UNIVERSITÀ POLITECNICA DELLE MARCHE

Facoltà di Ingegneria Informatica e dell'Automazione Industriale

CryptoFile: Development of a Distributed Storage System

with

AONT Transform

Student:

Manfrini Pier Luigi

Supervisor:

Spalazzi Luca

A.A. 2019/2020

INDEX

Chapter 1 ~ INTRODUCTION ~

The purpose of this project is showing and testing the possibility to realize a distributed online storage system whose security is independent from the cloud service providers.
In other words: is it possible to store data and access them in a way that they result securely stored and inaccessible, even partially, to anyone including those providers?
The answer, as we will see later on this discussion, is yes and this is where the AONT transform starts playing its crucial role as it lets us crypt (transform) a bunch of data without the necessity of storing the key used in the process thus allowing us to manage the encrypted file independently from its key which represent a remarkable advantage over other encrypting techniques.
As a matter of fact this characteristic distinguish this *"data masking"* method from the other ciphers algorithms and make it so that it doesn't belong to any encryption family because its key is public, stored in the data itself and therefore it doesn't need to be traded between two communicants in order for them to share some AONT masked data. More precisely it is defined as an unkeyed, invertible, randomized transformation and is impossible to invert unless we get to obtain all of its output. As it is suggested by its name, in fact, AONT stands for All-Or-Nothing-Transform and turned out to be an essential procedure when applied before any other encryption method since it enhances significantly its security.

What do we mean with distributed?

We call this whole system distributed since we are going to split a single data file into a chosen number of slices and store, after being encrypted, each in a different online storage cloud making them available in a wider and more distributed environment that will grant more flexibility and efficiency.

For the purpose of testing this technique in our project we worked with Google Drive and Dropbox API's (Application Programming Interface).

In this dissertation we will look closer at the AONT mechanism (*Chapter 2*), its relationship with the other encryption methods, and how and why they work so well together. In fact, we can consider AONT as a process to apply before another encryption to increase its strength without increasing its key size as in many bureaucratic situations it happens frequently that, in order to comply some government cryptography regulations, we may want to limit the key size used, for example, to a maximum of 8 bits.
Later (*Chapter* 3) we will take an overview of CryptoFile's software implementation and some usage samples. This will lead us towards some statistics regarding execution time depending from the number of slices wanted and we will also consider the estimated brute force attack time needed to get access to the plain data if an offender wanted to retrieve it in an illegal way (*Chapter 4*).

Chapter 2 ~ All-Or-Nothing-Transform ~

2.1 An introduction to AONT

The original AONT mechanism was firstly defined by Ronald Linn Rivest, a cryptographer and an Institute Professor at MIT, back in 1997 in his paper "All-Or-Nothing Encryption and The Package Transform" which we closely stuck to during the development of the CryptoFile's software.

In simple words its core functioning consists in splitting the original plaintext[1] into different blocks and encrypt each with a random key in order to form a *pseudo-message,* then appending one more block which consist in the result of the XOR operation between the previous blocks' hashing and the key used to encrypt them all.

As we just saw AONT makes use of a cipher algorithm, but since we eventually store the key in the data itself (obviously is not visible inside the data) it is not classified as an encryption but as a transformation which doesn't allow any attacker to even partially decode any plain data as he would need all the AONT output to obtain the source file.

In fact, as we have previously anticipated, this Rivest's technique is a clever way of "encryption" which has the peculiar property that one must obtain the entire ciphertext before he can determine even one message block. The only workaround would be a brute force to every single transformed block but, as AONT is not classified as a true encryption method, we don't have any restriction regarding the key size so we can use an heavy and secure cipher algorithm (such for example AES[2] 256-bit) to transform each plaintext block therefore giving a really hard life to the attacker who will spend an absurd quantity of time proportional to the number of slices.

In our project we used, for the encryption phase, the AES-256 in CBC *mode of operation* (Cipher Block Chaining) which now we are going to explain.

[1]From now on we will refer to the unencrypted file data as plaintext.
[2]Advanced Encryption Standard (AES), also known as Rijndael

2.2 AES ~ Advanced Encryption Standard

The Advanced Encryption Standard is today one of the most used encryption algorithms due to its security, performance and flexibility. It is also known as Rijndael which is a family of ciphers differing in key and block size, whose developers were two Belgian cryptographers Vincent Rijmen and Joan Daemen. In 1997 they submitted a proposal to NIST (National Institute of Standards and Technology) which were starting a campaign looking for a successor to the aging encryption standard DES[3].
At NITS they chose three members among the Rijndael family each with a block size of 128 bits, but with different key lengths: 128, 192, 256 bits.
As a result, in 2001 AES was announced as the new encryption standard in the United States.
The algorithm works repeating several times operations such as permutations, substitutions and linear transformations on data blocks (hence the name "blockcipher") of 16 bytes (128 bits) organized in a two-dimensional array as follows:

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Each cycle is called *"round"* and it is composed by the same aforementioned steps but executed with a unique round-key which is calculated out of the encryption key and then incorporated in the calculations.

Also, the key length determines the number of *rounds* applied to the input which is evaluated as follows:

- 10 rounds for 128-bit keys
- 12 rounds for 192-bit keys
- 14 rounds for 256-bit keys

[3] Data Encryption Strandard (DES)

So AES-128, AES-192, AES-256 use all a 16 bytes block-size but they differ in the key-size which is respectively of 16, 24 and 32 bytes.
At the time of writing no practicable attacks exist against AES since it would take billions of years to a today's supercomputer to crack a 128-bit AES key. And 256 bits keys also exist.
Resuming the speech about the block-size, what to do and how to deal with the encryption of files that are more than 16 bytes in size? Here comes the necessity to explain what a *mode of operation* is.
We are going to look at the CBC (Cipher Block Chaining) mode which, by the way, is the *mode of operation* chosen for AES-256 in CryptoFile's software.

2.3 CBC ~ *Mode of Operation*

In cryptography a block cipher *mode of operation* is an algorithm that uses a block cipher, the latter being not suitable for a secure cryptographic transformation (either encryption or decryption) of files longer than a blocksize (16 bytes for Rijndael).
Most of the *modes of operation* require to be initialized with an *Initialization Vector* which can be random and doesn't need to be securely stored, hence can be public. However, we need to keep in mind that different IVs produce different ciphertexts even when the same plaintext and key are used as inputs.
In our software the initialization vector is kept the same for every encryption/decryption process and is equal to { '1', '2' , '3' , '4' , '5' , '6' , '7' , '8' , '9' , '0' , '1' , '2' , '3' , '4' , '5' , '6' }
(16 chars so 16 bytes, hence matching the cipher blocksize).

Here (*Fig.1*) we can see how the whole CBC encryption process works. The trunks of plaintext arrays that we see are the result of the original plaintext divided in blocks of 16 bytes each with the last one being padded with zeros if needed.

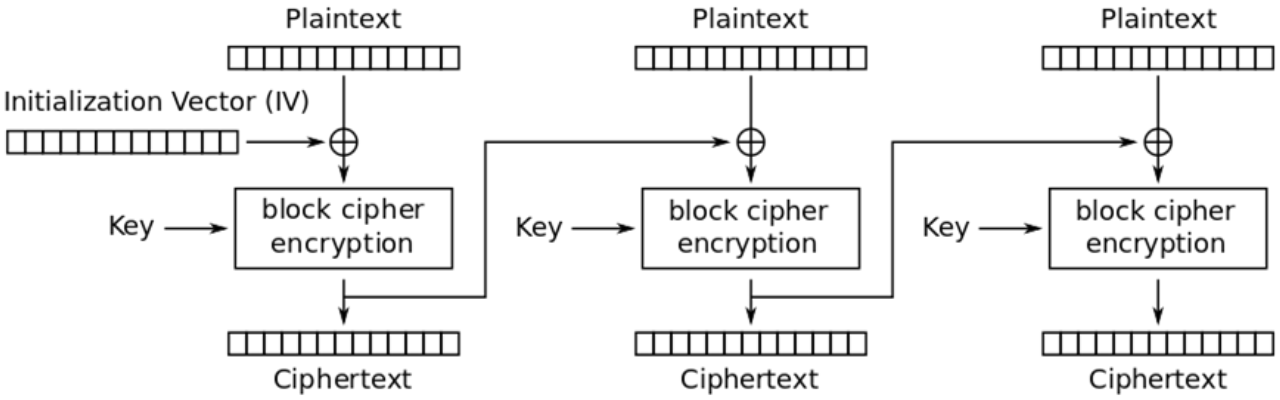

Figure 1 - Cipher Block Chaining (CBC) mode encryption

So, if the first block has index 0, the formula for CBC encryption is:

$$C_0 = E_K(P_0 \oplus IV)$$
$$C_i = E_K(P_i \oplus C_{i-1})$$

P : Plaintext block

IV : Initialization Vector

C : Ciphertext block

E_k : block cipher encryption

Here (*Fig.2*) we have a look also at the decryption process:

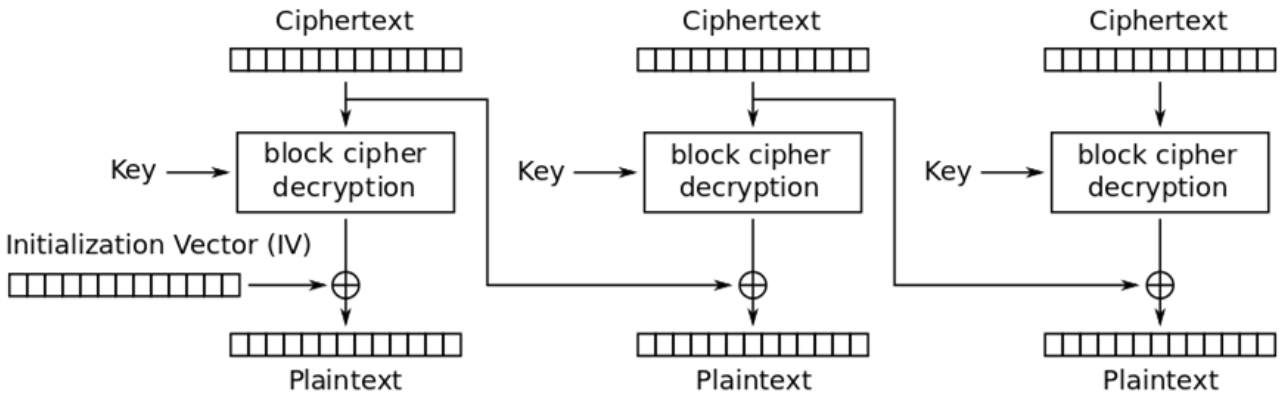

Figure 2 – Cipher Block Chaining (CBC) mode decryption

These are the mathematical formulas involved in the decryption process:

$$P_i = D_K(C_i) \oplus C_{i-1}$$

$$P_0 = D_K(C_0) \oplus IV,$$

AONT separable or not?

As already anticipated before, dispersing files across multiple platforms comes to be handy mostly in terms of availability, reliability and proximity, but what is not obvious about it is that it can enable security to be achieved without relying on encryption keys. Perfectly suiting this dispersal purpose is the All-Or-Nothing-Transform which at the same time grants a *strongly non-separable encryption.*
Let us explain a bit what that means…

2.4 Strongly non-separable encryption

Most popular encryption modes have the issue that an adversary can get a block of plaintext by decrypting just one corresponding block of ciphertext and this weakness is what defines a *separable encryption.*
Since we have already illustrated how CBC works, we are going to use it to explain in practical terms what a *separable encryption* is.

Let the *s* blocks of the plain message be denoted as m_1, m_2, m_3, … , m_s

The *Cipher-Block-Chaining* mode gets them as input and, using an *initialization vector* IV and a key K, produces blocks of ciphertext c_i *with* $i = 1, 2, \ldots, s + 1$.

Remembering the formulas above from CBC :

$$C_i = E_K(P_i \oplus C_{i-1})$$
$$P_i = D_K(C_i) \oplus C_{i-1}$$

It is clear that any one of the *s* Plain message blocks can be obtained just decrypting only one Ciphertext block and if the key is weak enough the adversary's key-search problem becomes relatively easy.
On the other hand, if a block cipher encryption produces blocks of ciphertext that are impossible to determine without decrypting all of them, then we can define that encryption mode as *strongly non-separable*.
AONT proposes to achieve *strongly non-separable* encryption if applied before a normal encryption, so we assume that the underlying process is such that all ciphertext blocks must be decrypted in order to obtain the plaintext or even to determine any property of any message block.

The ideal process to follow would be to transform a message sequence m_s into a *"pseudo-message"* sequence $m'_{s'}$ (where $s' \geq s$) with an All-Or-Nothing transform and then encrypt the pseudo-message with a public cryptographic key K to obtain the ciphertext sequence.
Let us present the transformation scheme as designed by Rivest.

2.5 All-Or-Nothing Transform

We propose here the all-or-nothing transform created by Rivest and called the "package transform". It will use a block cipher but, as previously said, no secret keys are used and that block cipher doesn't need to be the same as the one used later to encrypt the pseudo-message generated as the package transform output. We assume that the key used in the package transform is large enough to consider a brute-force against it infeasible, while the outer key chosen to encrypt the pseudo-message can be weak due to commercial regulations.

Here is how the transform works:

- Let the input message be m_1, m_2, …. , m_s

- Choose a random key K' for the block cipher

- Calculate the output pseudo-message sequence m'_1, m'_2, …. , $m'_{s'}$ with $s'=s+1$ as follows:

$$m'_i = m_i \oplus E(K', i) \quad \text{for i = 1, 2, 3, … , s}$$

$$m'_{s'} = K' \oplus h_1 \oplus h_2 \oplus \ldots \oplus h_s$$

where

$$h_i = E(K_0, m'_i \oplus i) \text{ for i = 1, 2, … , s}$$

where K_0 is the fixed publically-known encryption key used in the outer encryption.
The idea here is that we can choose for K' a size of 128 or 256 bit as it is not a secret shared key therefore is not subjected to any limitation/regulation which instead need to be followed in the pick of K_0.
It is easy to see that the package transform is invertible as, given all the pseudo-message, we can easily retrieve K' as shown:

$$K' = m'_{s'} \oplus h_1 \oplus h_2 \oplus \ldots \oplus h_s$$

Let's point out that if any pseudo-message block is unknown, we can not compute K'. Once we have the key, the process to get back to the original plaintext is quite straightforward:

$$m_i = m'_i \oplus E(K', i) \text{ for i = 1, 2, …, s}$$

This was the Rivest Transform as it explained in his paper published in 1997 and before concluding discussing it lets remark some "highlights" and do some clarifications.

- The all-or-nothing transform is merely a pre-processing step therefore can be used with already-existing encryption methods without the need of changing them.
- This method works only when the message to be encrypted has a finite size whereas for example CBC mode by itself works perfectly well with infinitely long messages.

2.6 Aont in CryptoFile

In the development of CryptoFile's software we managed to perform the package transform leaving out of the project's goal the subsequent encryption. We closely stuck to Rivest's algorithm, however we took out of our equations the i variable which represented the message block position since we managed every slice position in our local database. So, our pseudo-message is produced as follows:

$$m'_i = E(K', m_i)$$

and the last pseudo-message block remains defined as:

$$m'_{s'} = K' \oplus h_1 \oplus h_2 \oplus \ldots \oplus h_s$$

where, in our case, h_i is the hash digest of the corresponding pseudo-message block. So we can each time retrieve the randomly generated key exactly as Rivest does:

$$K' = m'_{s'} \oplus h_1 \oplus h_2 \oplus \ldots \oplus h_s$$

Now that we have reviewed the core functioning of the project we can move on to the practical side and see the software's implementation of CryptoFile.

Chapter 3 ~ CryptoFile's Implementation ~

Figure 3 - CryptoFile Sample Logo

✓ Language:	C++	ver. 17
✓ Compilers Used:	g++ / clang++	
✓ Developed in:	Ubuntu 64 bit	ver. 19.04
✓ APIs Handler:	C++ REST SDK	ver. 2.10.12
✓ Encryption Library:	CryptoPP	ver. 8.2.0
✓ Database Handler:	SQLite	ver. 3

GITHUB HTTPS download link: https://github.com/pyer96/CryptoFile.git

The CryptoFile's structure consists of 3 main modules inside the CryptoFile directory: the Aont, Db and Session folders, designed to manage the transformation, the data and the APIs respectively. Other files crucial for the User interaction are the InitCryptoFile and the CryptoFileSession class which we will explain after the 3 core aforementioned modules.

Before showing how CryptoFile works, we would like to start this guided CryptoFile's tour with the Database structure (*Fig.4*) as it gives a brief and general understanding of what data the software needs to get the job done.

Database schema:

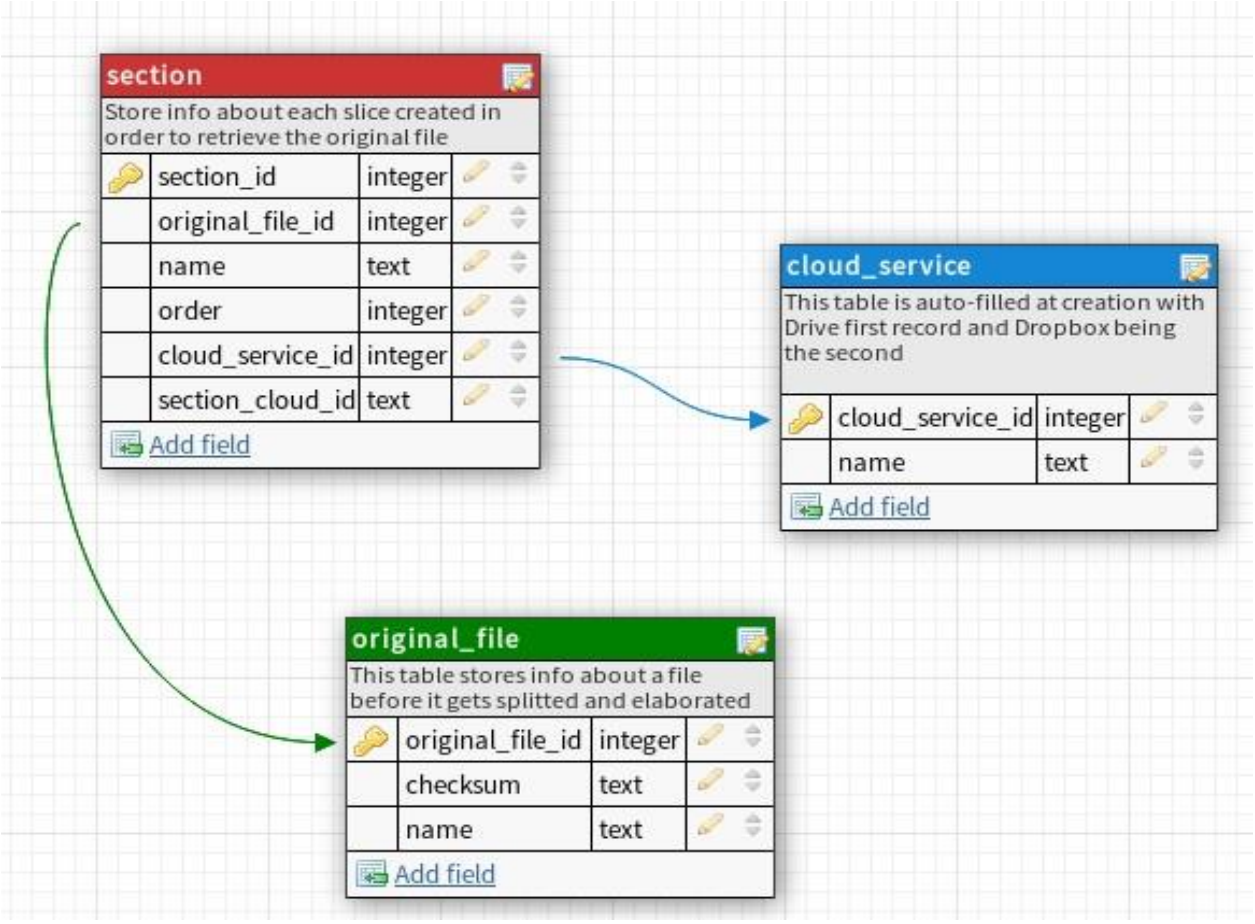

Figure 4 - CryptoFile's Database Structure

The early stages of the software development were dedicated to the establishment of a reliable communication with our clouds which are Google Drive and Dropbox.
We rely on the C++ REST SDK, a Microsoft cross-platform library for cloud-based client-server communication in native code for accessing and authorizing REST[4] services in a productive and, most importantly, asynchronous way which is crucial to achieve efficiency, responsiveness and to avoid annoying lags in the user interface.

The main requirement for a client-server communication with our cloud services is the authorization process which is a primary step we are going to illustrate in the following paragraph.

Authorization Flow

To keep the user authorization process secure and "outside" our software we adopted the Oauth 2.0 industry-standard protocol that provides a specific authorization flow for web, desktop and mobile applications. It was developed by the Internet Engineering Task Force (IETF) and it is mainly based on security tokens exchange which are set of information that facilitate the sharing of identity and security information in heterogeneous environments or across security domains.

With the "outside our software" statement we want to point out that the user will never log in giving us its credentials, instead its authentication will be held in a secure environment provided by the Cloud Services themselves.

The conventional Oauth 2.0 interaction involves, in fact, the exchange of some *representation* of resource owner authorization for an *access token* which has proven to be an extremely useful pattern in practice. This specification enables Clients to

[4] Representational State Transfer (REST) is an architectural style for APIs defined by Roy Fielding in 2000. In simple words when the Server responds to the Client request it will send a *representation* of the *state* of the requested resource. The Client needs to specify an identifier for the wanted resource through its URL (Uniform Resource Locator) , also known as **endpoint.** Client also need to specify the operation to apply to that resource. It can do this in the form of an HTTP method such as GET, POST, PUT, DELETE and more.

request and obtain security tokens from authorization Servers which act as STS (Security Token Service). The communication only involves an HTTP client and a JSON parser making it universally available in modern developed environments.

Here are some examples of this process as explained by Google's and Dropbox's Guides.

Google's visual explication:

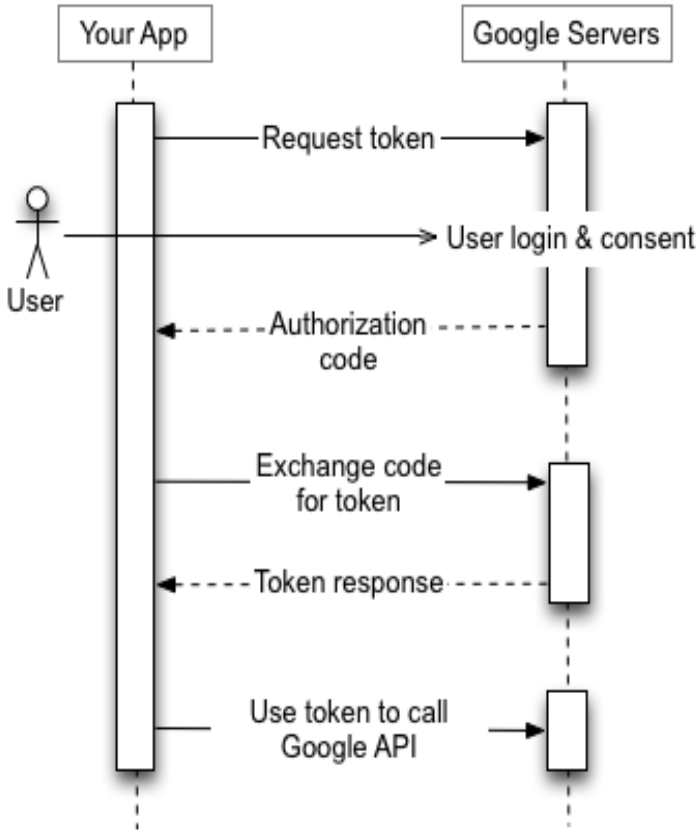

Figure 5 - Webflow_1

Dropbox's visual explication:

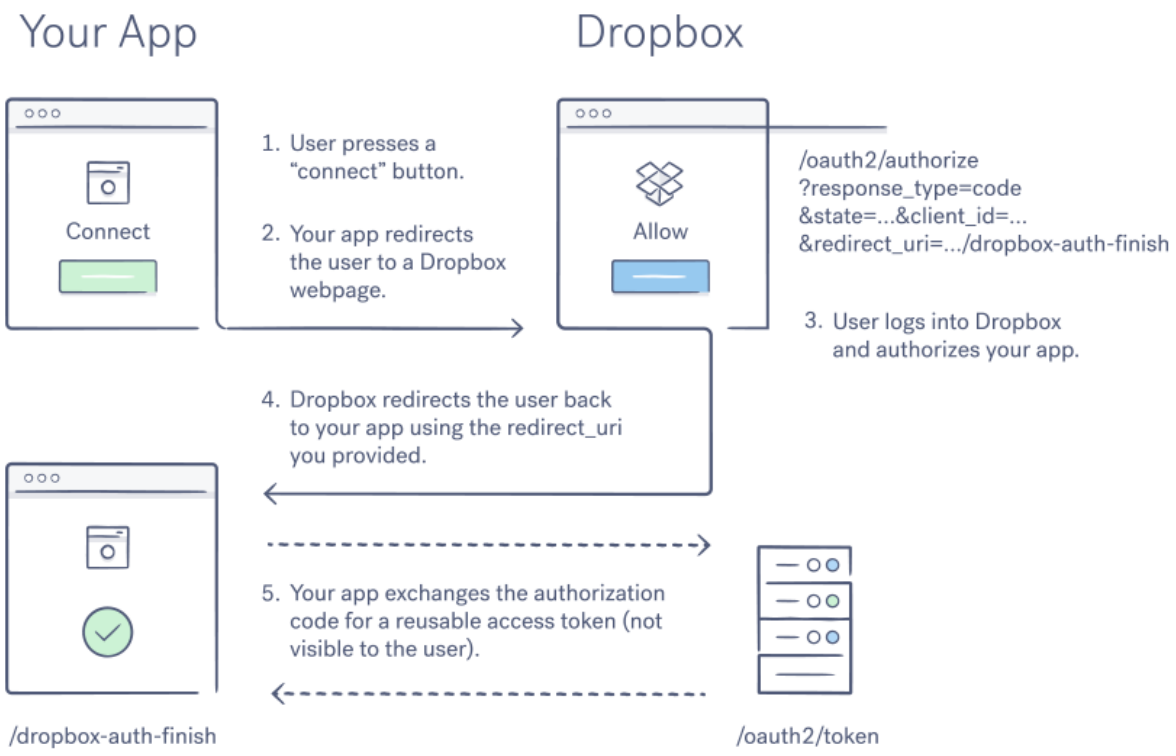

Figure 6 - Webflow_2

As we can notice from these schemes (*Fig.5 & Fig.6*) the basic concept behind them is that we need to exchange with the APIs the *authorization code* that we read from the URI we will be sent to for an *access token* which will grant us access to that API. However, in order to get the *authorization* token, we need to own some other parameters such as the *client_id* and the corresponding *client_secrets* strings that we obtained once the application CryptoFile has been registered to the cloud storage providers.

Below we present the code snippet that shows the CryptoFile's class header involved into the authorization flow.

```cpp
/** This class implements oauth2 functionalities for CPPRESTSDK and is used to
 * read the authorization token from redirect uri and trade it for an access
 * token. Reading is done with a handler added to a listener.
 */
class ServiceAuthenticator {

private:
  oauth2::experimental::oauth2_config m_oauth2_config;
  std::unique_ptr<experimental::listener::http_listener> m_listener;
  pplx::task_completion_event<bool> m_task_completion_event;
  std::mutex m_resplock;

public:
  ServiceAuthenticator(std::string auth_endpoint, std::string access_endpoint,
                       std::string client_id, std::string client_secrets,
                       std::string redirect_uri,
                       std::string scope = std::string());
  ~ServiceAuthenticator() { m_listener->close().wait(); }
  pplx::task<void> open() { return m_listener->open(); }
  pplx::task<void> close() { return m_listener->close(); }
  const auto &oauth2_config() const { return m_oauth2_config; }
};
```

To get a better understanding of the class composition for the communication with the API, we are showing in *Figure 7* the UML (Unified Modeling Language) scheme of the complete hierarchy for the *Session* namespace in our software which is the one responsible for the dialogue with the REST services.

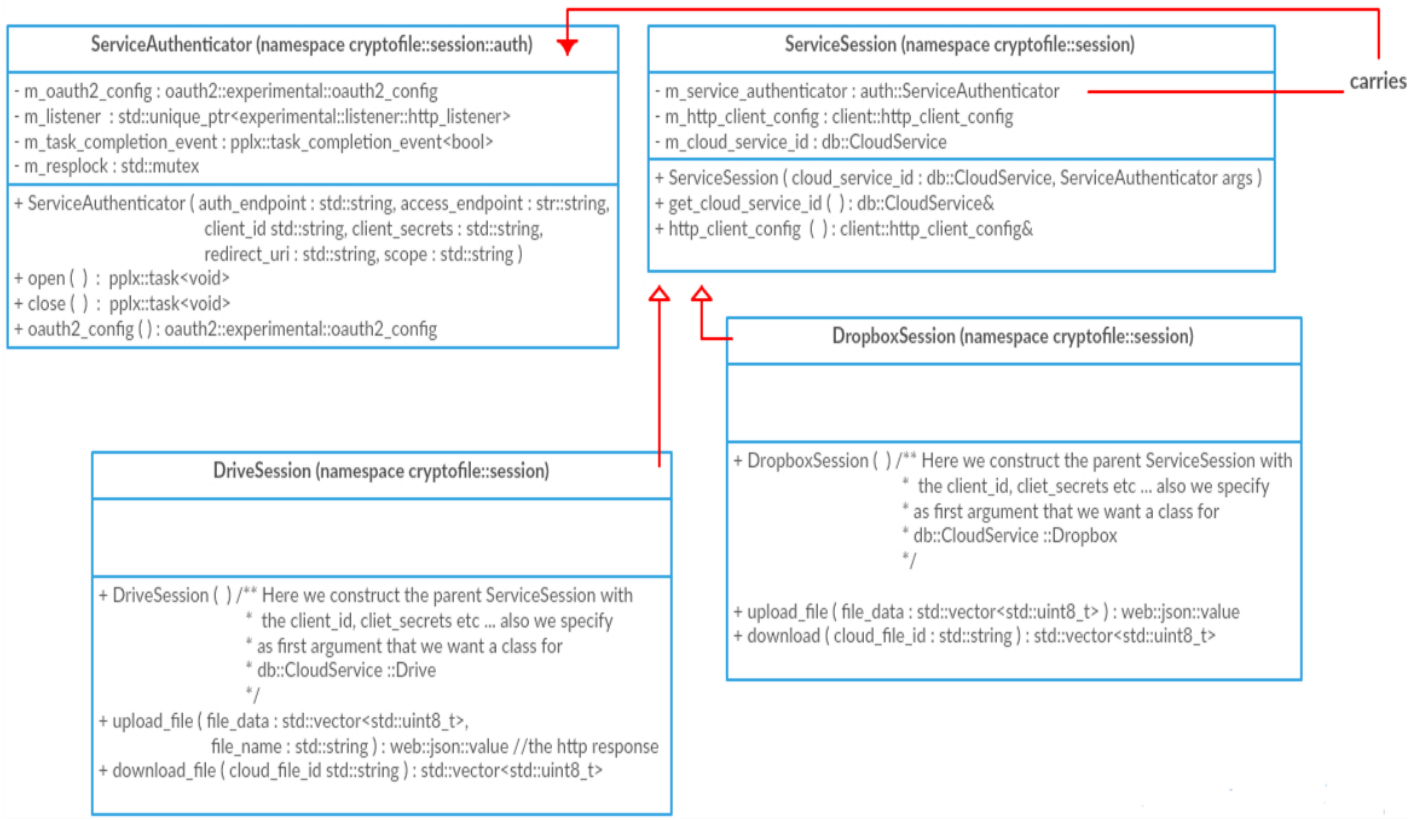

Figure 7- Session Namespace's Classes'Hierarchy

Throughout the code within the namespace *session* we rely on classes from the C++ REST SDK library such as *http_client_config* which gets the configuration parameter needed for the authentication flow from the *oauth2_config* in order to create an *http_client* class, used to maintain a connection to an HTTP service for an extended session and also practical to perform requests.

The *oauth2_config* class constructed inside the *ServiceAuthenticator* class automatically encapsulates functionalities for authenticating requests with an access token.

Hence to perform a valid request, once we are already authenticated, we will need to pass the *oauth2_config* with the access token to a *http_client* through the *http_client_config.*

We are showing an example of how the *upload_file* method is implemented within the DriveSession.cpp:

```
1.  /** First we initialize the client as a web::http::client::http_client class from
2.   *  the CPPRESTSDK library. It needs the base uri for the specified API as well as
3.   *  the config parameters to be constructed.
4.   *  Once we have a working client we define and model our web::http::http_request and pass
5.   *  it to the client to be submitted to the servers which will return a JSON response
6.   *  that we store in the provided web::json::value response variable.
7.   *  The response is the only way we can get to know the cloud given id for
8.   *  the uploaded resource, hence it is set as the return value.
9.   */
10.
11. web::json::value DriveSession::upload_file(std::vector<std::uint8_t> file_data,
12.                                            std::string file_name) {
13.   std::cerr << "upload_file\n";
14.   web::http::client::http_client client("https://www.googleapis.com",
15.                                         http_client_config());
16.   auto request = http_request(methods::POST);
17.   request.set_request_uri("/upload/drive/v3/files");
18.   auto &head = request.headers();
19.   head.set_content_type("application/octet-stream");
20.   head.set_content_length(file_data.size());
21.   head.add("uploadType", "media");
22.   head.add("name", file_name);
23.   request.set_body(file_data);
24.   web::json::value response_json;
25.   client.request(request)
26.       .then([=, &response_json](http_response response) {
27.         auto task = response.extract_json();
28.         task.wait();
29.         response_json = task.get();
30.         std::cerr << "resp = \n{\n" << response_json.serialize() << "\n}\n";
31.       })
32.       .wait();
33.   return response_json;
34. }
```

Aont's implementation

We already saw how an All-Or-Nothing Transform works in theory and how it fitted inside CryptoFile, so we are now going to evaluate it in practical terms to see its code implementation. The core elements of the AONT execution are the *Section* classes which are designed to hold and manage the single slices deriving from the plaintext partitioning or from the download phase.

To achieve a higher grade of performance we shaped those classes to perfectly fit their purpose which, depending on the case, can be different. For example, a Section that represent a slice to be encrypted needs a different approach from another one to be decrypted. Also, the m'$_s$' (referring to the Rivest's terminology) slice that stores the bit string resulting from the Xor operation between the hashed sections and the key K' needs a different holding class.

Sections UML scheme:

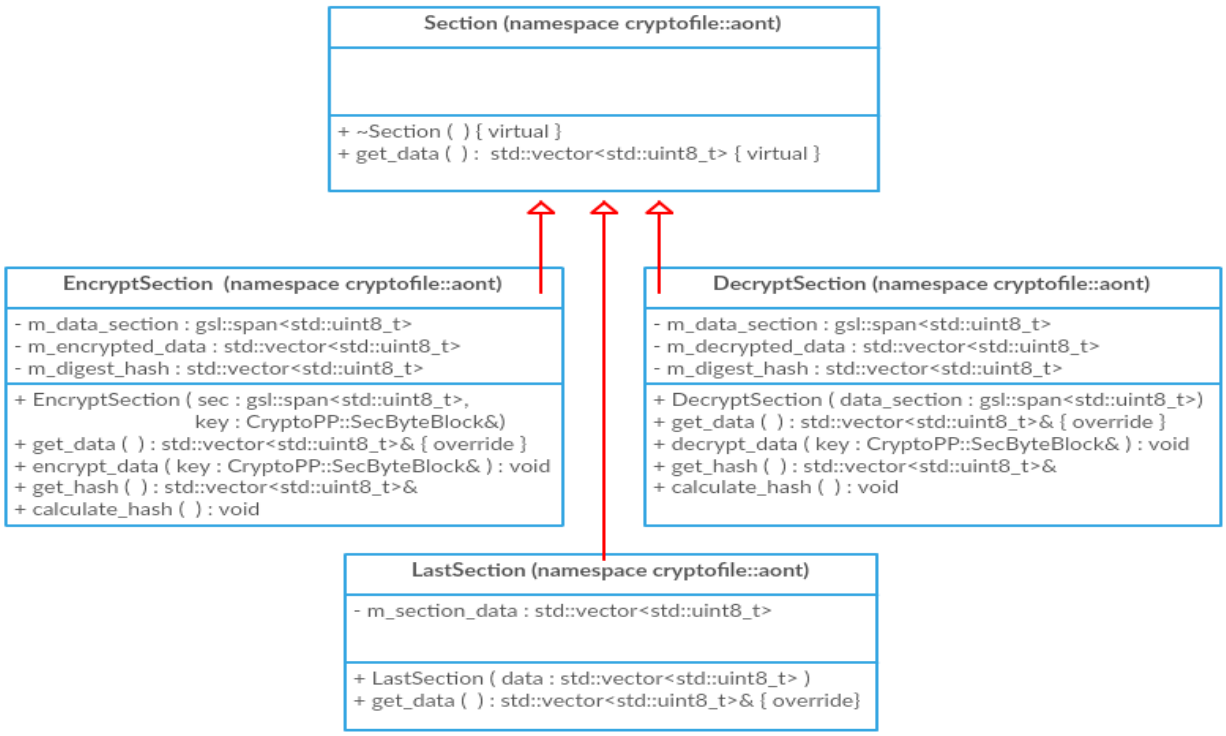

Figure 8 - Section Classes' Hierarchy

Where do we perform the actual All-Or-Nothing Tranform?

Here we present the two core functions, inside the *cryptofile::aont* namespace, that allow us to transform the plaintext into a Rivest's encrypted pseudo-message and to perform the reverse operation.

They are the *aont_mask* and *aont_restore* functions and are designed to reduce significantly the memory footprint of our software since we never pass the whole plain file between structures through a move/copy operation, instead we always refer to different portions of the original data using a span<T> (a lightweight abstraction of a contiguous sequence of value T stored only once in memory) where T in our case is a std::uint8_t.

The class template *span* is getting included into the upcoming C++ 20, but since we developed our software with the current 17 version of C++, we included it from the Guidelines Support Library (GSL) which contains functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation.

Throughout our software we manage all data as vectors of uint8_t; this grant us the ability to operate with any input file whose size is a multiple of 8 bits, so basically every kind of file.

Aont_mask

```
1.  void aont_mask(
2.      std::vector<std::uint8_t> &data, std::size_t sections_number,
3.      std::function<void(std::vector<std::unique_ptr<aont::Section>> &)>
4.          callback) {
5.    std::vector<std::unique_ptr<cryptofile::aont::Section>> sections;
6.    sections.reserve(sections_number);
7.    CryptoPP::AutoSeededRandomPool prng;
8.    CryptoPP::SecByteBlock key(
9.        CryptoPP::AES::MAX_KEYLENGTH);            // 32 bytes (256 bits)
10.   prng.GenerateBlock(key, key.size());
11.   /** We split the data in the number of section defined by the user. The
12.    * sections will have all the same size except for the last one that will
13.    * usually be slighty smaller.
14.    */
15.   double standard_section_size =
16.       static_cast<float>(data.size()) / static_cast<float>(sections_number);
```

```
  auto normal_section_size = std::ceil(standard_section_size);
  auto last_section_size =
      normal_section_size -
      ((normal_section_size * sections_number) - data.size());

  auto encrypted_key = std::vector<std::uint8_t>(&(key.BytePtr()[0]),
                                                 &(key.BytePtr()[key.size()]));
  for (std::size_t i = 0; i < sections_number; ++i) {
    auto section_size =
        i < sections_number - 1 ? normal_section_size : last_section_size;
    auto encrypt_section = std::make_unique<cryptofile::aont::EncryptSection>(
        gsl::span<std::uint8_t>(
           &(data.data()[static_cast<std::size_t>(normal_section_size) * i]),section_size),
           key);
    auto &hash = encrypt_section->get_hash();
    CryptoPP::xorbuf(encrypted_key.data(), encrypted_key.data(), hash.data(), hash.size());
    sections.emplace_back(std::move(encrypt_section));
  }
  sections.emplace_back(
      std::make_unique<cryptofile::aont::LastSection>(encrypted_key));
  callback(sections);
}
```

Aont_restore

```
void aont_restore(std::vector<std::vector<std::uint8_t>> sections_data,
                  std::function<void(std::vector<std::uint8_t> &)> callback) {
  std::vector<std::unique_ptr<aont::DecryptSection>> sections;
  sections.reserve(sections_data.size() - 1);
  std::vector<std::uint8_t> decrypted_key(
      &(sections_data.back().data()[0]),
      &(sections_data.back().data()[sections_data.back().size()]));
  for (std::size_t i = 0; i < sections_data.size() - 1; ++i) {
    sections.emplace_back(
        std::make_unique<aont::DecryptSection>(gsl::span<std::uint8_t>(
            &(sections_data[i].data()[0]), sections_data[i].size())));
    const auto &digest_hash = sections[i]->get_hash();
    CryptoPP::xorbuf(decrypted_key.data(), decrypted_key.data(),
                     digest_hash.data(), digest_hash.size());
  }
  std::vector<std::uint8_t> plain_text;
  CryptoPP::SecByteBlock key(decrypted_key.data(), decrypted_key.size()); //?
  for (auto &sec : sections) {
    sec->decrypt_data(key);
    plain_text.insert(plain_text.end(), sec->get_data().begin(),
                      sec->get_data().end());
  }
  callback(plain_text);
}
```

Both functions have a callback that allows us to keep the upload and download phases separated granting a great software flexibility and manageability.

From the *aont_mask's* callback we bring out of the function body the variable "sections" which is a vector of unique pointers to the Sections objects each containing the message, pseudo-message and hash string corresponding to the equivalent slice they represent.
On the other hand, with the *aont_restore*'s callback function we pass on the restored plaintext in order to store it and evaluate its integrity with better ease.
After each restoration we compare the checksum with the original file's one to state that the operation brought back the very same file with any loss of data.

The whole process exploits the already presented classes and organize them with the help of other two elements which are the *InitCryptoFile* and the *CryptoFileSession* files. As its name may suggests the first one handles the startup of the process as well as the terminal interfaced menu in order to get user's information to pass to the *CryptoFileSession* class, the latter taking care of the user's selected clouds' authorization and serving the menu functionalities such as the LIST, UPLOAD and DOWNLOAD operations. Therefore, we handle the *aont_mask* and *aont_restore* functions inside this class which, taking advantage of their callbacks, also stores the file either on the local system during the download phase or online if we are uploading slices.

Regarding the software usage and chronological flow, we are now going to show in few steps the actual execution of CryptoFile.

Starting up CryptoFile...

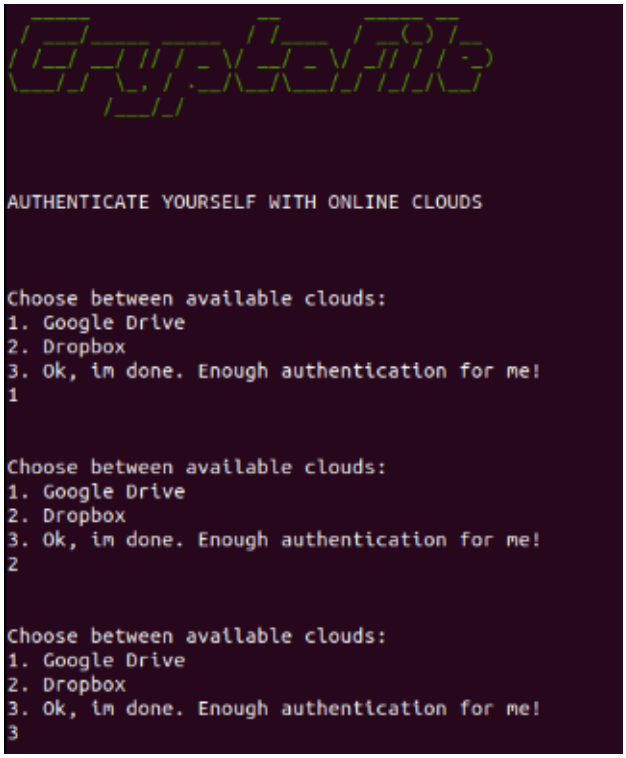

Figure 9 – CryptoFile's Start

➢ Firstly, the program asks which clouds the User wants to authenticate to. We are now inside the *InitCrytoFile* scope

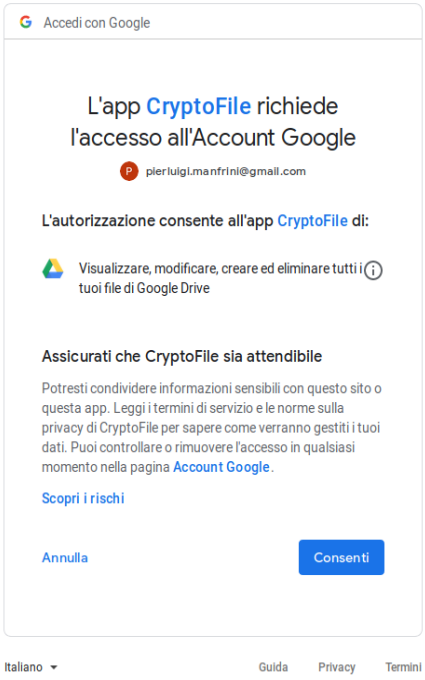

Figure 10 - Google Drive Authorization

➢ The browser will automatically open up in order to perform the oauth2 authentication with the chosen clouds.

Figure 11 - Dropbox's Authorization

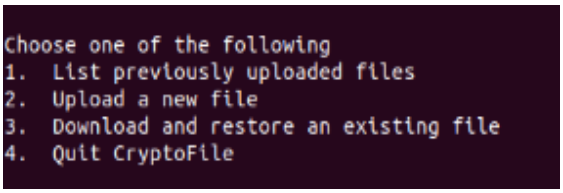

Figure 12 - CryptoFile's Menu

- After the authentication, menu is displayed. We are still inside the InitCryptoFile class which needs to get the user's input before using the CryptoFileSession to operate.

We are showing a sample upload and download of a text file of medium size. It will be partitioned into 7 slices, hence the total output will be composed of a total of 8 slices, the last one being the key holding slice.

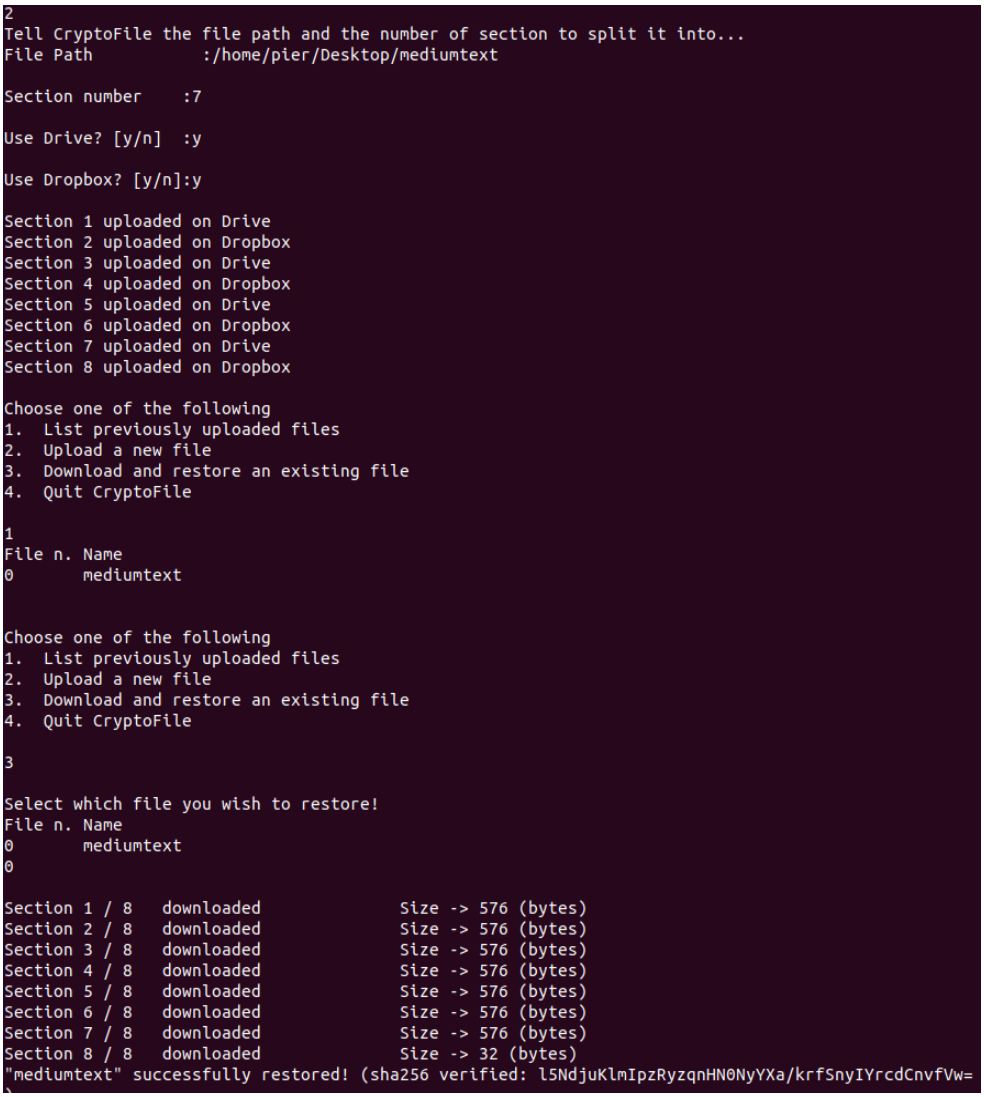

Figure 13 - Upload and Download Sample

- The slices are uploaded alternately on the chosen cloud storages in a shuffled order. Note that the size of the last section is, as it should be, 32 bytes. This is the length of the AES-256 Key and the hash digest we used (SHA-256). The final restored file is saved in "/tmp/CryptoFile/" and checked for integrity.

In order to better visualize the Classes' interactions, below we have a UML timeline scheme of an upload process (*Fig.14*) :

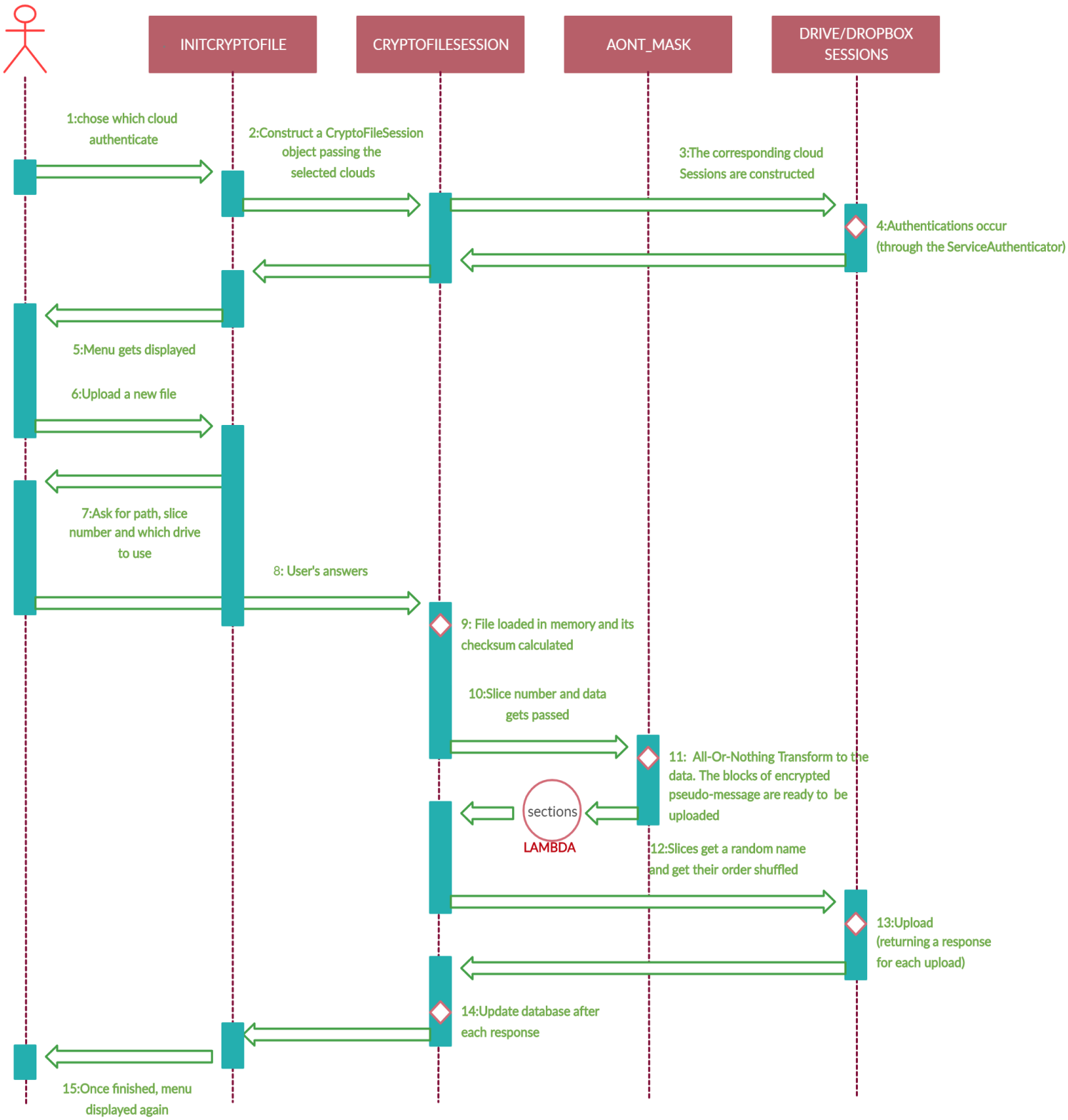

Figure 14 - Upload Flow UML

Chapter 4 ~ Aont's Efficiency ~

4.1 Brute Force Attacks

As previously shown, there are some situations in which we can not improve the security level of our data opting for a longer key since we may be constrained either by our encryption algorithm's structure or by some export regulations that restrict the key's length to a 40-bit.

This will lead to an increasing weakness and vulnerability which we could say being inversely proportional to the shortness of the key used to encrypt our private content. In such scenario this weakness will allow an attacker to attempt a brute force search for the key which our data are relying on to be kept secret.

Put in simple words, a brute force attack is a way of decrypting our ciphertext with all possible keys until the plaintext recovered makes sense; its most common applications are cracking passwords, encryption keys, SSH logins and API keys.

It literally simply tries using different combinations of characters until it can find the correct one. This really basic strategy makes brute force attacks at the same time really effective, because they always work, and really slow.

Their real effectiveness strongly depends on time that is what determines if a victorious attack is achievable or not, in fact, it is all a matter of how many combinations we can try per time unit.

In general, the work required to guess an unknown *k*-bit key of a known block cipher is 2^k total attempts for the worst case and 2^{k-1} on the average thus highlighting that the key variable for a successful brute force search is the computational power the attacker has.

To put this in perspective, let's consider the time needed to crack a 128-bit key which obliges the attacker to test 2 to the 128th power ($3.403 * 10^{38}$) total combinations:

- If we assume that the attack against this 128-bit key is carried on by every single person in the world concurrently (there are about 7 billion human beings on our planet)
- We assume even further that each person owns 10 computers each with a computational power of 1 billion keys per second
- Knowing also that on average we can crack a key after testing half (2^{127}) of the total 2^{128} possibilities

Then the whole earth's population will take $7.7 * 10^{25}$ years (*Fig.15*) to crack the 128-bit key which is a ridiculous amount of time, hence making the attack simply undoable.

Computation Reference for 128-Bit Key Crack Example	
People	7.00E+09
Computers per person	10.00
Computers	1.00E+09
Combos per second per computer	7.00E+19
Total combos per second	7.00E+19
Seconds per year	3.15E+07
Total combos per year	2.22E+12
128-bit key combos (*50%)	1.70E+38
Years to crack	7.66E+25

Figure 15 - Example's Computations

Below we show two interesting tables presented on the “ENCRYPT Yearly Report on Algorithms and Keysizes (2006)” :

➢

Attacker	Budget	Hardware	Min security
“Hacker”	0	PC	52
	< $400	PC(s)/FPGA	57
	0	“Malware”	60
Small organization	$10k	PC(s)/FPGA	62
Medium organization	$300k	FPGA/ASIC	67
Large organization	$10M	FPGA/ASIC	77
Intelligence agency	$300M	ASIC	88

Figure 16 - from Encrypt Yearly Report (2006)

This one (*Fig.16*) tells what should be the minimum keysizes (that will grant protection only for a few months) in relation to various kind of attackers.

FPGA : Field-Programmable Gate Array

ASIC : Application-Specific Integrated Circuit

➢

Security Level	Security (bits)	Protection	Comment
1.	32	Attacks in "real-time" by individuals	Only acceptable for auth. tag size
2.	64	Very short-term protection against small organizations	Should not be used for confidentiality in new systems
3.	72	Short-term protection against medium organizations, medium-term protection against small organizations	
4.	80	Very short-term protection against agencies, long-term prot. against small organizations	Smallest general-purpose level, $\leq$ 4 years protection
5.	96	Legacy standard level	Use of 2-key 3DES restricted to $\sim 10^6$ plaintext/ciphertexts, $\approx$ 10 years protection
6.	112	Medium-term protection	$\approx$ 20 years protection
7.	128	Long-term protection	Good, generic application-indep. recommendation, $\approx$ 30 years
8.	256	"Foreseeable future"	Good protection against quantum computers

Figure 17 - from Encrypt Yearly Report (2006) Table 7.4

This second table (*Fig.17*), instead, highlights the security level reached by the different key's lengths and, as you can notice, there are basically no reasons to choose, for example, an AES-256 over an AES-128, the latter being sufficient and even exceeding the nowadays safety requirements.

In their reference document, NITS (National Institute of Standards and Technology) has concluded that all the three key-lengths of AES (128 , 192 and 256 bits) will provide an adequate encryption until beyond the year 2031, not only against the key prediction threat (brute force) but also against any possible way of cracking the encryption algorithm itself. As there are no relevant differences between cracking the AES-128 algorithm and the AES-256 algorithm, there are no practical reasons to prefer

one to the other as they both grant a level of security way over the minimum nowadays requirements and whatever unseen breakthrough might crack 128-bit will probably also crack 256-bit.
The only real vulnerabilities hence regard the software implementation, the key storage system and the authentication process management.

4.2 AONT's benefits

From what we have read in the previous subchapter we may not have the necessity to enhance the encryption power of algorithms such AES-256, but we do have the real need to strengthen our cryptosystem if it relies on keys shorter than a certain size depending on who we want to be safe from.

As an example, if we consider two same plaintexts eight-megabytes in size and we encrypt them with a 40-bit key but applying the all-or-nothing transform before encrypting the second plaintext we raise its level of security by a factor of one million which is approximately 2^{20}. Therefore, the adversary has to test 2^{40} combination for the first plaintext and $2^{40} * 2^{20} = 2^{60}$ for the second, the equivalent of having to break a 60-bit key instead of a 40-bit key.
Since the Rivest's transform has a computational cost that is approximately twice of the cost of the encryption itself, its beneficiaries (we can identify them as the legitimate communicans during some private and encrypted data exchange) pay a total penalty of a factor of three, however gaining a huge advantage over the attacker who pays a penalty of a factor of *S* where *S* is the number of ciphertext blocks.

The following graphs show timing tests executed on our CryptoFile's AONT and AES implementation to see how the execution time depends on the number of slices asked as output

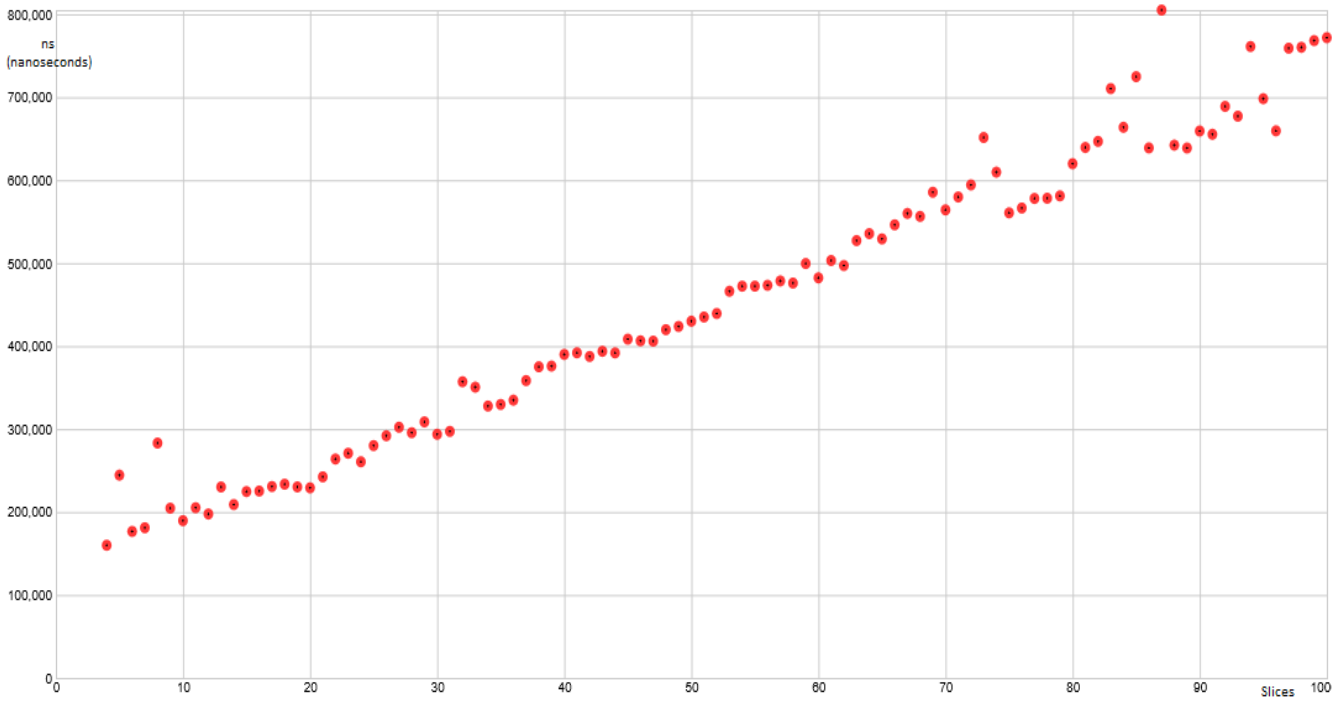

Figure 18 - aont_mask execution time

Here (*Fig.18*) we used, as the testing input file, a medium sized text (3951 bytes) and we took note of the *aont_mask* execution time in relation to the total number of output slices requested which range from 3 (2 slices partition + the extra final slice) to 100 on the X axis.

On the Y axis we reported the time needed for a complete execution in nanoseconds (1 nanosecond = 1 * 10^{-9} second) which reaches peaks of 800K nanoseconds at approximately 100 slices.

This test does not take into account the uploading phase since it would have strongly depended on our internet connection speed, so it basically consists in the execution

of the entire *aont_mask* function excluding its callback.

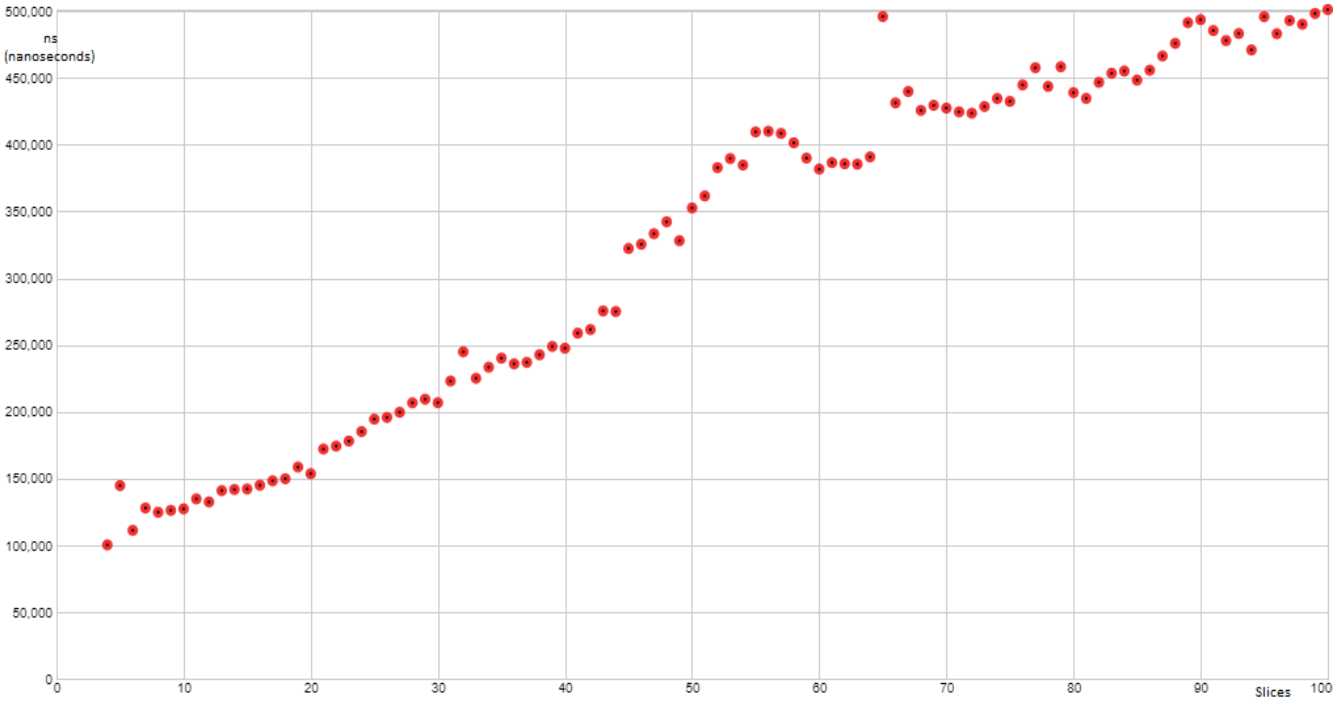

Figure 1911 - encryption time

On the other hand, here (*Fig.19*) we have the graph resulting from the timing of just the encryption phase which happens during the construction of the Sections objects we already discussed about.
Paying attention to the Y axis results we can tell that the AON-Transformation actually takes roughly double the time of the encryption alone.

All the tests were conducted on an intel i7-8550U quad-core CPU (1.80 – 4.00 GHz) relying on 16 GB 2400 MHz clocked RAM. The compiler used was gcc with still debug flags on, hence the execution times with a release version will surely be lower but proportions between AONT and encryption will be maintained.

Conclusions

In this paper we have presented the All-Or-Nothing package transform, as it was designed by Ronald L. Rivest in 1997, implemented together with the most notorious AES-256 encryption cipher in our project.
The whole CryptoFile program was an essential part of the AONT understanding and a significant opportunity to explore the world of cryptography and Cloud-based APIs in terms of practical implementation.

There are, nowadays, also other all-or-nothing transformations that will also grant a strongly non-separable encryption and even a higher level of security.
One even stronger implementation of the presented AONT is the AONT-RS where RS stands for Reed-Solomon: a group of error-correcting codes introduced by Irving S.Reed and Gustave Solomon in 1960.
The strategy resulting allows for a higher grade of both security and data availability since uses a wider degree of dispersion to make possible the information retrieval even when few servers are down.

Acknowledgments

I would like to thank my Supervisor Luca Spalazzi, an associate professor at the Università Politecnica delle Marche, who gave me the opportunity to conclude my studies with this project that was both extremely thrilling and challenging for me. He introduced me to the theoretical aspects that were constantly the core starting point during the CryptoFile's development phase.
Deserves a special thank Umberto Carletti, a close friend of mine and also a professional programmer, who took part in the project since he got as enthusiast as me when firstly heard about it. He owns the merit to made me grow significantly in software development and to have turned all the work into constructive fun.

References

- ‘Advanced Encryption Standard’. In *Wikipedia*, 29 September 2019. https://en.wikipedia.org/w/index.php?title=Advanced_Encryption_Standard&oldid=918677065.
- ‘All-or-Nothing Transform’. In *Wikipedia*, 14 May 2019. https://en.wikipedia.org/w/index.php?title=All-or-nothing_transform&oldid=897123214.
- ‘Block Cipher Mode of Operation’. In *Wikipedia*, 27 September 2019. https://en.wikipedia.org/w/index.php?title=Block_cipher_mode_of_operation&oldid=918228872.
- Dribbble. ‘Crypto File Icon Design’. Accessed 18 October 2019. https://dribbble.com/shots/3337217-Crypto-file-icon-design.
- ‘D.SPA.21-1.1.Pdf’. Accessed 18 October 2019. https://www.ecrypt.eu.org/ecrypt1/documents/D.SPA.21-1.1.pdf.
- Hardt, D. ‘The OAuth 2.0 Authorization Framework’. RFC Editor, October 2012. https://doi.org/10.17487/rfc6749.
- *Microsoft/Cpprestsdk*. C++. 2014. Reprint, Microsoft, 2019. https://github.com/microsoft/cpprestsdk.
- Dropbox. ‘OAuth Guide - Developers’. Accessed 18 October 2019. https://www.dropbox.com/developers/reference/oauth-guide.
- Resch, Jason K., and James S. Plank. ‘AONT-RS: Blending Security and Performance in Dispersed Storage Systems’. In *FAST*, 2011.
- ‘Riv97d.Prepub.Pdf’. Accessed 18 October 2019. https://people.csail.mit.edu/rivest/pubs/Riv97d.prepub.pdf.
- ‘Seagate128vs256.Pdf’. Accessed 18 October 2019. http://www.axantum.com/axcrypt/etc/seagate128vs256.pdf.
- Google Developers. ‘Using OAuth 2.0 to Access Google APIs | Google Identity Platform’. Accessed 18 October 2019. https://developers.google.com/identity/protocols/OAuth2.