consideration, which has been already mentioned in Chapter 3, needs to be done on the upload and download protocol in case large files must be moved between client and server: in those cases, we cannot load in memory the whole file, since its size could span to up to 4GB according to the requisites of the project, which would fill the memory, so we need to divide the file in small chunks to be sent in different messages. In the standard implementation for TCP contained in the *sys/socket.h* library, the default size for the sender and receiver buffers is equal to 32KB, which is doubled in order to accommodate the overhead of the send and receive operations: such size can be changed, but after some tries we notice that this change is not systematic and in particular is not reliable, since in some cases such change is not applied and results in an overflow. For this reason, we decided to use messages of up to 16KB for the payload, which reduces the probability that the sender/receiver buffer is filled completely. However, using messages of this size may lead to a huge overhead in terms of I/O operations: in fact, when sending a chunk over the network, we need first to read such chunk from the file, encrypt the chunk, send it over the network, decrypt it and then write it in the corresponding file at the right position (in this case, since TCP ensures that the order of the messages is maintained, a write in append is sufficient); if we use messages with payload equal to 16KB for a 50MB file, we need to perform 3,125 reads and 3,125 writes, which are costly operations since they work on the file system and are therefore very slow. To overcome this problem, we implemented a higher level buffer in which we load bigger chunks, in particular we load 16MB, which are a reasonable size and does not impact too much on the memory (remember that if we have multiple Workers performing multiple download operations, we may have issues with the amount of memory allocated for the chunks!), and we iterate over such buffer to collect 16KB for each iteration to be sent in a new message until all the bytes are sent, then we perform a new read to fill again the buffer; at the receiver, we perform the dual operation, so the buffer is incrementally filled upon receiving the messages and, when full, it is written in the corresponding file. In this way, we can perform only 3 reads and 3 writes for a 50MB file, which basically eliminates the impact of the I/O operations.



## 6.0 User Guide

In this Chapter we will briefly present how to test our application. First of all, you can find the application to the following [Github link](#): you can download the repository on your PC in any folder you like. Once the repository is cloned, you will need to enter into the “Secure-Cloud-Storage” folder to build the executable files, which will be stored in the “Build” folder. In the “Secure-Cloud-Storage” folder, run the “make” command to build the executables:



```
File Modifica Visualizza Cerca Terminale Aiuto
francesco@PC-dl-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage$ make
```

The main folder of the project is divided in 2 folders: in the “Build” folder, we have in “Keys” all the keys both for the server and the clients, stored accordingly to the pre-requisites of the project, so for example the “Keys/Client” folder contains all the private and public keys for the clients, while the “Keys/Server” folder contains a folder for the keys of all the clients and the keys of the server along with its certificate. In the “CertAuth” folder contained in “Build”, we can find the certificate for the CA along with its CRL. The build folder then contains two important folders which are “Client” and “Server”: the former is the current folder of the main application, therefore it contains the file which are downloaded from the cloud storage of the user (since we consider that each client application should run in a different machine, we do not separate the folders for each possible client, but all clients put their file in this same folder), so also calling an upload operation passing simply a filename will look into this folder for the presence of the corresponding file. This folder will also contain the executable for the client application. The “Server” folder instead contains the executable for the server application and a sub-folder “UserData”, in which we create a sub-folder for each user which corresponds to their dedicated storage in the cloud. Each user will therefore be able to access the sub-folder “Build/Server/UserData/” + username and not the other sub-folders.

The second sub-folder of the main folder is called “Source” and contains all the code for our project: in particular, it is divided into three sub-folder, one containing the code for the client, one containing the code for the server and the last one containing code which is useful on both sides. The client and the server files have a similar organization, since they both identify a main file (which is the starting file for the corresponding application), a conn\_functions file for the handling the connection between client and server, a protocol\_functions file for the functions implementing the DH-STS protocol and clean-up functions and a logic\_functions file containing the implementation for the different protocols (the server also contains the files for the definition and allocation of the workers). The Common\_Libs sub-folder contains instead two main file, one called crypto\_utilities

where all the security functions are defined and the other called communication\_utilities and containing the TCP operations and the I/O operations.

Finally, the main folder contains also a makefile and a CmakeLists which is a file created by CLion, which is the IDE we exploited to create our project.

After running the make command, open a terminal in “Build/Server” and launch the Server executable using the ./Server command: at this point, you will be asked for the server PEM pass phrase; for simplicity, all the pass phrases, both for clients and server, are FOC2022. The pass phrase will not be shown while writing it as a security measure. After the input of the password, the server will be up and listening for connections from users (notice that it allocates the listener on port 2210, so make sure it is free before running the server application):

```
francesco@PC-dl-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage$ cd Build/Server/
francesco@PC-dl-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage/Build/Server$ ./Server
Initialization in progress ...
Enter PEM pass phrase:
Initialization terminated ...
Listening for connections
█
```

At this point, in a new terminal you can do the same for the client, moving inside the “Build/Client” folder and running the ./Client command to launch the client, which will ask for a username first in this case (you can choose up between mirco and francesco) and the pass phrase, which is again FOC2022. The client application will print the list of available commands that can be run:

```
francesco@PC-dl-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage$ cd Build/Client
francesco@PC-dl-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage/Build/Client$ ./Client
Please, type your username (maximum 20 characters):
mirco
Enter PEM pass phrase:
*****
***** SECURE CLOUD STORAGE*** *****
*****
Type one of the following commands to start:
>$ HELP: print this message
>$ LIST: list all uploaded files
>$ DOWNLOAD: downloads requested file
>$ UPLOAD: loads requested file into your cloud storage space
>$ RENAME: renames a file in the cloud
>$ DELETE: removes a file from the cloud
>$ LOGOUT: closes connection with server and exits the service
>$
>$ Please, enter a command (type HELP to see a list of commands available):
>$ █
```

All the commands must be given in input in capital letters, otherwise the command is not recognized. The HELP command will print again the list of available commands. The LIST

command instead will print the list of files that are currently stored in the user's dedicated storage inside the server:

```
>$ Please, enter a command (type HELP to see a list of commands available):  
>$ LIST  
small_secret.txt  
very_big_secret.txt  
>$ Please, enter a command (type HELP to see a list of commands available):  
>$
```

To ask for a DOWNLOAD or any other operation on files, it is mandatory to first input the name of the operation, then input the name(s) of the file(s) when asked by the command prompt. For example, the DOWNLOAD command will print a string asking for the name of the file to be downloaded from the cloud storage: if the name of the file is correct and the file is present in the cloud storage, it will be downloaded on the client's current folder; if the file is already present in such folder, it will be overwritten. The first picture below is what is shown on the user's side, while the second picture shows what happens on server's side:

```
>$ DOWNLOAD  
Please insert the name of the file you want to download  
>$ small_secret.txt  
File size is: 15 bytes  
Downloading...  
File downloaded
```

```
New connection established with 127.0.0.1  
[Worker for: mirco]: Session established  
[Worker for: mirco]: Requested file size is: 0  
[Worker for: mirco]: File not found!  
[Worker for: mirco]: Requested file size is: 15  
[Worker for: mirco]: Download correctly completed
```

For the UPLOAD operation, the procedure is the same, but the client will actually print the progression in the upload itself. For what concerns the RENAME operation, the user will be asked to first input the old name which they want to change, then the new name they want to give to the file:

```
>$ Please, enter a command (type HELP to see a list of commands available):  
>$ RENAME  
Please insert the name of the file you want to rename  
>$ small_secret.txt  
  
>$ Please insert the new name you want to give to the file (must not be already  
present as a name of a file in the storage)  
>$ very_small_secret.txt  
The file was renamed successfully!  
>$  
>$ Please, enter a command (type HELP to see a list of commands available):
```

The DELETE operation works the same way as the UPLOAD operation. To logout from the client, you can run the LOGOUT operation, which closes the connection and deletes all the sensitive information established for the communication between client and server. To close the server, it is enough to send an interruption to it using CTRL+C, which raises an interrupt handler which closes all the active connections, deallocates all the workers and deletes all the sensitive information like the session keys which were still active at the time:

```
>$ Please, enter a command (type HELP to see a list of commands available):  
>$ LOGOUT  
The logout was successful! See you next time!  
Shutting down client  
francesco@PC-di-Francesco:~/Scrivania/Uni/Project/Secure-Cloud-Storage/Build/Client$
```