VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF INFORMATION TECHNOLOGY
FALCUTY OF COMPUTER NETWORKS AND COMMUNICATIONS

FINAL REPORT

CRYTALS-DILITHIUM ALGORITHM AND APPLICATION ON DATA INTEGRITY VERIFICATION IN PUBLIC ADMINISTRATION.

Instructor: Ph.D.Nguyễn Ngọc Tự

Course: NT219.N22.ATCL

Group members: Huỳnh Đình Khải Minh - 21521123
Trần Thành Lợi - 21522296
Nguyễn Nguyễn Duy An - 21520536

Acknowledgement

We sincerely thank you for your guidance during our project. Your constructive feedback played an important role in improving our work on this project.

We highly appreciate your instruction and your time you dedicated to reviewing our project and giving us insightful suggestions.

Nguyễn Nguyễn Duy An - 21520536 - ATCL.2021
Huỳnh Đình Khải Minh - 21521123 - ATCL.2021
Trần Thành Lợi - 21522296 - ATCL.2021

Table of contents

1 Introduction

1.1 Overview

Nowaday, in the field of public adminstration, the publishing of important administrative documents on the internet has become widespread due to its convenience and accessibility. However, this convenience also raises concerns about the security and integrity of the documents. There is a risk of unauthorized modifications to the content or fraudulent impersonation of the publisher, leading to the dissemination of inaccurate or misleading information.

To address these challenges, the emergence of digital signature technology has provided a reliable solution. Digital signatures serve as a safeguard against unauthorized alterations to the original document and enable recipients to verify the identity of the signer. By digitally signing the documents, the integrity and authenticity of the information are preserved, and recipients can have confidence in the accuracy and source of the published documents.

Overall, implementing digital signature technology in the publication of administrative documents ensures the preservation of information integrity, enhances security, and mitigates the risk of fraudulent activities. It allows for the efficient and secure dissemination of important information, enabling recipients to access and verify documents from any location and device. As organizations embrace the benefits of digitalization, leveraging digital signatures becomes essential in maintaining the trust, reliability, and credibility of administrative documents in the digital realm of Industry 4.0.

1.2 Problem statement

In the realm of publishing documents on the internet, the utilization of cloud databases is widely embraced due to their convenience. However, alongside the benefits, there are security concerns that need to be identified and addressed with appropriate solutions.

1.2.1 Scenario

Department A needs to publish an important notification document regarding a new policy change. They want to ensure the authenticity and integrity of the document when it is accessed by the recipients. They generate a digital signature which is unique to the document and is created using Department A's private key. Department A uploads the document along with the digital signature to a cloud database for storage and easy access.

Recipients B, who have the necessary permissions, access the cloud database and locate the notification document. B selects the document for download and retrieves it from the cloud database. Once downloaded, the recipient extracts the digital signature from the downloaded document and proceeds to verify its authenticity and integrity.

The digital signature verification process checks the integrity of the document by confirming that it has not been altered or tampered with since the digital signature was applied. The recipient's verification process also includes validating the authenticity of the signer's identity. They inspect the QR code in the document, which contains information about the signer or their organization. The recipient compares the verified identity of the signer with the expected source. This could involve checking against a trusted registry or contacting the organization directly for verification.

If the digital signature is valid, the recipient can trust the document's integrity, knowing that it has not been altered since it was signed by Department A. They can also confirm the identity of the signer, ensuring the document's authenticity.

The recipient can rely on the information contained in the document for decision-making or further actions, knowing that it comes from a trusted source and has not been tampered with.

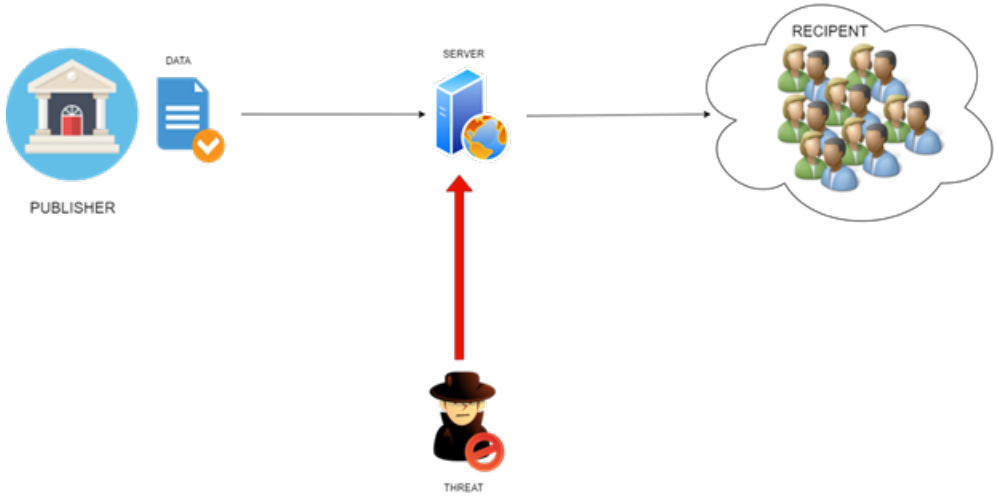

Figure 1: The signing process by the sender.

1.2.2 Related parties

In this scenario, the related parties are defined as follows:

- **Publishers**: These are individuals within the organization who are responsible for publishing documents to employees. They sign the PDF files and upload them to the server.
- **Cloud database server**: This server serves as the storage platform for the documents, as well as the repository for the published document, that include digital signature, and public key. It securely stores the documents and associated information.
- **Recipients**: These are the employees who are authorized to access the cloud database server and download the stored documents. They can retrieve the documents and perform verification checks on them.
- **Threat (Unrelated/Unauthorized party)**: This refers to individuals or entities who are not authorized to access or modify the documents. They pose a potential risk to the security and integrity of the published documents.

2 Proposed scheme

2.1 Overview of technique

We utilize the CRYTALS-DILITHIUM digital signature technology to generate the public key, digital signatures, and sign the files. These cryptographic operations provide a robust mechanism for ensuring the integrity and authenticity of the documents.

Once the files are signed using CRYTALS-DILITHIUM, they are ready to be published and securely stored. In this scenario, we employ a MongoDB cloud database server for document storage. MongoDB is a popular NoSQL database solution known for its scalability, flexibility, and document-oriented approach. The signed files, along with their corresponding digital signatures, are uploaded and stored within the MongoDB cloud database server. This server acts as a secure repository, preserving the integrity of the documents and associated cryptographic information.

The MongoDB cloud database server provides efficient and reliable storage capabilities, ensuring that the published documents are readily accessible to authorized recipients. Additionally, the server offers robust security features, including access controls, encryption, and authentication, to safeguard the stored data from unauthorized access or tampering.

By leveraging the CRYTALS-DILITHIUM digital signature technology and utilizing MongoDB as the cloud database server, we can establish a secure and trustworthy system for publishing and storing important documents. This approach enables recipients to verify the integrity and authenticity of the documents while benefiting from the scalability and reliability offered by the MongoDB cloud database platform.

2.2 Digital Signature

Firstly, we will discuss the fundamental operation of the digital signature algorithm, which comprises three stages: key generation, signing, and verification. The two primary components of this algorithm are the signing and verification of data.

- Key generation: A key generation algorithm that selects a private key uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.
- Signing:
  - A message that would be sent is first converted into a compact form called a message digest. Message digest (MD) is obtained by transforming message M using one-way hash function.

    $$MD = H(M)$$

  - Furthermore, the message digest (MD) is encrypted with the public key algorithm using a sender secret key (SK) into signature S

    $$\mathrm{S} = \boldsymbol{E}SK(MD)$$

  - Messages M is connected with signature S, then they are sent over through communication channel. In this case, we say that the message M has been signed by the sender with a digital signature S.
- Verifying:
  - After the message M and its digital signature S are transmitted through a communication channel and received by the receiver, the authenticity of the message is verified by decrypting the digital signature S with the sender's public key (PK). This decryption generates the original message digest, MD, according to the formula.

    $$\mathrm{MD} = \boldsymbol{D}PK(S)$$

  - The sender then converts message M into a message digest MD ' using the same one-way hash function. If MD'= MD, means that the received message is authentic and comes from the correct sender.

2.3 CRYSTALS-Dilithium

2.3.1 Overview

Dilithium is a digital signature scheme that is strongly secure under chosen message attacks based on the hardness of lattice problems over module lattices. The security notion means that an adversary having access to a signing oracle cannot produce a signature of a message whose signature he hasn't yet seen, nor produce a different signature of a message that he already saw signed. Dilithium is one of the candidate algorithms submitted to the NIST post-quantum cryptography project. Here is the template for the Crystal Dilithium digital signature algorithm.

Algorithm 1 Key Generator

$\mathbf{A} \leftarrow R_q^{k*l}$
$(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_n^l * S_n^k$
$\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$
return$(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1), \mathbf{s}_2))$

Algorithm 2 Sign(sk, M)

$\mathbf{z} := \perp$
while $\mathbf{z} = \perp$ **do**
    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^l$
    $\mathbf{w}1 :=$ **HighBits**$(\mathbf{Ay},\ 2\gamma_2)$
    $c \in B60 := \mathbf{H}(M \parallel \mathbf{w}_1)$
    $\mathbf{z} := \mathbf{y} + c\mathbf{s}1$
    if $\parallel \mathbf{z} \parallel \infty \geq \gamma_1 - \beta$ *or* $\parallel$ **LowBits**$(\mathbf{Ay} - c\mathbf{s}2, 2\gamma_2) \parallel \infty \geq \gamma_2 - \beta$ then z $:= \perp$
return $\sigma = (\mathbf{z}, c)$

Algorithm 3 Verify(pk, M, σ = (z, c))

$\mathbf{w}_1' :=$ **HighBits**$(\mathbf{Az} - c\mathbf{t},\ 2\gamma_2)$
if return $[\parallel \mathbf{z} \parallel_\infty < \gamma_1 - \beta]$ **and** $[c = H(M \parallel \mathbf{w}_1')$

2.3.2 Scientific Background

The design of Dilithium is based on the "Fiat-Shamir with Aborts" technique of Lyubashevsky which uses rejection sampling to make lattice-based Fiat-Shamir schemes compact and secure. The scheme with the smallest signature sizes using this approach is the one of Ducas, Durmus, Lepoint, and Lyubashevsky which is based on the NTRU assumption and crucially uses Gaussian sampling for creating signatures. Because Gaussian sampling is hard to implement securely and efficiently, we opted to only use the uniform distribution. Dilithium improves on the most efficient scheme that only uses the uniform distribution, due to Bai and Galbraith, by using a new technique that shrinks the public key by more than a factor of 2. To the best of our knowledge, Dilithium has the smallest public key + signature size of any lattice-based signature scheme that only uses uniform sampling.

2.3.3 Performance

The following table shows the performance of the Dilithium 3 algorithm we used when signing files of 1MB, 10MB, and 100MB. All benchmarks were obtained on one core of an AMD Ryzen 7 5800H CPU.

500 Iterations	1MB	10MB	100MB
KeyGen() Median Time	0.016s	0.016s	0.016s
Sign() Median Time	0.084s	0.108s	0.335s
Sign() Average Time	0.106s	0.127s	0.359s
Verify() Median Time	0.021s	0.045s	0.279s

3 Implementation

3.1 Overview

We have developed a program implementation using the Python language, utilizing the pymongo library for MongoDB integration and the Dilithium library for digital signature generation and verification:

- **Python**: We employ two main library.
  - **dilithium-py** library for digital signature argorithm
  - **Pymongo** library for connect to database
- **MongoDB**: being a popular NoSQL database solution, offers a flexible and scalable platform for storing and retrieving documents.

3.2 System design

In our system design contains 3 main nodes: Publisher, Cloud database service, Recipient

- **Publisher**: The publisher is responsible for issuing documents to the cloud. When publishing a document, it is signed and a QR code containing information about the publisher is generated.
- **Cloud Database Service**: The cloud service acts as a storage and access control node. It stores items such as documents with the publisher's signature and the publisher's public key. The cloud service manages access rights, allowing publishers to issue documents and recipients to access and download only the documents they are authorized to access.

- **Recipient**: The recipient receives the notification documents. They can access the cloud service and download the documents authorized for their access.

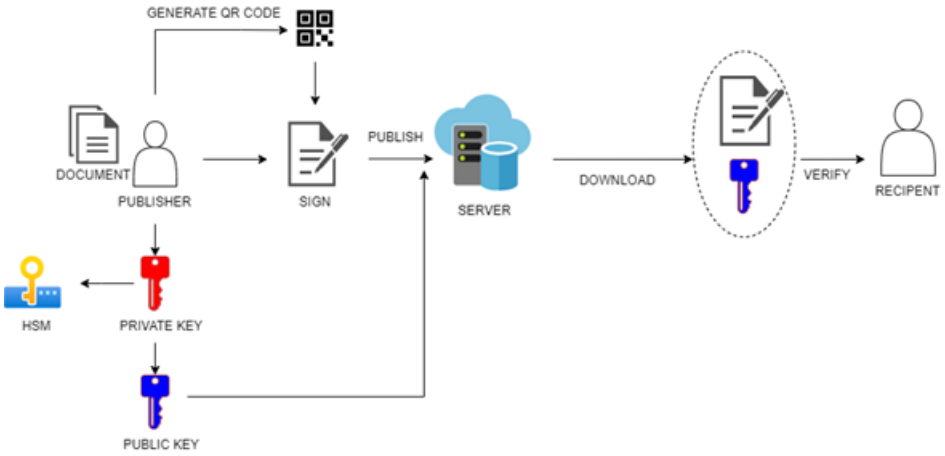

Figure 2: Demo the operation process of the program.

3.2.1 Database

3.2.1.a Storage file

Once the document needs to be published, it will be transformed into binary data and appended with a digital signature generated. Additionally, a QR code containing information about the publisher is created. The modified document, along with the digital signature, QR code, and the publisher's public key, is then stored on the MongoDB cloud server using GridFS. GridFS is a MongoDB specification that allows storing large-sized files in MongoDB. By utilizing GridFS, the stored data includes the digitally signed document, QR code, and the publisher's public key.

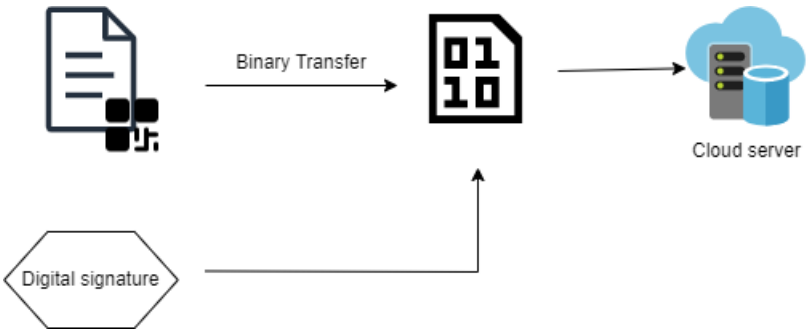

Figure 3: The format of the published file.

When using MongoDB's GridFS storage method, the files are stored in two parts: the fs.file section and the fs.chunks section.

- **The fs.file collection**: contains the meta data of the file, including essential information such as the file name, file size, upload date, and any additional metadata associated with the file.
- **The fs.chunks collection**: is responsible for storing the actual file content. It breaks down the file into smaller chunks, which are then stored as separate documents in the database. Each chunk is typically a fixed size, allowing for efficient retrieval and storage.

By dividing the file storage into these two sections, GridFS enables the efficient management and retrieval of large files in MongoDB, providing a reliable and scalable solution for storing and accessing files in a distributed environment.

Figure 4: The information of the file is stored in the fs.file collection.

Figure 5: The contents of the file are stored in the fs.chunks collection in bytes format.

3.2.1.b Access Management

Access management is supported by MongoDB. When creating a database, we can set permissions for users to control their access rights, such as modifying information, publishing documents within the database, or merely accessing and downloading documents.

In this case, we will configure access permissions for publishers to have full control over the documents they upload and manage. They will be granted the privilege of managing their own documents. However, for documents outside their management scope, publishers will have access rights similar to recipients. Recipients, on the other hand, will be granted access permissions limited to accessing, searching, and downloading the files they are authorized to view.

By implementing these access permissions, we ensure that publishers have the necessary control over their own documents while recipients are granted appropriate access rights to securely access and download relevant documents.

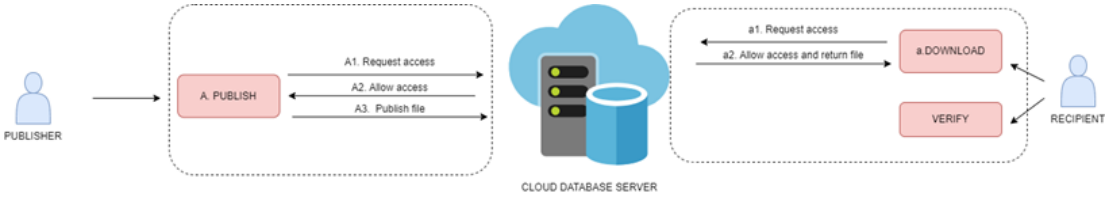

Figure 6: Execution flow of access request.

3.2.2 Publisher

3.2.2.a Generate digital signature & sign file

We will generate a key pair consisting of a public key and a secret key using the Dilithium3 algorithm. The secret key will then be combined with the Dilithium3 signature algorithm to create a digital signature for the PDF file.

Listing 1: Project.py

```
def PublisherPermission():
    try:
        doccument = user_collection.find_one({"username": account})
        if(doccument["role"] == "0"):
            pk, sk = Dilithium3.keygen()
            print("File path:")
            path =input()
            print("File name:")
            file_name = input()
            print("Waiting for publish...")
            path = makePdf(path, makeWatermark())
            with open(path, "rb") as file:
                pdf_file = file.read()
            sig = Dilithium3.sign(sk, pdf_file)
            pkh_ex = binascii.hexlify(pk).decode('utf-8')
            now = datetime.now()
            dt_string = now.strftime("%d/%m/%Y")

            # Upload signature to MongoDB

            fs = gridfs.GridFS(file_collection.database)
            with open(path, "ab") as f:
                f.write(sig)
            with open(path, "rb") as file:
                pdf_file = file.read()
            file_id = fs.put(pdf_file, filename = file_name, publisher =
                account,Date = dt_string, publickey = pkh_ex )
            with open(file_name+"_signed.pdf", "wb") as f:
                f.write(pdf_file)
            print("Published!")
        else:
            print("You do not have permission to do that.")

    except Exception as e:
        print(e)
        print("Duong dan khong hop le")
        PublisherPermission()
```

The PDF file will be created by selecting a chosen PDF file and marking a QR code on it. The QR code will include the name of the publisher and the timestamp of the signature. The file will be opened and read in binary mode. The data of the file will be combined with the publisher's secret key to generate a digital signature for the file. This digital signature will be inserted at the end of the PDF file. Afterward, we will use MongoDB's GridFS to publish the file to the database, along with additional information such as the filename, publisher's information, and their public key.

3.2.2.b QR code generate.

The QR code include name of publisher and the timestamp of the signature. This QR code will be signed onto the first page of the PDF file.

Listing 2: Project.py

```
def makeWatermark():
    watermarkName = "qr.pdf"
    doc = canvas.Canvas(watermarkName)
    qr = QRCodeImage(
        size=30 * mm,
        fill_color='black',
        back_color=(0, 0, 0, 0),
        border=4,
        version=2,
        error_correction=qrcode.constants.ERROR_CORRECT_H,
    )
    now = datetime.now()
    dt_string = now.strftime("%d/%m/%Y %H:%M:%S")
    qr.add_data("Signed by: "+account+ "\n"+"Day/time: "+ dt_string)
    qr.drawOn(doc, 20 * mm, 40 * mm)
    doc.save()
    return watermarkName

def makePdf(src, watermark):
    merged = src + "_signed.pdf"
    with open(src, "rb") as input_file, open(watermark, "rb") as
        watermark_file:
        input_pdf = PdfReader(input_file)
        watermark_pdf = PdfReader(watermark_file)
        watermark_page = watermark_pdf.pages[0]
        output = PdfWriter()
        #Sign in first page
        for i in range(len(input_pdf.pages)):
            if(i==0):
                pdf_page = input_pdf.pages[i]
                pdf_page.merge_page(watermark_page)
                output.add_page(pdf_page)
            else:
                pdf_page = input_pdf.pages[i]
                output.add_page(pdf_page)


        with open(merged, "wb") as merged_file:
            output.write(merged_file)
        return merged
```

3.2.3 Recipient

3.2.3.a Search & Download

When a recipient uses the "/download" command, they will have the privilege to search for the files they need and then download the file. This file includes the signed document, the publisher's public key, and other relevant information provided to enable users to perform integrity verification and identity authentication of the file.

When the search function is executed, the files will be iterated once to search for files that match the information entered by the user for searching.

Listing 3: Project.py

```
def download_file():
    list_files()
    global account
    print("File name:")
    file_name = input()
    fs = gridfs.GridFS(file_collection.database)
    user = user_collection.find_one({"username": account })
    file = fs.find_one({"filename": file_name})
    if file:
        with open(file_name+"sign.pdf", "wb") as f:
            f.write(file.read())
        print("Downloaded successfully!")
    else:
        print("File not found!")
        download_file()

def list_files():
    print("Available files:")
    for i, document in enumerate(file_collection.find(), 1):
        file_name = document["filename"]
        file_date = document["uploadDate"]
        print(f"{i}: {file_name} (Uploaded on: {file_date})")

def search():
    query = input()
    pipeline = [
        {
            "$search": {
                "index": "Searching",
                "text": {
                    "query": query,
                    "path": {
                        "wildcard": "*",
                    }

                }
            }
        },
        {
            "$project": {
                "_id": 0,
                "filename": 1,
                "Date": 1
            }
        }
]
    results = file_collection.aggregate(pipeline)
    for i, doc in enumerate(list(results), start=1):
```

```
49          print(f"{i}: {doc}")
```

3.2.3.b Verify the doccument

When a recipient uses the "/verify" command, since the digital signature is inserted at the end of the file, and in the dilithium algorithm, the size of the converted digital signature is default to 3,293 bytes. Therefore, we will extract the digital signature and perform verification using the dilithium3 algorithm, along with the original file data and the publisher's public key, which is also downloaded along with the file. The function will output the verification result.

Listing 4: Project.py

```
1  def RecepientPermission():
2      print("File path:")
3      path = input()
4      flag = 0
5      try:
6          for document in file_collection.find():
7              public_key = binascii.unhexlify(document["publickey"])
8              with open(path, "rb") as file:
9                  file.seek(0, os.SEEK_END)
10                 file_size = file.tell()
11                 file.seek(0, os.SEEK_SET)
12                 pdf_file = file.read(file_size - 3293)
13                 file.seek(-3293, 2)
14                 signature = file.read()
15             verify = Dilithium3.verify(public_key, pdf_file, signature)
16             if(verify == True):
17                 print(f"Verification result for signature {document['_id']}: {
                       verify}")
18                 flag = 0
19                 break
20             else:
21                 flag+=1
22         if(flag != 0):
23             print("false")
24
25     except Exception as e:
26         print(e)
27         print("File khong hop le")
28         RecepientPermission(file_collection)
```

4 Demo application

4.1 Publisher

The Publisher will sign the file and publish it to the cloud database. The signed file will include a QR code containing the date, time, and publisher information. The digital signature will be stored in the last 3,293 bytes of the signed file for authentication purposes.

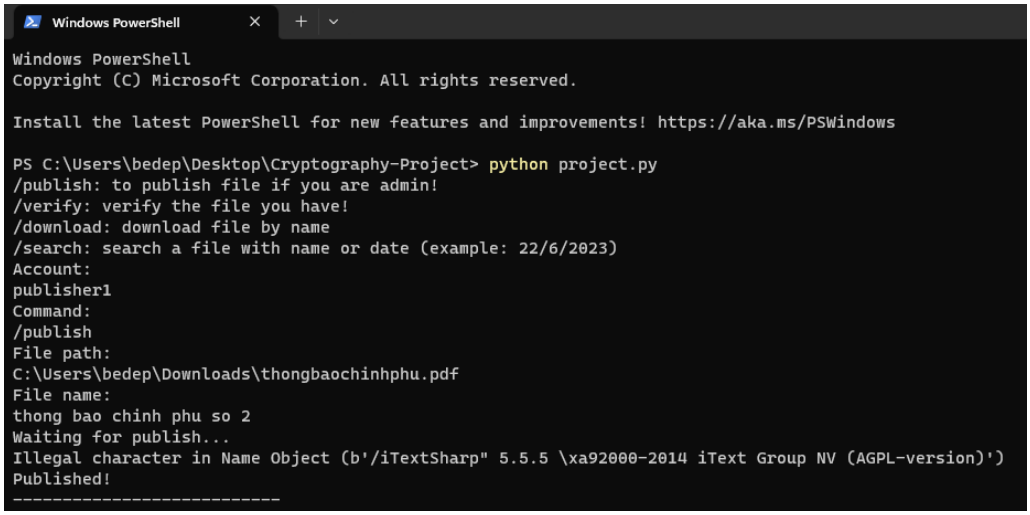

Figure 7: The Publisher performs the signing process and publish the file to the database.

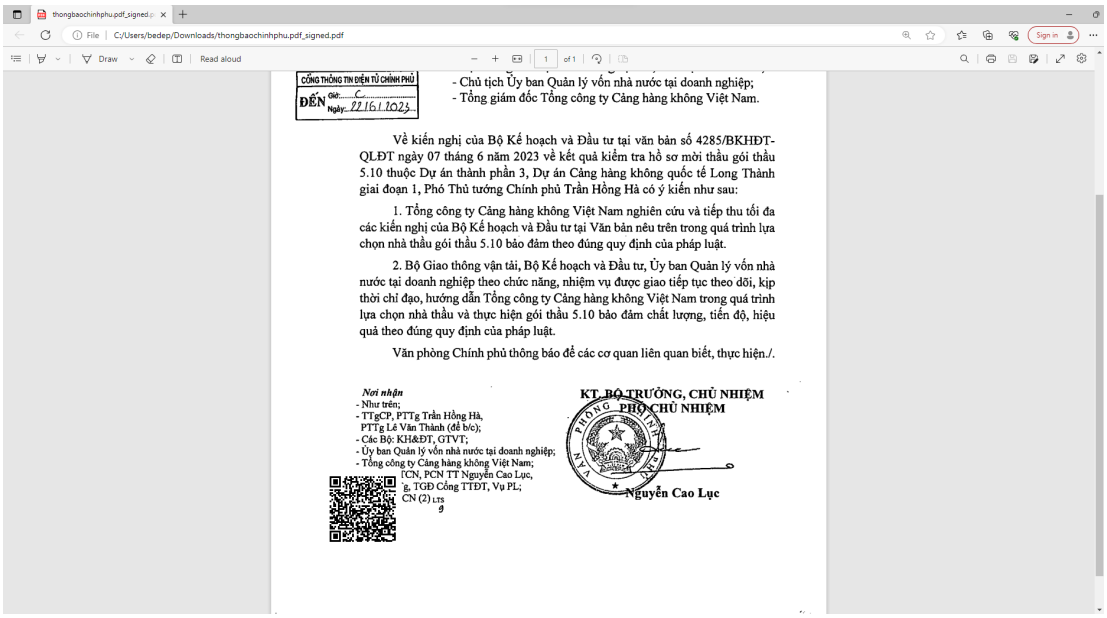

Figure 8: The signed file contains a QR code.

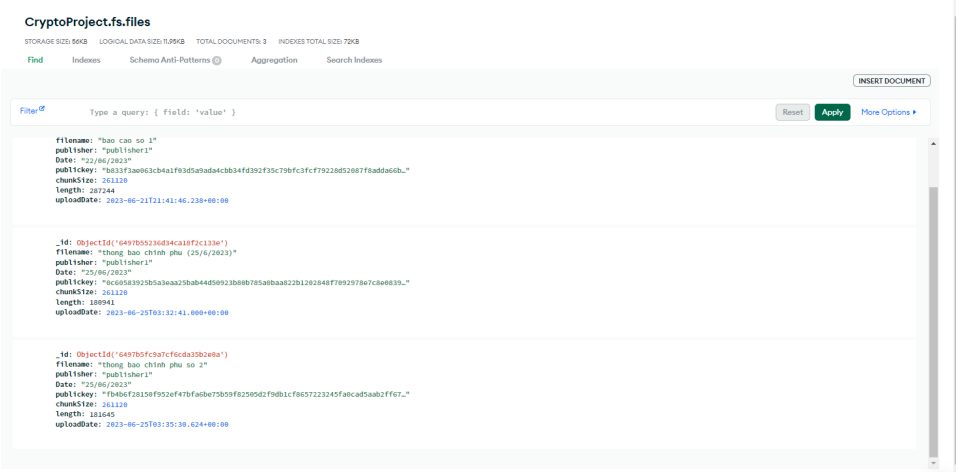

Figure 9: The data and information of the file are stored in the database.

4.2 Recipient

The recipient can perform operations such as searching, downloading, and verifying the downloaded file.

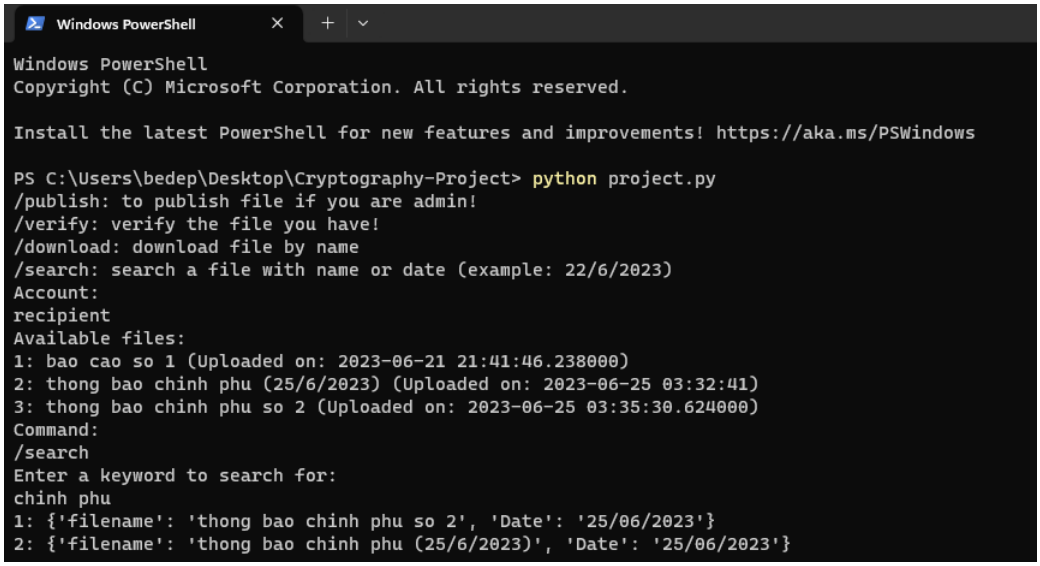

Figure 10: Searching file with name.

```
/publish: to publish file if you are admin!
/verify: verify the file you have!
/download: download file by name
/search: search a file with name or date (example: 22/6/2023)
Account:
recipient1
Available files:
1: bao cao so 1 (Uploaded on: 2023-06-21 21:41:46.238000)
2: thong bao chinh phu (25/6/2023) (Uploaded on: 2023-06-25 03:32:41)
3: thong bao chinh phu so 2 (Uploaded on: 2023-06-25 03:35:30.624000)
Command:
/download
Enter file name to download:
thong bao chinh phu so 2
Downloaded successfully!
```

Figure 11: The download of a file.

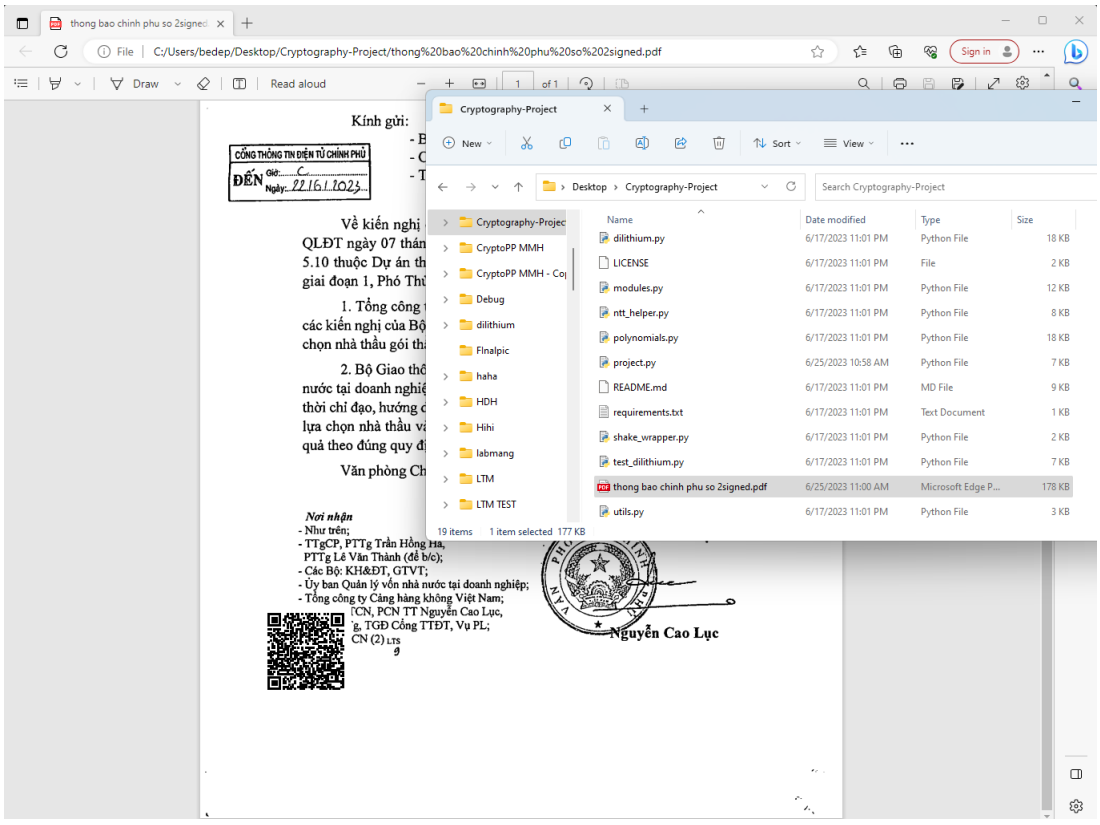

Figure 12: The file has been downloaded.

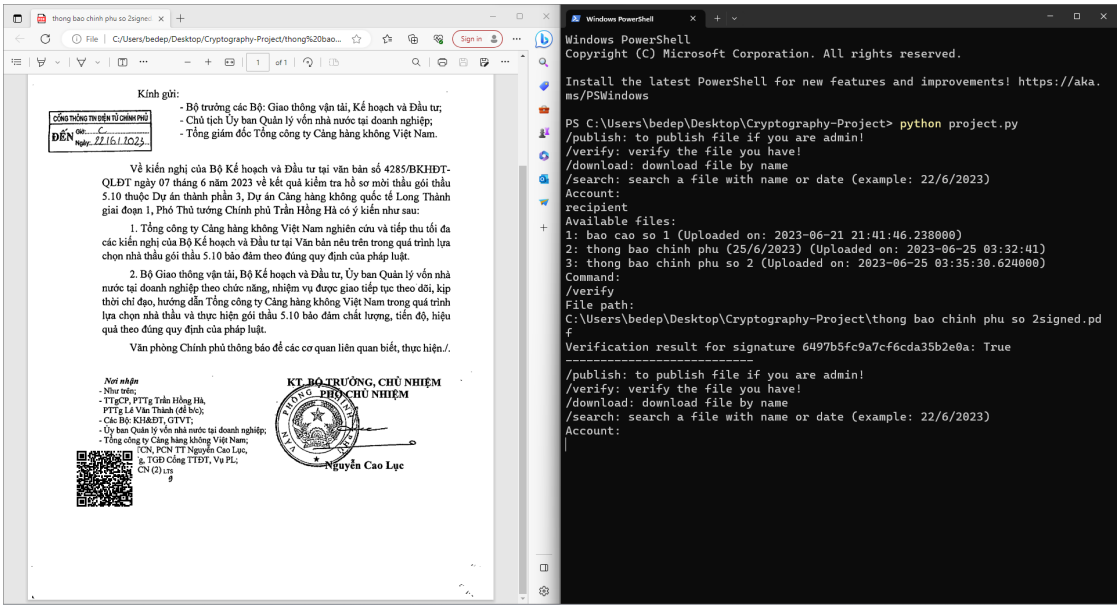

Figure 13: Verification with the downloaded file.

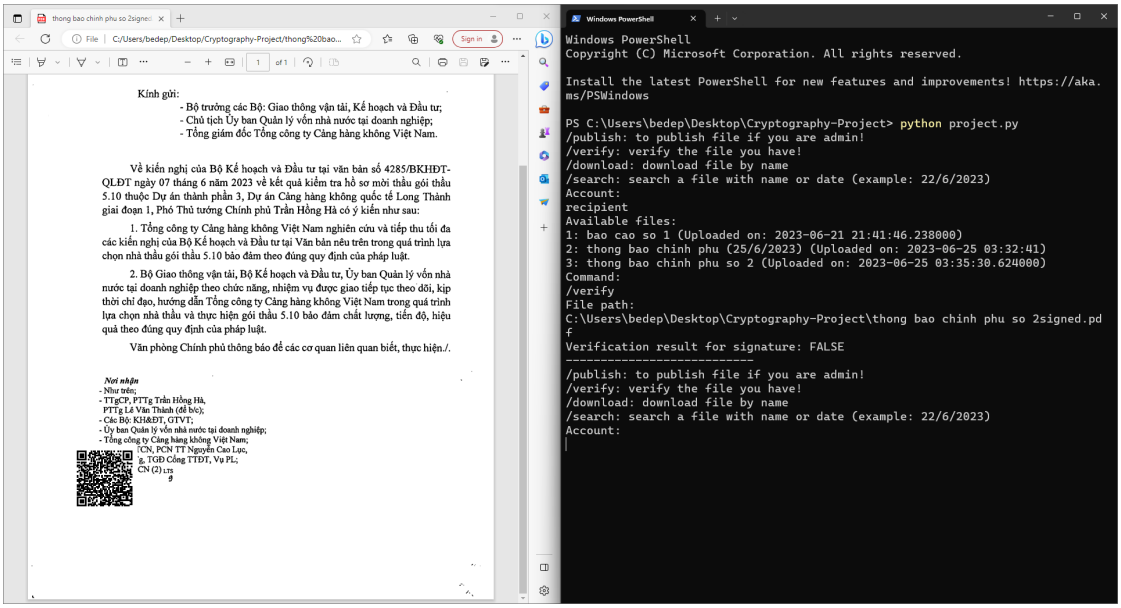

Figure 14: Verification with the modified file.

5 References

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., & Stehlé, D. (2017). Crystals-dilithium algorithm specifications and supporting documentation.

Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schwabe, P., Seiler, G., & Stehlé, D. (2018). Crystals-dilithium: A lattice-based digital signature scheme. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, *2018*, 238–268.

Kaur, R., & Kaur, A. (2012). Digital signature. *2012 International Conference on Computing Sciences*, 295–301. https://doi.org/10.1109/ICCS.2012.25